Sadržaj

Predgovor ix

1 Zašto softversko inženjerstvo? **1**
1.1 Šta je softversko inženjerstvo 2
1.2 Koliko smo bili uspešni? 5
1.3 Šta je dobar softver? 9
1.4 Ko se bavi softverskim inženjerstvom? 14
1.5 Sistemski prilaz 16
1.6 Inženjerski pristup 21
1.7 Članovi razvojnog tima 25
1.8 Kako se izmenilo softversko inženjerstvo 27
1.9 Primer iz informacionog sistema 36
1.10 Primer iz prakse 37
1.11 Šta ovo poglavlje znači za vas 39
1.12 Šta ovo poglavlje znači za razvojni tim 40
1.13 Šta ovo poglavlje znači za istraživače 40
1.14 Seminarski rad 40
1.15 Ključne reference 42
1.16 Vežbe 43

2 Modelovanje procesa i životnog ciklusa **45**
2.1 Značenje termina proces 45
2.2 Modeli procesa izrade softvera 48
2.3 Alati i tehnike za modelovanje procesa 63
2.4 Praktično modelovanje procesa 70
2.5 Primer iz informacionog sistema 73
2.6 Primer iz prakse 75
2.7 Šta ovo poglavlje znači za vas 76
2.8 Šta ovo poglavlje znači za razvojni tim 76
2.9 Šta ovo poglavlje znači za istraživače 77
2.10 Seminarski rad 77
2.11 Ključne reference 77
2.12 Vežbe 80

3 Planiranje projekta i upravljanje njime **82**
3.1 Praćenje napretka 82
3.2 Osoblje na projektu 95

3.3 Procena napora 104
3.4 Upravljanje rizicima 119
3.5 Plan projekta 123
3.6 Modeli procesa i upravljanje projektom 125
3.7 Primer iz informacionog sistema 133
3.8 Primer iz prakse 135
3.9 Šta ovo poglavlje znači za vas 136
3.10 Šta ovo poglavlje znači za razvojni tim 136
3.11 Šta ovo poglavlje znači za istraživače 137
3.12 Seminarski rad 137
3.13 Ključne reference 137
3.14 Vežbe 138

4 Evidentiranje zahteva 141
4.1 Proces vezan za zahteve 142
4.2 Izvođenje zahteva 144
4.3 Tipovi zahteva 148
4.4 Karakteristike zahteva 155
4.5 Notacije modela 156
4.6 Zahtevi i specifikacioni jezici 187
4.7 Izrada prototipa zahteva 191
4.8 Dokumentovanje zahteva 193
4.9 Validacija i verifikacija 198
4.10 Merenje zahteva 204
4.11 Izbor tehnike za specifikaciju 206
4.12 Primer iz informacionog sistema 209
4.13 Primer iz prakse 211
4.14 Seminarski rad 215
4.15 Ključne reference 218
4.16 Vežbe 219

5 Dizajniranje sistema 223
5.1 Šta je to dizajn? 223
5.2 Raščlanjivanje i modularnost 226
5.3 Arhitektonski stilovi i strategije 229
5.4 Pitanja u kreiranju dizajna 237
5.5 Karakteristike dobrog dizajna 248
5.6 Tehnike za poboljšanje dizajna 259
5.7 Procena i provera valjanosti dizajna 267
5.8 Dokumentovanje dizajna 278
5.9 Primer iz informacionih sistema 279
5.10 Primer iz prakse 281
5.11 Šta ovo poglavlje znači za vas 282
5.12 Šta ovo poglavlje znači za razvojni tim 282
5.13 Šta ovo poglavlje znači za istraživače 283
5.14 Seminarski rad 283

5.15 Ključne reference 283
5.16 Vežbe 284

6 Razmatranje objekata 286
6.1 Šta je OO? 286
6.2 OO proces razvoja 291
6.3 Slučajevi korišćenja 294
6.4 Predstavljanje oo: primer upotrebe uml-a 298
6.5 OO dizajn sistema 300
6.6 OO dizajn programa 315
6.7 OO merenja 322
6.8 Primer iz informacionih sistema 332
6.9 Primer iz prakse 334
6.10 Šta ovo poglavlje znači za vas 334
6.11 Šta ovo poglavlje znači za razvojni tim 335
6.12 Šta ovo poglavlje znači za istraživače 335
6.13 Seminarski rad 335
6.14 Ključne reference 335
6.15 Vežbe 336

7 Pisanje programa 337
7.1 Standardi i procedure programiranja 337
7.2 Smernice za programiranje 340
7.3 Dokumentacija 351
7.4 Proces programiranja 355
7.5 Primer iz informacionih sistema 359
7.6 Primer iz prakse 361
7.7 Šta ovo poglavlje znači za vas 362
7.8 Šta ovo poglavlje znači za razvojni tim 362
7.9 Šta ovo poglavlje znači za istraživače 362
7.10 Seminarski rad 363
7.11 Ključne reference 363
7.12 Vežbe 363

8 Testiranje programa 365
8.1 Greške i otkazi softvera 365
8.2 Pitanja testiranja 371
8.3 Jedinično testiranje 376
8.4 Integraciono testiranje 390
8.5 Testiranje objektno orijentisanih sistema 397
8.6 Planiranje testa 400
8.7 Automatizovani alati za testiranje 402
8.8 Kada treba prestati sa testiranjem 405
8.9 Primer iz informacionih sistema 410
8.10 Primer iz prakse 411

8.11 Šta ovo poglavlje znači za vas 412
8.12 Šta ovo poglavlje znači za razvojni tim 412
8.13 Šta ovo poglavlje znači za istraživače 413
8.14 Seminarski rad 413
8.15 Ključne reference 413
8.16 Vežbe 414

9 Testiranje sistema 417
9.1 Principi testiranja sistema 417
9.2 Funkcionalni test 430
9.3 Testiranje performanse 436
9.4 Pouzdanost, dostupnost i mogućnost održavanja 438
9.5 Test prihvatanja 447
9.6 Instalacioni test 450
9.7 Automatizovano testiranje sistema 450
9.8 Test dokumentacija 451
9.9 Testiranje sistema kod kojih je bezbednost kritična osobina 462
9.10 Primer iz informacionih sistema 473
9.11 Primer iz prakse 475
9.12 Šta ovo poglavlje znači za vas 477
9.13 Šta ovo poglavlje znači za razvojni tim 477
9.14 Šta ovo poglavlje znači za istraživače 477
9.15 Seminarski rad 478
9.16 Ključne reference 478
9.17 Vežbe 479

10 Isporučivanje sistema 483
10.1 Obuka 483
10.2 Dokumentacija 488
10.3 Primer iz informacionih sistema 495
10.4 Primer iz prakse 496
10.5 Šta ovo poglavlje znači za vas 496
10.6 Šta ovo poglavlje znači za razvojni tim 496
10.7 Šta ovo poglavlje znači za istraživače 497
10.8 Seminarski rad 497
10.9 Ključne reference 497
10.10 Vežbe 497

11 Održavanje sistema 499
11.1 Sistem koji se menja 499
11.2 Priroda održavanja 507
11.3 Problemi održavanja 510
11.4 Merenje karakteristika održavanja 518
11.5 Tehnike i alati za održavanje 524
11.6 Podmlađivanje softvera 532

11.7 Primer iz informacionih sistema 538
11.8 Primer iz prakse 539
11.9 Šta ovo poglavlje znači za vas 540
11.10 Šta ovo poglavlje znači za razvojni tim 540
11.11 Šta ovo poglavlje znači za istraživače 541
11.12 Seminarski rad 541
11.13 Ključne reference 541
11.14 Vežbe 542

12 Procenjivanje proizvoda, procesa i resursa 544
12.1 Pristupi proceni 544
12.2 Izbor tehnike za procenu 549
12.3 Procena u odnosu na predviđanje 553
12.4 Procenjivanje proizvoda 559
12.5 Procenjivanje procesa 574
12.6 Procenjivanje resursa 590
12.7 Primer iz informacionih sistema 595
12.8 Primer iz prakse 596
12.9 Šta ovo poglavlje znači za vas 596
12.10 Šta ovo poglavlje znači za razvojni tim 597
12.11 Šta ovo poglavlje znači za istraživače 597
12.12 Seminarski rad 597
12.13 Ključne reference 598
12.14 Vežbe 598

13 Poboljšanje predviđanja, proizvoda, procesa i resursa 600
13.1 Poboljšavanje predviđanja 601
13.2 Poboljšavanje proizvoda 609
13.3 Poboljšanje procesa 613
13.4 Poboljšavanje resursa 621
13.5 Opšte smernice za poboljšanje 623
13.6 Primer iz informacionih sistema 625
13.7 Primer iz prakse 626
13.8 Šta ovo poglavlje znači za vas 626
13.9 Šta ovo poglavlje znači za razvojni tim 626
13.10 Šta ovo poglavlje znači za istraživače 627
13.11 Seminarski rad 627
13.12 Ključne reference 628
13.13 Vežbe 628

14 Budućnost softverskog inženjerstva 629
14.1 Kako smo radili 629
14.2 Prenos tehnologije 632
14.3 Donošenje odluka u softverskom inženjerstvu 641
14.4 Profesionalizacija softverskog inženjerstva: licenciranje, sertifikacija i etika 653

14.5 Seminarski rad 671
14.6 Ključne reference 671
14.7 Vežbe 672

Bibliografija sa napomenama 673

Indeks 705

Predgovor

UPOTREBA MODELOVANJA I PROJEKTOVANJA ZA PREMOŠĆAVANJE JAZA IZMEĐU ISTRAŽIVANJA I PRAKSE

Softversko inženjerstvo je prešlo dug put od 1968, kada je kao pojam prvi put upotrebljeno na konferenciji NATO-a. Pre desetak godina samo malobrojni su mogli da predvide na koje će sve načine softver ući u naš život. Zbog toga je čvrsto teorijsko i praktično utemeljenje softverskog inženjerstva neophodno za razumevanje načina izrade dobrog softvera, kao i za procenjivanje rizika i mogućnosti koje softver unosi u naš svakodnevni život. Ovaj tekst predstavlja kombinaciju dva savremena pogleda na softversko inženjerstvo: pogleda praktičara, čiji je osnovni cilj gradnja funkcionalno iskoristivog proizvoda visokog nivoa kvaliteta, i pogleda istraživača, koji stremi pronalaženju puteva za unapređenje kvaliteta proizvoda i produktivnosti onih koji u njegovoj izgradnji učestvuju. Edsgar Dykstra nas je neprestano podsećao da strogost u istraživanju i praksi iskušava naše razumevanje softverskog inženjerstva i pomaže nam da unapredimo vlastiti način razmišljanja, pristupe i, u krajnjem cilju, naše proizvode. U tom duhu je naša knjiga i unapređena, izgradnjom okvira za ove rasprave i poboljšanja.

Modelovanje i projektovanje predstavljaju ključne pojmove za formulisanje okvira i premošćavanje jaza između istraživanja i prakse. Softverski inženjeri nisu samo programeri koji slede instrukcije, oni su više šefovi kuhinje nego obični kuvari koji slede recepte. Izrada dobrog softvera sadrži elemente umetnosti koja je oličena u razumevanju načina uopštavanja (apstrahovanja) i modelovanja suštinskih elemenata problema, a zatim korišćenju takvih apstrakcija za projektovanje rešenja. Često čujemo da kvalitetni projektanti govore o „elegantnim" rešenjima, podrazumevajući pri tome da su takva rešenja usmerena na suštinu problema na takav način da softver ne samo rešava konkretan problem, već se menja u skladu sa evolucijom samog problema. Zbog toga, ovo treće izdanje sadrži obimnu građu u vezi s apstrahovanjem i modelovanjem problema, i korišćenjem modela za projektovanje adekvatnih rešenja. Na taj način, studenti uče da kombinuju istraživanje sa praksom i umetnost sa naukom, u cilju izgradnje kvalitetnog softvera.

Nauka je uvek utemeljena na stvarnosti. Namenjena nastavnom planu predmeta iz oblasti softverskog inženjerstva na studijama, ova knjiga daje pragmatičnu sliku odnosa istraživanja i prakse u okviru softverskog inženjerstva. Služeći se njome, studenti mogu da ono što su naučili neposredno primene kada pokušavaju da pronađu rešenje nekih problema iz stvarnog života. Primeri, iako prilagođeni ograničenom iskustvu studenata, jasno pokazuju kako složeni softverski projekti prelaze put od ideje do realizacije. Oni prikazuju situacije u kojima će se čitaoci verovatno kad-tad naći: velike i male projekte, agilne i visoko strukturisane metode, objektno orijentisane i proceduralne pristupe, rad u realnom vremenu i transakcionu obradu, razvoj i održavanje.

Ova knjiga je takođe pogodna i za postdiplomske studije, gde se pruža uvod u koncepte i praksu softverskog inženjerstva, a takođe je korisna u praktičnom radu onima koji žele da prošire svoje znanje u izloženoj oblasti. Materija izneta u poglavljima 12, 13 i 14 predstavlja poseban izazov unet sa ciljem da zaintrigira postdiplomce u sklopu njihovih istraživačkih projekata.

KLJUČNE OSOBINE

Ovo štivo ima mnoge ključne osobine koje ga razlikuju od ostalih knjiga.

- Za razliku od ostalih knjiga o softverskom inženjerstvu, koje razmatraju merenje i modelovanje kao nezavisne teme, ova knjiga kombinuje merenje i modelovanje sa uopštenijom raspravom o softverskom inženjerstvu. Drugim rečima, merenje i modelovanje se smatraju integralnim delom strategija softverskog inženjerstva, umesto da se posmatraju kao zasebne discipline. Na taj način studenti uče kako da apstrahuju i modeluju, i pri tome uključuju kvantitativno ocenjivanje i poboljšanja u svoje dnevne aktivnosti. Mogu da koriste svoje modele za razumevanje važnih elemenata problema koji rešavaju, kao i alternativna rešenja. Mogu da koriste merenja za procenjivanje postignutog napretka na nivou pojedinca, grupe ili celog projekta.
- Na isti način, koncepti kao što su višekratna upotrebljivost, upravljanje rizicima i inženjerstvo kvaliteta, ne razmatraju se posebno nego su direktno uključeni u relevantne aktivnosti softverskog inženjerstva.
- U ovom izdanju razmatramo korišćenje agilnih metoda, uključujući i ekstremno programiranje, opisujemo dobit i rizik koji sobom nosi davanje veće autonomije učesnicima u razvoju, i takvu agilnost suprotstavljamo tradicionalnijim pristupima razvoja softvera.
- Koncepti izloženi u svakom poglavlju ilustrovani su sa dva karakteristična primera: prvi je tipičan informacioni sistem, a drugi je sistem sa radom u realnom vremenu. Oba primera zasnovana su na realnim projektima. Primer iz informacionog sistema opisuje softver potreban za određivanje cene oglašavačkog vremena velike britanske televizijske kompanije. Sistem koji radi u realnom vremenu je upravljački softver rakete Ariane-5. Videćemo kakvi se sve problemi ovde javljaju i istražićemo kako tehnike softverskog inženjerstva mogu da pomognu za pronalaženje i otkrivanje nekih od tih problema. Studenti mogu da prate napredak ova dva tipična projekta i da pri tome uoče kako se različite praktične metode, opisane u knjizi, povezuju sa tehnologijama koje se koriste za gradnju sistema.
- Na kraju svakog poglavlja, data je rekapitulacija gradiva sa tri aspekta: šta sadržaj poglavlja znači razvojnom timu, šta pojedinačnim učesnicima u razvoju, a šta istraživačima. Studenti mogu jednostavno da naprave pregled bitnih delova svakog poglavlja i ocene relevantnost poglavlja sa aspekta istraživačkog i praktičnog rada.
- Ovu knjigu prati veb stranica, do koje se može doći preko veb adrese Prentice Halla, http://www.prenticehall.com. Stranica sadrži aktuelne primere iz literature i primere konkretnih proizvoda iz realizovanih projekata. Takođe sadrži veze ka veb stranicama sa podacima o dobavljačima alata i metoda. To je mesto gde studenti mogu da pronađu dokumente sa projektnim zadacima, projektima, kodom, planovima testiranja i drugim. Studenti koji tragaju za dodatnim, temeljnim informacijama, upućuju se na renomirane veb stranice i publikacije. Veb stranice se redovno ažuriraju da bi štivo udžbenika ostalo aktuelno, a predstavljaju i mehanizam pomoću kog autor i izdavač dobijaju povratne informacije.
- Knjiga je puna studija konkretnih slučajeva i primera iz literature. Mnoge studije, kojima su u knjizi posvećeni numerisani podsetnici na rasterskim stranama, detaljno su prikazane u sklopu veb stranice. Student može da vidi kako su teoretski koncepti, dati u knjizi, primenjeni na primere iz prakse.

- Na kraju svakog poglavlja nalaze se pitanja koja navode na razmišljanje o pravnim i etičkim pitanjima u softverskom inženjerstvu. Studenti vide softversko inženjerstvo u svom socijalnom i političkom kontekstu. Kao i u drugim naučnim disciplinama, odluke u softverskom inženjerstvu moraju se sagledati zavisno od konsekvenci koje imaju na ljude.
- U svakom poglavlju razmatraju se aspekti proceduralnog i objektno orijentisanog pristupa razvoju. Pored toga, u poglavlju 6 koje je posvećeno objektno orijentisanom razvoju objašnjeni su koraci procesa objektno orijentisanog razvoja. Uz oslonac na UML notaciju, svaki korak je objašnjen na zajedničkom primeru, počevši od specifikaciranja zahteva do dizajna programa.
- Knjiga sadrži bibliografiju sa komentarima koja ukazuje na mnoge značajne članke iz oblasti softverskog inženjerstva. Pored toga, veb stranica sadrži pokazivače na bibliografije sa komentarima i diskusione grupe za pojedinačne oblasti, kao što su pouzdanost softvera, otpornost na greške, bezbednost računara i mnoge druge.
- Svako poglavlje sadrži opis seminarskog rada, vezan za razvoj softvera za sistem obračuna kamata. Predavač, u sklopu izvođenja nastave, može koristiti dati seminarski rad ili neku njegovu varijaciju.
- Na kraju svakog poglavlja nalazi se spisak ključnih referenci vezanih za iznete koncepte, što studentima omogućava da pronađu detaljne informacije o pojedinim alatima i metodima koje se u određenom poglavlju razmatraju.

SADRŽAJ I ORGANIZACIJA

Knjiga je organizovana u tri dela. Prvi deo motiviše čitaoca, objašnjavajući zašto je poznavanje softverskog inženjerstva jednako važno praktičarima i istraživačima. Deo I takođe razmatra potrebu razumevanja aspekata procesa, da bi se donela odluka o stepenu agilnosti koji će imati učesnici u razvoju, kao i da bi se projekat pažljivo planirao. Deo II prolazi kroz glavne korake razvoja i održavanja, nezavisno od modela procesa izrade softvera: iskazivanje, modelovanje i proveravanje zahteva, dizajniranje rešenja problema, pisanje i testiranje koda, kao i uvođenje u eksploataciju kod kupca. Deo III se usredsređuje na procenjivanje i unapređenje. Razmatra mogućnosti ocene kvaliteta procesa i proizvoda i korake za njihovo poboljšanje.

Poglavlje 1: Zašto softversko inženjerstvo?

U ovom poglavlju bavimo se našim dostignućima, motivišući čitaoca, ističući gde su ključna pitanja analizirana u kasnijim poglavljima. Posebno, razmatramo Wassermanove ključne faktore koji pomažu da se definiše softversko inženjerstvo: apstrakcije, metode i notacije za analizu i dizajn, modularnost i arhitekturu, softverski proces i životni ciklus softvera, višekratnu upotrebljivost, merenja, alate i integrisana razvojna okruženja, korisnički interfejs i izradu prototipa. Razmatra se razlika između računarske nauke i softverskog inženjerstva, uz objašnjavanje osnovnih tipova problema sa kojima se možemo susresti, čime se postavljaju osnove za ostatak knjige. Pored toga, istražujemo potrebu za sistemskim pristupom izradi softvera, i uvodimo dva zajednička primera koji će se koristiti u svakom narednom poglavlju knjige. Ovde se upoznajemo i sa kontekstom seminarskog rada.

Poglavlje 2: Modelovanje procesa i životnog ciklusa

U ovom poglavlju, predstavljamo pregled različitih tipova procesnih modela i životnog ciklusa, uključujući model vodopada, V-model, spiralni model, kao i razne modele zasnovane na izradi prototipa. Razmatramo potrebu za agilnim metodama, kod kojih je učesnicima u razvoju dat visok stepen autonomije, i poredimo ih sa tradicionalnim postupcima razvoja softvera. Takođe opisujemo nekoliko tehnika i alata za modelovanje, uključujući dinamiku sistema, SADT, i druge uobičajene pristupe. Svaki od dva tipska primera se delom modeluje uz oslonac na tehnike koje se uvode u poglavlju .

Poglavlje 3: Planiranje i upravljanje projektom

Ovde razmatramo planiranje projekta i pravljenje rasporeda aktivnosti. Upoznaćemo se sa pojmovima kao što su aktivnosti i kontrolne tačke, definisanje strukture posla, grafovi aktivnosti, upravljanje rizicima, troškovima i procenom troškova. Modeli procenjivanja su primenjeni za procenu troškova i i pravljenje rasporeda u dva tipična primera. Usredsređujemo se na stvarne studije slučajeva, uključujući upravljanje razvojem softvera za avion F-16 i AXP programima za Digitalove alfa-procesore.

Poglavlje 4: Prikupljanje zahteva

U ovom izdanju, poglavlje 4 je značajno unapređeno, posebno naglašavajući kritičnu ulogu apstrahovanja i modelovanja u softverskom inženjerstvu. Modele koristimo da bismo otklonili pogreške u razumevanju i ukazali na detalje koji nedostaju u prikupljenim zahtevima, kao i za razmenu zahteva. Istražujemo određeni broj obrazaca za modelovanje, proučavamo primere njihovih notacija, uslove korišćenja i dajemo savet o tome kako doneti odluke vezane za apstrahovanje i modelovanje. Razmatramo različite izvore i tipove žahteva (funkcionalne zahteve prema zahtevima u pogledu kvaliteta ili ograničenjima vezanim za dizajn), objašnjavamo kako se formulišu zahtevi koje je moguće testirati, i opisujemo načine za razrešavanje konflikta. Druge razmatrane teme obuhvataju iskazivanje zahteva, dokumentovanje zahteva, preispitivanje zahteva, kvalitet zahteva i mogućnost merenja kvaliteta i primer izbora metode za specifikaciju. Poglavlje se završava prikazom primene nekih navedenih metoda na dva tipska primera.

Poglavlje 5: Dizajn sistema

Ovo poglavlje se usredsređuje na pitanja arhitekture, koje otvaramo razmatranjem okvira arhitekture softvera koju su formulisali Shaw i Garlan. Dalje, opisujemo razliku između konceptualnog i tehničkog dizajna. Razmatramo uloge osoblja koje izvodi dizajn i opisujemo dva osnovna pristupa projektovanju: kompoziciju i dekompoziciju. Dalje, identifikujemo karakteristike dobrog dizajna, uvodimo nekoliko strategija, i dajemo primere više tehnika i alata za dizajn sistema. U ovom poglavlju čitalac uči o klijent-server arhitekturi, komponentama koje su višekratno upotrebljive, projektovanju interfejsa čovek-računar, projektovanju bezbednih i pouzdanih sistema (uključujući rukovanje greškama i otpornost na otkaze), obrascima za dizajn, formalnim metodama projektovanja, i ocenjivanju prednosti i mana pri izboru metoda projektovanja. Nakon objašnjavanja kako da ocenimo i proverimo kvalitet projektovanja i dokumentujemo rezultate, okrećemo se pitanjima dizajna programskog rešenja.

Objašnjene su smernice za dizajn programa, uključujući poređenje pristupa odozgo-nadole odozdo-nagore, modularnost i nezavisnost, kao i razlika između logičkog i fizičkog dizajna. Posmatramo dizajn konkurentnih sistema i sistema sa kritičnom misijom, i razmatramo tok dizajna koji je doveo do lošeg funkcionisanja sistema za medicinsko zračenje Therac-25. Opisujemo nekoliko alata za projektovanje, a tu je i temeljna rasprava o kvalitetu projektovanja i o tome kako da se on izmeri. Ovo poglavlje uvodi pojam višekratne upotrebljivosti projektnih rešenja, procene i inspekcije, i objašnjava potrebu dokumentovanja projektnih rešenja. Na kraju, poglavlje se završava primerima projektovanja informacionog sistema i sistema za rad u realnom vremenu.

Poglavlje 6: Razmatranje objekata

Poglavlje 6 za trenutak skreće sa glavnog pravca radi razmatranja posebnih osobina objektno orijentisanog razvoja. Počinjemo sa opisivanjem scenarija slučajeva korišćenja, raspravljamo kako da izdvojimo objekte i njihove karakteristike na osnovu zahteva iskazanih govornim jezikom. Koristeći alate za modelovanje koje smo uveli u poglavlju 4, predstavljamo zahteve u formi različitih modela koji, uzeti zajedno, opisuju razmatrani problem iz različitih uglova. Dalje, ispitujemo dizajn sistema, u cilju korišćenja modela za generisanje informacija na visokom nivou apstrakcije neophodnih za rešavanje problema. Nakon toga dizajn sistema obogaćujemo dodavanjem nefunkcionalnih zahteva i detalja neophodnih za dizajn programa. Oslanjajući se na UML i njegove konstrukcije, generišemo objektno orijentisanu specifikaciju i dizajn za tipski primer, u ovom slučaju servisne stanice (*Royal Service Station*).

Pažljivim uvidom u objektno orijentisana merenja, primenjujemo neke od uobičajenih objektno orijentisanih metrika, na primer servisne stanice. Utvrđujemo kako je moguće koristiti promene u metrici kao pomoć pri odlučivanju o dodeli resursa i traženju grešaka. Konačno, na primerima informacionog sistema i sistema za rad u realnom vremenu primenjujemo objektno orijentisane koncepte o kojima smo raspravljali.

Poglavlje 7: Pisanje programa

U ovom poglavlju, bavimo se odlukama u projektovanju na nivou samog koda, kao i pitanjima implementiranja dizajna da bi se proizveo programski kôd visokog nivoa kvaliteta. Raspravljamo o standardima i procedurama, i predlažemo neka jednostavna uputstva za programiranje. Prikazani su primeri realizovani na različitim objektno orijentisanim i proceduralnim programskim jezicima. Razmatramo potrebu za programskom dokumentacijom i strategijom za rukovanje greškama. Poglavlje se završava primenom nekih od navedenih koncepata na dva tipska primera.

Poglavlje 8: Testiranje programa

U ovom poglavlju istražujemo nekoliko aspekata testiranja programa. Razdvajamo konvencionalni pristup testiranju programa od metoda *Cleanroom*, i razmatramo testiranje različitih sistema. Uvodimo definicije i kategorije softverskih problema, i raspravljamo kako ortogonalna klasifikacija defekata može da učini efikasnijim proces prikupljanja i analize podataka. Nakon toga objašnjavamo razliku između testiranja jedinica i testiranja u fazi integracije. Nakon uvođenja nekoliko automatskih alata i tehnika za testiranje,

РЕПУБЛИКА СРПСКА
Природно-математички
факултет Бања Лука
Ф И З И К А

objašnjavamo potrebu za životnim ciklusom testiranja i kako se u njega mogu integrisati alati. Na kraju je data primena ovih koncepata na dva tipska primera.

Poglavlje 9: Testiranje sistema

Počinjemo sa principima testiranja sistema, uključujući višekratnu upotrebljivost skupova testova i test podataka, kao i potrebom za pažljivim rukovanjem konfiguracijom. Koncepti koji se uvode uključuju testiranje funkcija, testiranje performansi, testove prihvatanja i instalacione testove. Razmatramo posebne aspekte testiranja objektno orijentisanih sistema. Prikazano je više alata za testiranje, a razmotrene su i uloge članova tima za testiranje. Dalje, upoznajemo čitaoca sa modelovanjem pouzdanosti softvera, i razmatramo pitanja pouzdanosti, mogućnosti održavanja i raspoloživosti. Čitalac uči kako da koristi rezultate testiranja za procenu očekivanih karakteristika isporučenog proizvoda. Upoznajemo se sa više tipova dokumentacije za testiranje, a poglavlje se završava opisom strategije testiranja na dva tipska primera.

Poglavlje 10: Isporuka sistema

U ovom poglavlju razmatra se potreba za obukom i dokumentacijom, gde se uvodi nekoliko primera obuke i dokumenata koji mogu da idu uz primer iz informacionog sistema i sistema za rad u realnom vremenu.

Poglavlje 11: Održavanje sistema

U ovom poglavlju bavimo se posledicama izmene sistema. Biće objašnjeno kako izmene mogu da se dogode tokom životnog ciklusa sistema i kako dizajn sistema, kodiranje, proces testiranja i dokumentovanja moraju da prihvate takve izmene. Razmatraju se tipični problemi održavanja, kao i potreba za pažljivim upravljanjem konfiguracijom. Tu je i temeljna rasprava o upotrebi merenja u cilju predviđanja verovatnih izmena, kao i evaluacija efekata izmena. Reinženjering i restrukturisanje se razmatraju u sveobuhvatnom kontekstu podmlađivanja nasleđenih sistema. Na kraju poglavlja je izvršena evaluacija dva tipska primera, zavisno od verovatnoće izmena.

Poglavlje 12: Evaluacija proizvoda, procesa i resursa

Kako mnoge odluke u softverskom inženjerstvu obuhvataju uključivanje i integraciju postojećih nezavisno razvijenih komponenti, ovo poglavlje se bavi načinima za evaluaciju procesa i proizvoda. Razmatra se potreba za empirijskom evaluacijom i daje više primera da bi se prikazalo kako se merenje može koristiti za uspostavljanje osnovne linije kvaliteta i produktivnosti. Videćemo nekoliko modela kvaliteta, način evaluacije sistema na osnovu stepena ponovne iskoristivosti, izvođenje naknadne analize (postmorteme) i shvatiti ulogu povraćaja investicija u informacione tehnologije. Ovi koncepti su primenjeni na dva tipska primera.

Poglavlje 13: Unapređenje predviđanja, proizvoda, procesa i resursa

Ovo poglavlje se nastavlja na poglavlje 11, prikazujući kako može da se postigne poboljšanje predviđanja, proizvoda, procesa i resursa. Ono sadrži više temeljnih studija slučaje-

va radi prikaza kako se modeli predviđanja, tehnike inspekcije i drugi aspekti softverskog inženjerstva mogu razumeti i poboljšati oslanjanjem na različite istraživačke tehnike. Ovo poglavlje završava sa skupom smernica za evaluaciju aktuelne situacije i identifikacije mogućnosti za poboljšanje.

Poglavlje 14: Budućnost softverskog inženjerstva

U ovom finalnom poglavlju razmatramo više otvorenih pitanja u softverskom inženjerstvu. Ponovo razmatramo Wassermanov koncept da bismo videli koliko ga dobro sprovodimo kao disciplinu. Istražujemo neka pitanja transfera tehnologije i donošenja odluka, da bismo utvrdili da li radimo dobro u postupku primene značajnih istraživačkih ideja u praksi. Novo u ovom izdanju jeste razmatranje kontroverznih pitanja, kao što su licenciranje softverskih inženjera i trendovi razvoja metoda i rešenja specifičnih za domen.

ZAHVALNOSTI

Knjige se pišu ako prijatelji i porodica pružaju tehničku i emocionalnu podršku. Nemoguće je navesti ovde sve one koji su pomagali da izdržimo tokom pisanja i ispravljanja, a unapred se izvinjavamo za eventualne omaške. Veliku zahvalnost dugujemo čitaocima ranijih izdanja, čije je pažljivo i pomno analiziranje štiva generisalo odlične sugestije za ispravke i razjašnjenja. Koliko nam je poznato, sve te sugestije su uključene u ovo izdanje. I dalje ćemo ceniti povratne informacije od naših čitalaca, bilo da su pozitivne ili negativne.

Carolyn Seaman (University of Maryland, Baltimore Campus) je bila odličan recenzent prvog izdanja, gde je predložila načine za razjašnjavanje i pojednostavljenja, što je vodilo ka sažetijem i razumljivijem štivu. Ona je takođe pripremila mnoga rešenja vežbanja i pomogla kod postavljanja rane verzije veb lokacije ove knjige. Zahvalna sam zbog njenog prijateljstva i pomoći. Yiqing Liang i Carla Valle su ažurirale veb lokaciju i dodale suštinski novu građu koja se nalazi u drugom izdanju; Maria Augusta Vieira Nelson (Catholic University of Minas Gerais, Brazil) je uradila reviziju veb lokacije za treće izdanje, s posebnim osvrtom na notacije kod modelovanja i agilne metode.

Veliku zahvalnost dugujemo Forrest Shullu (Fraunhofer Center, Maryland) i Roseanni Tesoriero (Washington College) koji su razvili inicijalni priručnik uz ovu knjigu, a tu je i Maria Nelson koja je izvršila reviziju priručnika i uputstva sa rešenjima uz treće izdanje. Takođe posebnu zahvalnost dugujemo Guilhermu Travassosu (Federal University of Rio de Janeiro) za upotrebu materijala koji je razvijen sa Pfleegerom (University of Maryland, College Park), koji je onda on značajno obogatio i proširio za upotrebu u narednim predavanjima.

Ozbiljni i korisni recenzenti sva tri izdanja: Barbara Kitchenham (Keele University, Velika Britanija), Bernard Woolfolk (Lucent Technologies), Ana Regina Caval-canti da Rocha (Federal University of Rio de Janeiro), Frances Uku (University of California at Berkeley), Lee Scott Ehrhart (MITRE), Laurie Werth (University of Texas), Vickie Almstrum (University of Texas), Lionel Briand (Carleton University, Ottawa), Steve Thibaut (University of Florida), Lee Wittenberg (Kean College of New Jersey), Philip Johnson (University of Hawaii), Daniel Berry (University of Waterloo, Kanada), Nancy Day

(University of Waterloo), Jiawei Niu (University of Waterloo), Chris Gorringe (University of East Anglia, Velika Britanija), Ivan Aaen (Aalborg University) i više drugih neimenovanih recenzenata koje je angažovao Prentice Hall. Rasprave koje su sa autorom vodili Greg Hislop (Drexel University), John Favaro (Intecs Sistemi, Italija), Filippo Lanubile (Università di Bari, Italija), John d'Ambra (University of New South Wales, Australija), Chuck Howell (MITRE) i James i Suzanne Robertson (Atlantic Systems Guild, Velika Britanija), doveli su do mnogih poboljšanja i unapređenja.

Još jednom, hvala Toniju Holmu i Alanu Aptu, koji su učinili rad na trećem izdanju knjige zanimljivim i relativno bezbolnim. James i Suzzane Robertson iskoristili su primer Picadilly, i na tome smo im zahvalni, kao i Normanu Fentonu za upotrebu materijala za softversku metriku koji potiče iz zajedničke knjige.

Mnogo hvala izdavačima za više slika i primera jer su dali dozvolu da ih reprodukujemo na ovom mestu. Grada iz knjige *Complete Systems Analysis* i *Mastering the Requirements Process* (Robertson i Robertson 1994) preuzeta je sa dozvolom od izdavača Dorset House Publishing, na lokaciji www.dorsethouse.com; sva prava su zadržana. Članak u vežbi 1.1 reprodukovan je iz *Washington Posta* sa dozvolom Associated Pressa. Slike 2.15 i 2.16 reprodukovane su iz Barghouti i dr. (1995) uz dozvolu koju je dao izdavač John Wiley and Sons Limited. Slike 12.14 i 12.15 reprodukovane su iz Routa (1995) uz dozvolu izdavača John Wiley and Sons Limited.

Slike i tabele u poglavljima 2, 3, 4, 5, 9, 11 i 12 koje su označene zaštitnim znakom IEEE, štampane su sa dozvolom organizacije Institute of Electrical and Electronics Engineers. Slično, tri tabele u poglavlju 14, koje su označene zaštitnim znakom ACM, štampane su sa dozvolom organizacije Association of Computing Machinery. Tabela 2.1 i slika 2.11, čiji je autor Lai (1991), reprodukovane su sa dozvolom Software Productivity Consortiuma. Slike 8.16 i 8.17 čiji je autor Graham (1996a) štampane su sa dozvolom Dorothy R. Grahama. Slika 12.11 i tabela 12.2 adaptirane su iz reference Liebman (1994) sa dozvolom Centre for Science in the Public Interest, 1875 Connecticut Avenue NW, Washington DC. Tabele 8.2, 8.3, 8.5 i 8.6 reprodukovane su sa dozvolom kompanije McGraw-Hill. Slike i primeri iz referenci Shaw i Garland (1996), Card i Glass (1990), Grady (1997), kao i Lee i Tepfenhart (1997), reprodukovane su sa dozvolom Prentice Halla.

Tabele 9.3, 9.4, 9.6, 9.7, 13.1, 13.2, 13.3 i 13.4, kao i slike Figures 1.15, 9.7, 9.8, 9.9, 9.14, 13.1, 13.2, 13.3, 13.4, 13.5,13.6 i 13.7 reprodukovane su ili adaptirane iz reference Fenton and Pfleeger (1997), u celini ili delimično, sa dozvolom Norman Fentona. Slike 3.16, 5.19, and 5.20 reprodukovane su ili adaptirane iz beležaka sa predavanja Norman Fentona, što nam je on rado dozvolio.

Posebno smo zahvalni našem poslodavcu, RAND Corporationu. Podjednaku zahvalnost upućujemo i kolegama sa University of Waterloo, za njihovo ohrabrivanje tokom pripremanja izdanja[1]. Takođe zahvaljujemo našim prijateljima i porodici koji su nam pružili ljubaznost, podršku i strpljenje jer je pisanje knjige oduzelo vreme koje smo upravo sa njima provodili. Posebno, Shari Lawrence Pfleeger je zahvalna Manny Lawrencu, upravniku Royal Service Stationa, kao i njegovom knjigovođi, Bei Lawrence, radeći ne samo sa njom i njenim studentima na specifikaciji Royal sistema, već takođe na njenom

1 Napominjemo da ova knjiga nije proizvod RAND korporacije i nije pretrpela RAND-ov proces provere kvaliteta. Ovaj rad predstavlja nas kao autore, a ne nas kao zaposlene u pomenutoj instituciji.

uticaju i podučavanju u njihovom drugom zadatku: ulozi njenih roditelja. Jo Atlee posebno zahvaljuje svojim roditeljima, Nancy i Gary Atlee, koji su ja podržavali i ohrabrivali u svemu što je radila (i u svakom njenom pokušaju); a takođe i svojim kolegama i studentima, koji su blagonaklono učestvovali u radu tokom glavnog perioda pisanja. Takođe, najviše zahvaljujemo Charlesu Pfleegeru i Ken Salemu koji su bili stalni i veoma dobrodošli izvori podrške, ohrabrenja i dobrog raspoloženja.

Shari Lawrence Pfleeger
Joanne M.Atlee

1 Zašto softversko inženjerstvo?

U ovom poglavlju obradićemo:

- šta se podrazumeva pod softverskim inženjerstvom;
- dostignuća softverskog inženjerstva;
- šta podrazumevamo pod pojmom „dobar softver";
- zašto je važan sistemski pristup;
- kako se softversko inženjerstvo promenilo od 1970. godine.

Softver prožima sve pore sveta koji nas okružuje, ali mi ponekad nismo ni svesni koliko on čini naš život udobnijim i uspešnijim. Na primer, posmatrajmo jednostavne radnje koje koristimo za pripremanje tosta za doručak. Programski kôd koji se nalazi u tosteru kontroliše koliko će hleb da se prepeče i kada će iskočiti iz aparata. Programi kontrolišu i regulišu isporuku električne energije u našem domu, a softver obračunava njen utrošak. U stvari, možemo da koristimo automatizovane programe za plaćanje računa za struju, naručivanje namirnica, čak i za kupovinu novog tostera! U današnje vreme rad softvera, kako onaj koji je svima vidljiv, tako i onaj koji je sakriven, prisutan je u gotovo svim aspektima našeg života, uključujući i sisteme od presudnog značaja za naše zdravlje i sveopšte stanje. Iz tog razloga, softversko inženjerstvo je danas važnije nego ikada. Dobra softver-inženjerska praksa mora da obezbedi da softver daje pozitivan doprinos načinu na koji živimo.

Ova knjiga stavlja u prvi plan ključna pitanja koja se postavljaju u softverskom inženjerstvu, opisujući naše znanje o tehnikama i alatima i kako oni utiču na konačne proizvode koje gradimo i koristimo. Napravićemo pregled i teorije i prakse: onoga šta znamo i kako to primenjujemo u tipičnom projektu gde se softver razvija ili održava. Takođe ćemo istražiti i stvari koje do sada nismo znali, ali koje mogu da nam pomognu da naše proizvode učinimo pouzdanijim, bezbednijim, korisnijim i pristupačnijim.

Počećemo pregledom kako analiziramo probleme i kako realizujemo rešenja. Onda ćemo istražiti razlike između problema koji se javljaju u računarskoj nauci i softverskom inženjerstvu. Naš krajnji cilj je da proizvedemo rešenja koja uključuju softver visokog kvaliteta, i razmotrimo karakteristike koje doprinose kvalitetu.

Takođe ćemo videti koliko smo bili uspešni u razvoju softverskih sistema. Istražujući nekoliko primera softverskih otkaza, videćemo gde se nalazimo na putu ovladavanja majstorstvom razvoja kvalitetnog softvera.

РЕПУБЛИКА СРПСКА
Природно-математички
факултет Бања Лука
Ф И З И К А

Nakon toga, pažnju ćemo posvetiti učesnicima u razvoju softvera. Nakon što opišemo uloge i odgovornosti kupca, korisnika i učesnika u razvoju, okrenućemo se proučavanju samog sistema. Videćemo da sistem može da se predstavi kao omeđena grupa objekata povezanih skupom aktivnosti. Alternativno, sistem možemo posmatrati očima inženjera; razvijanje sistema ima dosta sličnosti sa zidanjem kuće. Kada definišemo korake potrebne za izgradnju sistema, razmotrićemo uloge razvojnog tima u svakom od tih koraka.

Na kraju, razmotrićemo neke promene koje su uticale na način praktičnog sprovođenja softverskog inženjerstva. Predstavićemo osam Wassermanovih ideja da bismo praksu i ideje povezali u jedinstvenu koherentnu celinu.

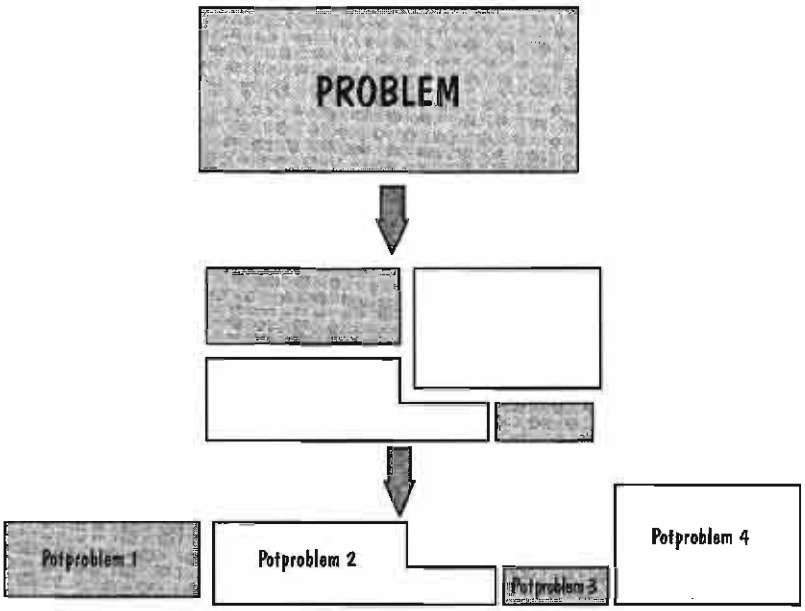
1.1 ŠTA JE SOFTVERSKO INŽENJERSTVO

Kao softverski inženjeri, koristimo naše poznavanje računara i računarstva kao pomoć prilikom rešavanja problema. Često je problem kojim se bavimo povezan sa računarom ili postojećim računarskim sistemom, ali ponekad poteškoće na kojima on počiva nemaju dodirnih tačaka sa računarima. Stoga je osnovno da prvo razumemo prirodu problema. Zato posebno moramo paziti da ne namećemo računare ili računarske tehnike kao rešenje svakog problema sa kojim se susretnemo. Prvo je potrebno da rešimo problem. Zatim, ako je potrebno, možemo da koristimo tehnologiju kao sredstvo za implementaciju rešenja. U daljim razmatranjima u sklopu ove knjige, pretpostavićemo da je naša analiza pokazala da je neka vrsta računarskog sistema neophodna ili poželjna za rešavanje problema koji razmatramo.

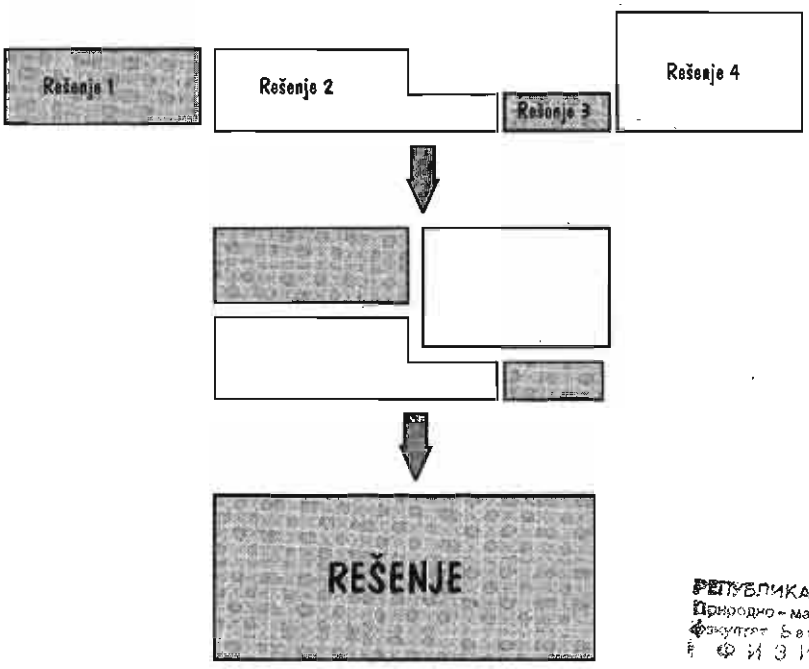
Rešavanje problema

Mnogi problemi su obimni i nekada su nezgodni, posebno ako predstavljaju nešto novo što nikada ranije nije rešavano. Stoga istraživanje problema moramo započeti **analizom**, tj. razlaganjem problema na delove koje možemo da razumemo i sa kojima ćemo pokušati da se izborimo. Na osnovu toga možemo jedan veliki problem opisati kao skup manjih međusobno povezanih problema. Slika 1.1 ilustruje kako se vrši analiza. Važno je da zapamtimo da su veze (na slici su to strelice, uz relativne položaje potproblema) jednako važne kao i sami potproblemi. Ponekad, upravo su relacije, a ne priroda potproblema, ključni faktori u iznalaženju rešenja složenog problema.

Po obavljenoj analizi problema, sledi sastavljanje kompletnog rešenja iz delova koji se odnose na različite aspekte problema. Slika 1.2 ilustruje ovaj obrnuti proces: **Sinteza** je sklapanje veće strukture na osnovu manjih gradivnih elemenata. Kao i u slučaju analize, sklapanje pojedinačnih rešenja može biti jednako izazovno kao i postupak traženja rešenja. Da bismo videli zašto je to tako, poslužićemo se analogijom sa pisanjem romana. Rečnik sadrži sve reči koje možda želite da upotrebite tokom pisanja. Ali najteži deo pisanja jeste odlučivanje kako da organizujemo i ukomponujemo reči u rečenice, odnosno rečenice u pasuse i poglavlja, da bismo na kraju uobličili celokupnu knjigu. Stoga, svaka tehnika rešavanja problema mora da sadrži dva dela: analizu problema radi određivanja njegove prirode i, na analizi zasnovanu, sintezu rešenja.



SLIKA 1.1 Postupak analize



РЕПУБЛИКА СРПСКА
Природно-математички
факултет Бања Лука
Ф И З И К А

SLIKA 1.2 Postupak sinteze

Da bismo pomogli sebi u rešavanju problema, primenićemo raznolike metode, alate, procedure i paradigme. **Metod** ili **tehnika** je formalna procedura za pravljenje nekog ishoda. Na primer, glavni kuvar može da pripremi preliv pomoću niza sastojaka kombinovanih u pažljivom redosledu i odabranim trenucima tako da se preliv zgusne, ali da se ne zgruša ili da ostane nesjedinjen. Postupak pripreme preliva uključuje merenje vremena i sastojke, ali može da zavisi i od vrste opreme koja se koristi za kuvanje.

Alat je instrument ili automatizovani sistem koji pomaže da se nešto obavi na bolji način. Taj „bolji način" može značiti da zahvaljujući alatu radimo preciznije, efikasnije ili delotvornije, ili dobijamo bolji kvalitet rezultujućeg proizvoda. Na primer, koristimo pisaću mašinu ili tastaturu i štampač za pisanje pisama jer je rezultujući dokument čitljiviji nego rukopis. Ili koristimo makaze kao alat jer možemo da sečemo brže i pravije nego kad rukom cepamo list papira. Međutim, alat nije uvek neophodan da bismo nešto napravili dobro. Na primer, bolji preliv može da se napravi zahvaljujući boljoj tehnici kuvanja a, ne zato što kuvar koristi bolju šerpu ili kutlaču.

Procedura je kao recept: kombinacija alata i tehnika, koji, u međusobnom skladu, proizvode dati proizvod. Na primer, kao što ćemo videti u kasnijim poglavljima, planovi za testiranje opisuju procedure za testiranje; oni nam govore koji alati treba koristiti nad kojim skupovima podataka i pod kojim okolnostima, da bismo odredili da li naš softver zadovoljava postavljene zahteve.

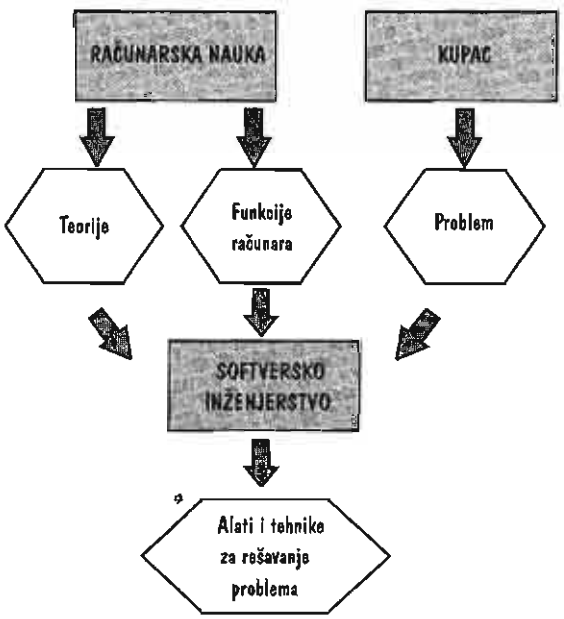
Na kraju, **paradigma** je kao stil kuvanja; ona predstavlja poseban pristup ili filozofiju gradnje softvera. Kao što jednostavno razlikujemo francusku kuhinju od kineske, na isti način razlikujemo i paradigme, kao što uostalom postoji i razlika između proceduralnog i objektno orijentisanog razvoja. Jedan nije bolji od drugog; svaki ima svoje prednosti i mane, a postoje situacije u kojima je jedan od njih pogodniji.

Softverski inženjeri koriste alate, tehnike, procedure i paradigme da bi poboljšali kvalitet softverskih proizvoda. Njihov cilj je da koriste efikasne i produktivne pristupe za generisanje efektivnog rešenja problema. U poglavlju koje sledi, biće istaknuti posebni pristupi koji služe kao podrška opisanim aktivnostima razvoja i održavanja. Na pratećoj veb stranici za ovu knjigu nalazi se ažurni skup pokazivača na alate i tehnike.

Gde se tu uklapa softversko inženjerstvo?

Da bismo razumeli kako se softverski inženjer uklapa u svet računarske nauke, pogledaćemo kao primer jednu drugu disciplinu. Uzmimo, na primer, hemiju i način na koji se u hemiji rešavaju problemi. Hemičar istražuje hemikalije: njihovu strukturu, uzajamno delovanje i teorijsku osnovu njihovog ponašanja. Hemijski inženjeri primenjuju rezultate hemijskog proučavanja na razne probleme. Za hemičare je hemija kao predmet proučavanja, ali je s druge strane, za hemijskog inženjera, hemija je i alat koji se koristi za bavljenje nekim opštim problemom (koji možda čak i nije „hemijski" po svojoj prirodi).

Računarstvo možemo da posmatramo u sličnom svetlu. Možemo da se usredsredimo na računare i programske jezike, ili možemo da ih gledamo kao alate koji se koriste u projektovanju i primeni rešenja nekog problema. Softversko inženjerstvo kao što je prikazano na slici 1.3. predstavlja ovaj drugi pogled. Umesto da istražujemo dizajn hardvera ili da dokazujemo teoreme koje opisuju rad algoritama, softverski inženjer se usredsređuje na računar kao sredstvo za rešavanje problema. Videćemo kasnije u ovom poglavlju da softver-inženjer koristi računarske funkcije kao delove generalnog rešenja, umesto da se bavi organizacijom ili teorijom rada računara.



SLIKA 1.3 Odnos između računarske nauke i softverskog inženjerstva

1.2 KOLIKO SMO BILI USPEŠNI?

Pisanje softvera je i umetnost i nauka, i naročito je važno da to shvate studenti računarskih nauka. Računarski naučnici i softver-inženjeri istraživači proučavaju računarske mehanizme i teorije da bi ih učinili produktivnijim ili efikasnijim. Međutim, oni takođe projektuju računarske sisteme i pišu programe za obavljanje poslova na tim sistemima, praktični rad koji je dobrim delom umetnost, domišljatost i veština. Postoji više načina da se određeni zadatak izvede na određenom sistemu, pri čemu su neki načini bolji od drugih. Jedan način je možda efikasniji i precizniji, lakše se modifikunje, koristi ili razume. Svaki haker može da napiše programski kôd koji nešto radi, ali su potrebni umeće i znanje profesionalnog softverskog inženjera da bi se proizveo stabilan i razumljiv kôd koji se lako održava i koji efikasno i efektivno radi ono zbog čeka je napravljen. Prema tome, kod softverskog inženjeringa suština je u projektovanju i razvoju visoko kvalitetnog softvera.

Pre nego što istražimo šta je potrebno za pravljenje kvalitetnih softverskih sistema, osvrnućemo se na to koliko smo bili uspešni do sada. Da li su korisnici zadovoljni sa postojećim softverskim sistemima? I da i ne. Softver nam je omogućio da radimo poslove brže i efikasnije nego ikada do sada. Prisetimo se kako smo živeli pre pojave, na primer, programa za obradu teksta, radnih tabela, elektronske pošte ili sofisticirane telefonije. Softver podržava održavanje i spasavanje života u medicini, poljoprivredi, transportu i većini drugih industrijskih grana. Pored toga, softver je omogućio da primenjujemo stvari koje nekada nismo ni mogli da zamislimo: mikrooperativne zahvate, multimedijalno obrazovanje, robotiku i mnoge druge.

Međutim, softver nije bez vlastitih problema. Sistemi često ne funkcionišu baš na očekivani način. Svi smo čuli priče o sistemima koji jedva da rade. Svi mi smo pisali programe u kojima je bilo grešaka: kôd koji sadrži greške, ali je dovoljno dobar da položimo

ispit ili da demonstriramo izvodljivost nekog pristupa. Jasno je da takvo ponašanje nije prihvatljivo kada razvijamo sistem koji treba da se isporuči kupcu.

Postoji ogromna razlika između greške u školskom projektu i one u velikom softverskom sistemu. U stvari, u literaturi i kuloarima se često razmatraju softverske greške i poteškoće u izradi softvera koji ne sadrži greške. Neke greške su zaista dosadne, dok nas druge koštaju dosta vremena i novca. Ima i onih koje su opasne po život. Podsetnik 1.1 objašnjava odnose između nedostataka, grešaka i otkaza. Pogledajmo nekoliko primera otkaza gde ćemo videti šta je pošlo naopako i zašto.

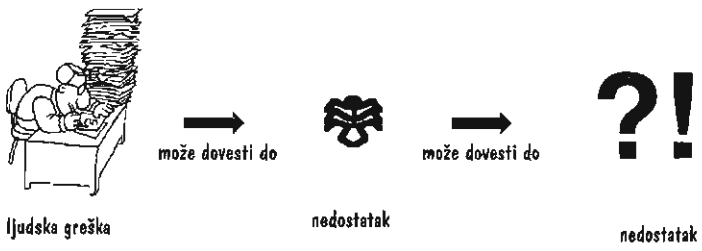
PODSETNIK 1.1 TERMINOLOGIJA ZA OPISIVANJE „PROGRAMSKIH BUBICA"

Često govorimo o „bubicama" u softveru, i pri tome mislimo na više različitih stvari, zavisno od konteksta. Pojam „bubica" može da bude pogreška u tumačenju programskog zahteva, sintaksna greška u delu koda ili (do tada nepoznat) uzrok pada sistema. Organizacija IEEE je sugerisala standardnu terminologiju (u IEEE standardu 729) za opisivanje „bubica" u softverskim proizvodima (IEEE 1983).

Nedostatak se događa kada ljudsko biće nešto zabrlja u izvođenju neke softverske aktivnosti, i to nazivamo **greška**. Na primer, projektant može loše da razume programski zahtev i osmisli program koji ne odgovara stvarnoj potrebi analitičara i korisnika. Ovaj nedostatak u projektu je podloga za grešku, pa može da dovede do drugih nedostataka, kao što su neispravan kôd ili netačan opis u korisničkom uputstvu. Stoga vidimo da jedna greška može da generiše više nedostataka, pri čemu nedostatak može da se nalazi u svakom razvijanom ili održavanom proizvodu.

Otkaz je odstupanje od neophodnog ponašanja sistema. Može biti otkriven pre ili nakon isporuke sistema, tokom testiranja, tokom korišćenja ili održavanja. Kako dokumenti sa programskim zahtevima mogu da sadrže nedostatke, otkazi ukazuju da se sistem ne ponaša kao što je neophodno, iako se možda ponaša u skladu sa specifikacijom.

Stoga nedostatak predstavlja unutrašnji pogled na sistem, kako ga vide oči učesnika u razvoju, dok otkaz predstavlja pogled na sistem spolja, tj. problem kakvog vidi korisnik. Svaki nedostatak nema uvek odgovarajući otkaz. Na primer, ako se programski kôd sa nedostatkom nikada ne izvršava ili se u to posebno stanje nikada ne ulazi, tada nedostatak nikada neće prouzrokovati otkaz. Slika 1.4 prikazuje genezu otkaza.



SLIKA 1.4 Kako ljudske greške prouzrokuju otkaze

U ranim 1980-im, američka poreska uprava angažovala je korporaciju Sperry za izgradnju sistema za automatsku obradu obrazaca za savezni porez na prihode. Prema onome što je objavio dnevnik *Washington Post*, „sistem ... je dokazano neodgovarajući za takvo radno opterećenje, koštao je skoro duplo nego što smo očekivali i mora biti uskoro zamenjen" (Sawyer 1985). U 1985. godini, dodatnih 90 miliona dolara bilo je potrebno za poboljšanje početne Sperryjeve opreme vredne 103 miliona dolara. Dodatno, jer je postojeći problem sprečio poresku upravu da refundira poreske obveznike u predviđenom roku, pa su bili prinuđeni da plati 40,2 miliona dolara kamate i 22,3 miliona nadoknade za prekovremeni rad svojih radnika koji su pokušavali da sustignu rokove. U 1996. godini situacija se nije poboljšala. Dnevne novine *Los Angeles Times* izvestile su 29. marta da još uvek ne postoji glavni plan za modernizaciju računara poreske uprave, već samo tehnički dokument od 6 000 stranica. Kongresmen Jim Lightfoot nazvao je taj projekat „fijasko od 4 milijarde dolara, koji je zapeo zbog neodgovarajućeg planiranja" (Vartabedian 1996). U poglavlju 2 videćemo zašto je planiranje projekta bitno za proizvodnju kvalitetnog softvera.

Više godina je javnost bez pogovora prihvatala upliv softvera u njenu svakodnevicu. Međutim, strateška odbrambena inicijativa (*Strategic Defense Initiative*, SDI), koju je predložio predsednik Regan, podigla je svest javnosti o teškoćama proizvodnje softverskog sistema koji ne sadrži greške. Izveštaji popularnih dnevnih novina i časopisa (kao što su Jacky 1985, Parnas 1985 i Rensburger 1985) izražavali su skepticizam u računarskoj zajednici. A sada, 20 godina kasnije, kako je od američkog kongresa traženo da dodeli sredstva za gradnju sličnog sistema, mnogi računarski naučnici i softverski inženjeri nastavili su da veruju da ne postoji način pisanja i testiranja softvera koji garantuje adekvatnu pouzdanost.

Na primer, mnogi softverski inženjeri misle da bi jedan antibalistički raketni sistem zahtevao makar 10 miliona redova koda; neki procenjuju red veličine čak sto miliona redova. Radi poređenja, softver koji služi za podršku američkoj svemirskoj letelici šatl sastoji se od 3 miliona redova koda, uključujući i zemaljske računare koji kontrolišu ispaljivanje rakete i let. Godine 1985. bilo je u samom šatlu 100 000 redova koda (Rensburger 1985). Stoga bi protivraketni softverski sistem zahtevao testiranje enormne količine koda. Povrh svega, ograničenja u pogledu pouzdanosti bilo bi nemoguće testirati. Da bismo videli zašto je to tako, razmotrićemo pojam bezbednosno kritičnog softvera. Obično kažemo da nešto što je **važno za bezbednost** (tj. nešto čiji otkaz predstavlja pretnju po život ili zdravlje), treba da ima pouzdanost od najmanje 10^{-9}. Kao što ćemo videti u poglavlju 9, to znači da sistem sme da otkaže samo jedanput u 10^9 časova rada. Ali, 10^9 časova je preko 114 000 godina, suviše dugo kao interval za testiranje.

Videćemo takođe u poglavlju 9 da inače korisna tehnologija može postati smrtonosna ako je softver projektovan ili programiran na neodgovarajući način. Na primer, medicinski krugovi bili su zaprepašteni kada je Therac-25, mašina za radio terapiju i rendgen, loše funkcionisala i ubila više pacijenata. Projektanti softvera jednostavno nisu predvideli istovremenu upotrebu više tastera sa strelicama na nestandardni način. Posledica toga je bila da je softver zadržavao visoke vrednosti parametara i emitovao visoko koncentrovane doze radijacije u situacijama kada su zahtevane male doze (Leveson i Turner 1993).

Sličan primer nepredviđene primene i njenih opasnih posledica opisan je u časopisu *Pilot*, o izveštaju podnesenom na Risks Forumu (*Pilot* 1996). Dva policajca u Škotskoj

koristili su pištolj-radar za registrovanje brzine vozila. Radar se neočekivano zaglavio, prikazujući brzinu veću od 300 milja na čas. Nekoliko sekundi kasnije, proleteo je niskoleteći mlaznjak Harrier. Tragač meta na lovcu Harrier prepoznao je radar i shvatio da pripada „neprijatelju". Srećom, Harrier nije bio naoružan, jer bi uobičajeno ponašanje bilo da ispali raketu sa automatskim navođenjem!

Neočekivana upotreba sistema mora da se uzme u obzir u toku razvoja softvera. To se može obaviti na dva načina: zamišljanjem kako sve sistem može biti zloupotrebljavan (i pravilno upotrebljavan), kao i pretpostavljanjem da će sistem biti zloupotrebljavan, i projektovanje softvera za rukovanje takvim zloupotrebama. Ti pristupi biće razmatrani u poglavlju 8.

I pored toga što mnogi proizvođači teže softveru bez mana, u stvarnosti mnogi softverski proizvodi sadrže nedostatake. Tržište diktira brzi razvoj softvera jer je cilj da se proizvodi brzo isporuče, što ostavlja malo vremena za temeljno testiranje. Tim za testiranje po pravilu može da testira jedino one funkcije koje će se najverovatnije koristiti, ili one koje bi najverovatnije mogle da ugroze, odnosno iritiraju korisnike. Zato su mnogi korisnici opravdano zabrinuti kada instaliraju prvu verziju koda, znajući da programske greške neće biti ispravljene sve do druge verzije. Nadalje, nekada je teže realizovati modifikacije potrebne za ispravljanje uočenih nedostataka, nego ponovo napisati kompletan softver od početka. U poglavlju 11 istražićemo pitanja vezana za održavanje softvera.

Uprkos nekim spektakularnim uspesima i sveopštem prihvatanju softvera kao sastavnog dela života, još uvek ima dosta prostora za njegovo poboljšanje. Na primer, odsustvo kvaliteta je skupo, jer što duže nedostatak ostaje neotkriven, njegovo otklanjanje više košta. Procene pokazuju da trošak ispravljanja greške u fazi analize projekta, predstavlja svega jednu desetinu cene ispravljanja iste greške nakon isporuke sistema korisniku. Nažalost, većina grešaka se ne otkriva u ranoj fazi. Na greške napravljene rano u životnom ciklusu sistema odlazi polovina troškova ispravljanja grešaka otkrivenih tokom testiranja i održavanja. U poglavljima 12 i 13 razmotrićemo načine za ocenjivanje efikasnosti razvojne aktivnosti i načine za poboljšanje procesa u cilju što ranijeg otkrivanja grešaka.

Jedna od jednostavnih, ali moćnih tehnika koju ćemo predložiti jesu pregledi i inspekcije. Mnogi studenti su navikli na samostalni razvoj i testiranje softvera. Međutim, njihovo samostalno testiranje ume da bude neefikasnije nego što to oni misle. Na primer, Fagan je proučavao način na koji se otkrivaju nedostaci. On je otkrio da testiranje programa njegovim izvršavanjem na uzorku podataka predviđenom za testiranje otkriva samo petinu od svih ukupno otkrivenih nedostataka tokom razvoja sistema. Preostale četiri petine otkrivaju se neposrednim pregledanjem koda, odnosno kada kolege ispituju i komentarišu jedni drugima način projektovanja i programski kôd (Fagan 1986). To znači da kvalitet softvera može značajno da se poveća ako kolege jednostavno pregledaju naš rad. U kasnijim poglavljima pokazaćemo kako postupak pregledanja i inspekcije može da se koristi nakon svakog glavnog razvojnog koraka u cilju otklanjanja nedostataka što je ranije moguće. Videćemo u poglavlju 13 kako je moguće unaprediti i sam postupak inspekcije.

1.3 ŠTA JE DOBAR SOFTVER?

Kao što proizvođači traže načine da osiguraju kvalitet proizvoda koji prave, isto tako softverski inženjeri moraju da pronalaze metode za obezbeđivanje prihvatljivog kvaliteta i korisnosti njihovih proizvoda. Stoga dobro softversko inženjerstvo mora uvek da uključuje i strategiju za proizvodnju kvalitetnog softvera. Ali pre nego što smislimo strategiju, moramo da shvatimo šta se podrazumeva pod kvalitetnim softverom. Podsetnik 1.2 prikazuje kako gledište utiče na ono što zovemo „kvalitet". U ovom odeljku ispitaćemo po čemu se dobar softver razlikuje od lošeg.

PODSETNIK 1.2 GLEDIŠTA NA KVALITET

Garvin (1984) je razmatrao kako različiti ljudi doživljavaju kvalitet. On opisuje kvalitet sa pet različitih gledišta:

- *transcendentalni pogled*, gde je kvalitet nešto što možemo da prepoznamo ali ne i da definišemo;
- *korisnički pogled*, gde je kvalitet usklađenost sa namenom;
- *pogled sa aspekta proizvodnje*, gde je kvalitet usklađenost sa specifikacijom;
- *pogled sa aspekta proizvoda*, gde je kvalitet vezan za karakteristike samog proizvoda;
- *pogled na bazi vrednosti*, gde kvalitet zavisi od toga koliko je kupac spreman da za njega plati.

Transcendentalni pogled je vrlo sličan Platonovom opisu ideala ili Aristotelovom konceptu oblika. Drugim rečima, kao što je svaki stvarni sto aproksimacija idealnog stola, tako je i kvalitet softvera u stvari ideal kome stremimo, ali koji možda nikada nećemo u potpunosti ostvariti.

Transcendentalni pogled je prilično metafizički, nasuprot mnogo konkretnijem pogledu koji ima korisnik. Korisnički pogled prihvatamo kada merimo karakteristike proizvoda, kao što su npr. gustina nedostataka ili pouzdanost, u cilju razumevanja sveukupnog kvaliteta proizvoda.

Pogled sa aspekta proizvodnje posmatra kvalitet u toku procesa izrade i nakon isporuke. U stvari, ovaj pogled proverava da li je proizvod inicijalno sagrađen ispravno, čime se izbegavaju skupe prepravke radi otklanjanja isporučenih nedostataka. Stoga fabrički pogled predstavlja u stvari pogled sa stanovišta postupka, tj. zagovara saobraznost dobrom postupku. Međutim, malo je dokaza koji govore o tome da strogo sprovođenje postupka daje proizvode sa manje nedostataka i otkaza. Ispravan postupak zaista može da ka visoko kvalitetnom proizvodu, ali može i da institucionalizuje proizvodnju osrednjeg kvaliteta. Neka od tih pitanja biće istražena u poglavlju 12.

Korisnički i proizvodački pogled posmatraju proizvod spolja, ali pogled sa aspekta proizvoda posmatra stvari iznutra i ocenjuje karakteristike samog proizvoda. Ovaj pogled najčešće zastupaju stručnjaci za softversku metriku. Oni podrazumevaju da će dobri interni pokazatelji kvaliteta dovesti i do dobrih eksternih pokazatelja kvaliteta, kao što su npr. pouzdanost i lako održavanje. Međutim, potrebno je još istraživanja za proveru tih pretpostavki i za određivanje koja gledišta kvaliteta utiču na stvarnu upotrebu proizvoda. Možda ćemo morati da razvijemo modele koji povezuju pogled proizvoda sa korisničkim pogledom.

Kupci ili prodavci često usvajaju korisnički pogled na kvalitet. Istraživači ponekad posmatraju stvari sa aspekta proizvoda, dok razvojni tim posmatra sa aspekta procesa proizvodnje. Ako razlike u gledištima nisu izričito naglašene, onda konfuzija i nerazumevanje mogu da dovedu do loših odluka i loših proizvoda. Pogled na bazi vrednosti može da poveže te sasvim različite pogledo na kvalitet. Stavljanjem znaka jednakosti između kvaliteta i onoga što je kupac voljan da plati, možemo sagledati kompromis koji treba napraviti između cene i kvaliteta, i na osnovu njega razrešavati konfliktne situacije, kada se pojave. Slično tome, kupci porede cenu proizvoda sa potencijalnom korišću, razmišljajući o kvalitetu kao vrednosti koje dobijamo u zamenu za novac.

Kitchenham i Pfleeger (1996) su istraživali odgovor na to pitanje u uvodniku za specijalno izdanje časopisa *IEEE Software*, posvećeno kvalitetu. Oni naglašavaju da kontekst pomaže da se utvrdi odgovor. Nedostaci koji mogu da se tolerišu u softveru za obradu teksta, ne mogu da se prihvate u sistemima gde je bezbednost faktor od izuzetnog značaja, ili onima sa kritičnom misijom. Stoga, kvalitet moramo posmatrati na najmanje tri načina: kvalitet proizvoda, kvalitet postupka izrade proizvoda, i kvalitet proizvoda u kontekstu poslovnog okruženja u kojem će se on koristiti.

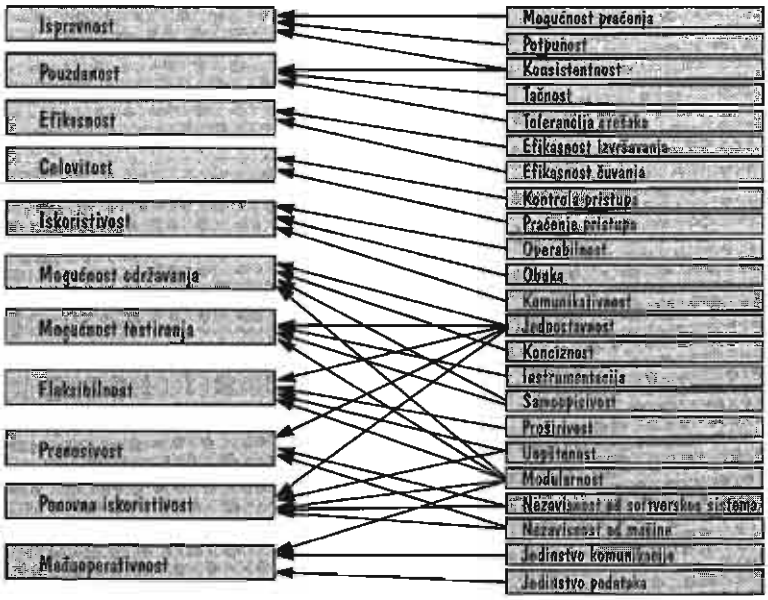
Kvalitet proizvoda

Ako pitamo različite ljude da navedu karakteristike koje učestvuju u ukupnom kvalitetu softvera, verovatno ćemo dobiti različite odgovore od svakog. Razlike se javljaju zato što značaj neke karakteristike zavisi od toga ko analizira softver. Korisnici smatraju da je softver visokog kvaliteta ako radi na način na koji oni žele, ako se lako uči i koristi. Međutim, nekada su kvalitet i finkcionalnost isprepleteni. Ako je nešto teško naučiti ili koristiti, ali ima funkcionalnost koja je vredna truda, onda visok kvalitet nije sporan.

Kvalitet softvera pokušavamo da merimo poređenjem jednog proizvoda sa drugim. Da bismo to postigli, identifikujemo one aspekte sistema koji doprinose njegovom ukupnom kvalitetu. Otuda, kada mere kvalitet softvera, korisnici ocenjuju spoljašnje karakteristike kao što su broj otkaza i tip otkaza. Na primer, oni mogu da klasifikuju otkaze kao minorne, glavne i katastrofične, i nadaju se da se događaju samo minorni otkazi.

Softver takođe moraju da procenjuju oni koji projektuju i pišu kôd i oni čiji je zadatak kasnije održavanje programa. Ovi praktičari teže razmatranju internih karakteristika proizvoda, nekada čak i pre nego što je proizvod isporučen korisniku. Detaljnije, praktičari često posmatraju broj i tip nedostataka kao dokaz kvaliteta proizvoda (ili kao dokaz nedostatka kvaliteta). Na primer, programeri prate broj nedostataka koji se nalaze u programskim zahtevima, projektu i inspekcijama koda i koriste ih kao indikatore željenog kvaliteta finalnog proizvoda.

Zatočesto pravimo modele koji dovode u vezu spoljašnji pogled korisnika i unutrašnji pogled programera na softver. Slika 1.5 je primer jednog od prvih modela kvaliteta, koji je napravio McCall sa svojim kolegama da bi pokazao u kakvom su odnosu spoljašnji faktori kvaliteta (na levoj strani) sa kriterijumima kvaliteta proizvoda (na desnoj stra-



SLIKA 1.5 McCallov model kvaliteta

ni). McCall je svakom kriterijumu sa desne strane pridružio merenje radi prikazivanja stepena uticaja na posmatrani element kvaliteta (McCall, Richards i Walters 1977). U poglavlju 12 ispitaćemo nekoliko modela kvaliteta proizvoda.

Kvalitet procesa

Postoji mnogo aktivnosti koje imaju uticaja na konačni kvalitet proizvoda. Ako neka od tih aktivnosti pođe naopako, to može da pogorša kvalitet proizvoda. Iz tog razloga, mnogi softverski inženjeri osećaju da je kvalitet postupka razvoja i održavanja važan jednako kao i kvalitet proizvoda. Jedna od prednosti modelovanja postupka jeste da možemo da ga analiziramo i nađemo načine da ga poboljšamo. Na primer, možemo da postavimo pitanja kao što su:

- Gde i kada ćemo verovatno da nađemo određenu vrstu nedostatka?
- Kako možemo što ranije da pronađemo nedostatke u postupku razvoja?
- Kako možemo da ugradimo toleranciju na greške, da bismo smanjili verovatnoću da nedostatak pređe u otkaz?
- Da li postoje alternativne aktivnosti koje mogu da načine proces efektivnijim ili efikasnijim, uz osiguranje kvaliteta?

Ova pitanja mogu da se primene na celokupan proces razvoja ili pojedinačne pot-postupke, kao što su upravljanje konfiguracijom, ponovno korišćenje ili testiranje. Te postupke istražićemo u kasnijim poglavljima.

U 1990-im, značajan publicitet dobili su modelovanje i poboljšanje procesa u softverskom inženjerstvu. Standardi procesa inspirisani su radom Deminga i Jurana, a primenjivale su ih razne kompanije, između ostalih i IBM. Neki od standarda, kao što su CMM (*Capability Maturity Model*), ISO 9000 i SPICE *(Software Process Improvement and Capability dEtermination*) sugerišu da unapređenjem postupka razvoja softvera, možemo da poboljšamo kvalitet rezultujućeg proizvoda. U poglavlju 2 videćemo kako možemo da identifikujemo relevantne aktivnosti postupka i kako da modelujemo njegove efekte na poluproizvode i konačni proizvod. Poglavlja 12 i 13 detaljno ispituju modelovanje postupka i okvire poboljšanja.

Kvalitet u kontekstu poslovnog okruženja

Kada se u žiži ocenjivanja kvaliteta nalaze proizvodi i postupci, kvalitet obično merimo matematičkim izrazima koji uključuju nedostatke, otkaze i vreme. Ređe se opseg proširuje uključivanjem aspekata poslovanja, gde se kvalitet posmatra zavisno od proizvoda i usluga koje pruža poslovni sistem čiji je softver sastavni deo. Drugim rečima, posmatramo tehničku vrednost naših proizvoda i jedino na osnovu njih donosimo odluke, umesto da ih posmatramo sa mnogo šireg aspekta kakav je njihova poslovna vrednost. Pretpostavljamo da se unapređenje tehničkog kvaliteta automatski odslikava na poslovnu vrednost.

Više istraživača detaljno je proučavalo odnose između poslovne i tehničke vrednosti. Na primer, Simmonsova je ispitala mnoge australijske preduzetnike da bi odredila kako donose poslovne odluke u vezi sa informacionim tehnologijama. Ona predlaže okvir za razumevanje šta kompanije podrazumevaju pod „poslovnom vrednošću" (Simmons 1996). U izveštaju Favara i Pfleegera (1997), Steve Andriole, izvršni direktor za informatiku velike američke osiguravajuće kompanije Cigna Corporation, opisao je kako njegova kompanija razlikuje tehničku vrednost od poslovne:

> Mi merimo kvalitet (našeg softvera) pomoću očiglednih meta: odnos između vremena operativnosti i vremena zastoja, troškova održavanja, troškova izmena i tome slično. Drugim rečima, razvojem upravljamo na osnovu radnih performansi u okviru parametara troškova. Na koji način isporučilac obezbeđuje isplativu performansu manje je važno u odnosu na rezultate rada ... Pitanje poslova prema tehničkoj vrednosti je blisko i drago našem srcu ... i to je ono na šta usredsređujemo veliki deo pažnje. Mislim da bih se iznenadio kada bih saznao da jedna kompanija sa drugom ugovara tehničku vrednost na račun gubitka poslovne vrednosti. Makar i grešili na drugoj strani! Ako ne postoji jasna (očekivana) poslovna vrednost (kvantitativno izražena: broj obrađenih reklamacija itd.) onda ne pokrećemo projekat. Veoma ozbiljno shvatamo fazu specifikacije zahteva u sklopu projekta kao „utvrđivanje svrhe", kada postavljamo pitanje: „zašto zaista želimo ovaj sistem?" i „zašto nas se to tiče?".

Bilo je više pokušaja za nalaženje veza između tehničke i poslovne vrednosti na kvantitativan i smislen način. Na primer, Humphrey, Snyder i Willis (1991) napominju da je unapređenjem procesa razvoja u skladu sa CMM skalom „zrelosti" (biće razmatrana u poglavlju 12), kompanija Hughes Aircraft poboljšala produktivnost četiri puta i uštedela milione dolara. Slično tome, Dion (1993) izveštava da je Raytheonovo udvostručenje produktivnosti bilo praćeno sa 7,7 dolara zarade na svaki dolar investiran u unapređenje procesa. Osoblje u vazduhoplovnoj bazi Tinker u Oklahomi zabeležilo je poboljšanje produktivnosti od 6,35 puta (Lipke i Butler 1992).

Međutim, Brodman i Johnson (1995) detaljnije su proučili poslovnu vrednost unapređenja postupka. Oni su ispitali 33 kompanije koje su aktivnošću neke vrste unapredili postupak, pri čemu su istražili više ključnih pitanja. Između ostalog, Brodman i Johnson pitali su kompanije kako definišu povraćaj od investicije, koncept koji je jasno definisan u poslovnim krugovima. Oni napominju da školska definicija **povraćaja od investicije**, izvedena iz finansijskog okruženja, opisuje investiciju zavisno od toga kakvu je vrednost ona donela. To jest: „investicija ne sme da vrati samo početni kapital, već najmanje onoliko više koliko bi ta sredstva zaradila na drugom mestu, plus dodatak za rizik" (Putnam i Myers 1992). Obično, poslovno okruženje koristi jedan od tri modela za ocenu povraćaja od investicije: model naknade, računovodstveni model stope povraćaja i diskontni model toka gotovine.

Međutim, Brodman i Johnson (1995) su utvrdili da vlada i privreda SAD tumače povraćaj od investicije na vrlo različite načine, a ta oba tumačenja različita su od standardnog pristupa u poslovnim školama. Vlada tretira povraćaj od investicija u zavosnosti od dolara, posmatrajući i redukujući operativne troškove, predviđajući dolarsku štednju i računajući trošak primene novih tehnologija. Vladine investicije takođe su izražene u dolarima, kao npr. cena uvođenja novih tehnologija ili inicijativa za unapređenje postupka.

Na drugoj strani, privreda posmatra investicije zavisno od uloženog rada, umesto cene ili pak dolara. To jest, kompanijama je u interesu da uštede vreme ili da koriste manje ljudi, a njihova definicija povraćaja od investicije odražavala je takav njihov interes kao što manji uloženi rad. Među kompanijama koje su ispitivane, povraćaj od investicije obuhvata je elemente kao što su:

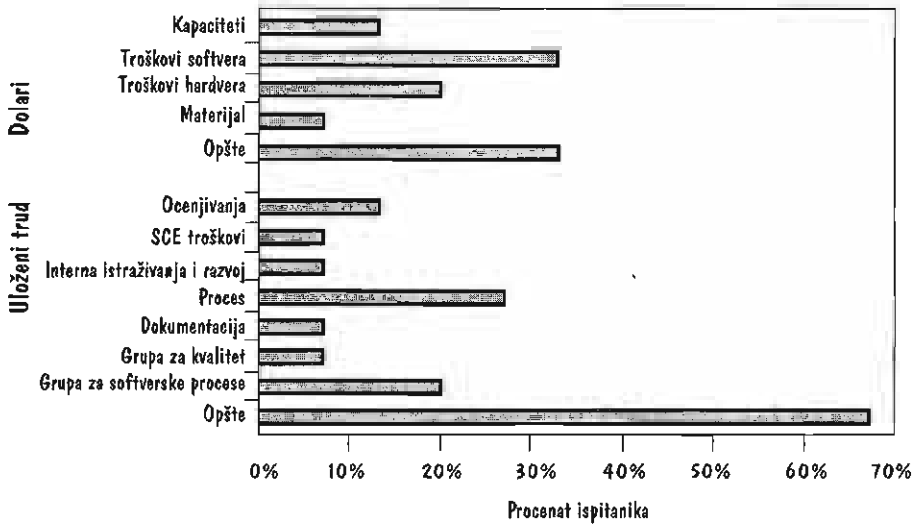
- obuka,
- raspored rada,
- rizik,
- kvalitet,
- produktivnost,
- postupak,
- kupac,
- troškovi,
- preduzetništvo.

Pitanje troškova sadržano u definiciji odnosi se na predviđanje troškova, poboljšanje performansi troškova i ostajanje u okviru budžeta, a ne na smanjivanje operativnih troškova ili reorganizaciju projekta ili organizacionog sistema. Slika 1.6 prikazuje učestanost sa kojom razne organizacije uključuju stavku investicija u svoju definiciju pojma povraćaja od investicije. Na primer, oko 5 procenata ispitanika uključilo je rad grupa za kvalitet u izračunavanje povraćaja od investicija, a oko 35 procenata uključilo je softverske troškove kada su razmatrali iznos investiranih dolara.

Razlika u pogledima unosi nespokojstvo, jer to znači da se povraćaji od investicija u raznim organizacijama ne mogu međusobno porediti. Ali postoje dobri razlozi za te različite poglede. Dolarske uštede nastale redukovanjem delatnosti, viši nivo kvaliteta i uvećana produktivnost pre se vraćaju vladi nego izvođaču radova. S druge strane, izvođači radova obično teže konkurentnosti i većim radnim kapacitetima, kao i ostvarenju većeg profita. Stoga je povraćaj od investicija za izvođača radova zasnovaniji na ulože-

РЕПУБЛИЧКА СРПСКА
Математички

nom radu nego na troškovima. Posebno, tačnija procena troškova i rasporeda rada može da rezultuje zadovoljstvom kupca i novim poslovima. Smanjenje vremena za iznošenje proizvoda na tržište i unapređenje kvaliteta proizvoda takođe pružaju poslovnu vrednost. Primećuje se da poslovnu vrednost nude takođe i kraći put do kupca, kao i unapređen kvalitet proizvoda.



SLIKA 1.6 Termini uključeni u definiciju povraćaja od investicija u privredi

Čak i ako se različite kalkulacije povraćaja od investicija mogu opravdati kod različitih organizacija, zabrinjavajuće je da povraćaj od investicija u softverske tehnologije nije isti kao finansijski povraćaj od investicija. U nekom trenutku, uspeh programa se mora saopštiti višim upravljačkim nivoima, od kojih mnogi nemaju dodirnih tačaka sa softverom, već sa osnovnom delatnošću kompanije, kao što su telekomunikacije ili bankarstvo. Mnogo konfuzije potiče od upotrebe iste terminologije za označavanje veoma različitih stvari. Stoga, kriterijum uspešnosti mora da ima smisla ne samo u slučaju softverskih projekata i postupaka, već i u slučaju mnogo opštijih delatnosti, čiju podršku oni predstavljaju. Detaljnije ćemo istražiti ovo pitanje u poglavlju 12, gde ćemo videti nekoliko uobičajenih mera za izbor tehnoloških opcija zasnovanih na poslovnoj vrednosti.

1.4 KO SE BAVI SOFTVERSKIM INŽENJERSTVOM?

Ključna komponenta softverskog razvoja jeste komunikacija između kupca i projektanta. Ako ta komunikacija doživi neuspeh, isto će zadesiti i sistem. Da bismo mogli da napravimo sistem koji treba da pomogne kupcu u rešavanju njegovog problema, moramo da razumemo šta kupac želi i šta mu je potrebno. Da bismo to uradili, usredsredićemo našu pažnju na pojedince uključene u razvoj softvera.

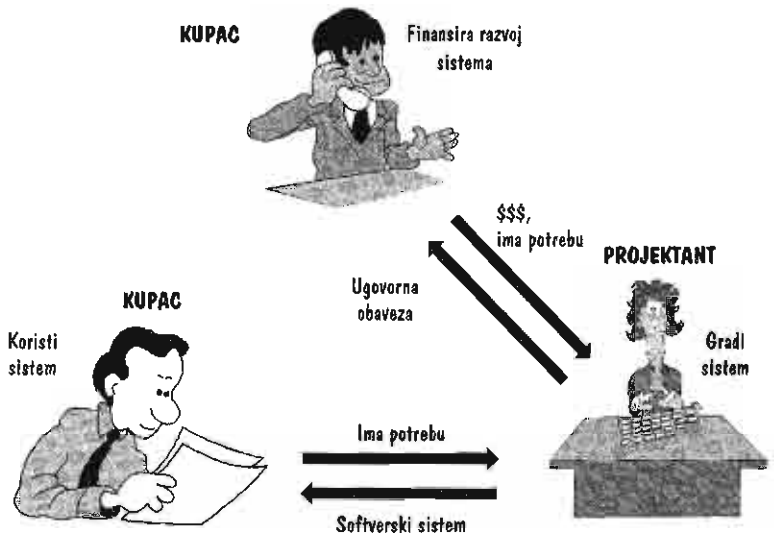
Broj osoba koje rade na razvoju softvera zavisi od veličine projekta i stepena složenosti. Međutim, nije bitno koliko je ljudi uključeno, uloge koje oni igraju tokom trajanja projekta jasno se razlikuju. Stoga, kod velikog projekta, svim pojedincima ili grupama

može biti dodeljena jedna od identifikovanih uloga. Kod malog projekta, jedna osoba ili grupa mogu istovremeno da imaju više uloga.

Obično, učesnici u projektu imaju jednu od tri uloge: kupca, korisnika i projektanta. **Kupac** je kompanija, organizacija ili pojedinac koji plaća za softverski sistem koji se razvija. **Razvoj** obavlja kompanija, organizacija ili pojecinac koji pravi softverski sistem za kupca. Ova kategorija uključuje i rukovodioce potrebne za koordinisanje i vođenje programera i testera. **Korisnik** je jedan ili više pojedinaca koji će stvarno koristiti sistem. To su oni koji sede za terminalom, unose podatke ili čitaju izlazne rezultate. Iako kod nekih projekata ista osoba ili grupa predstavljaju i kupca i korisnika i projektanta, u većini slučajeva se radi o različitim učesnicima. Slika 1.7 prikazuje osnovne odnose između tri tipa učesnika.

Kupac, koji kontroliše novčana sredstva, obično pregovara o ugovoru i potpisuje papire o tehničkom prijemu. Međutim, ponekad kupac nije u isto vreme i korisnik. Na primer, pretpostavimo da kompanija za vodosnabdevanje Wittemberg Water Works potpiše ugovor sa kompanijom Gentle Systems za pravljenje kompjuterizovanog obračunskog sistema. Predsednik Wittenberga može da opiše predstavnicima Gentle Systemsa tačno šta želi i potpisaće ugovor. Međutim, predsednik neće neposredno koristiti obračunski sistem. Korisnici će biti knjigovođe i šalterski radnici. Stoga je važno da razvojni tim razume tačno šta žele kupac i korisnik, odnosno šta im je potrebno.

S druge strane, pretpostavimo da je Wittenberg Water Works toliko velik da ima sopstveni sektor za razvoj računarskih sistema. Taj sektor može da odluči da želi automatski alat pomoću kojeg će pratiti troškove sopstvenih projekata i projektni raspored. Ako taj alat sami prave, taj sektor je istovremeno korisnik, kupac i projektant.



SLIKA 1.7 Učesnici u razvoju softvera

Poslednjih godina, razlika između kupca, korisnika i projektanta postala je mnogo složenija. Kupci i korisnici su uključeni u razvojni proces na više načina. Kupac može da odluči da kupi gotov, kataloški proizvod, koji će biti ugrađen u finalni proizvod koji će

razvojni tim da isporuči i za koji će davati podršku. Kada se to dogodi, kupac je uključen u odluke o arhitekturi sistema, gde postoji mnogo više ograničenja u pogledu razvoja. Slično tome, za razvoj se mogu angažovati razvojni inženjeri ili timovi spolja, koje nazivamo podizvođači, koji grade podsisteme i isporučuju ih razvojnom timu, radi uključenja u finalni proizvod. Podizvođači mogu da rade rame uz rame sa primarnim razvijaocima, isporučujući podsistem kasnije u razvojnom procesu. Podsistem može biti po sistemu **„ključ u ruke"**, gde se programski kôd ugrađuje kao celina (bez dodatnog koda koji treba integrisati), ili se mora posebno integrisati da bi se uspostavile veze između glavnog sistema i podsistema.

Stoga je pojam „sistem" važan u softverskom inženjerstvu, ne samo da bi se razumeli analitički prolemi i sintetisanje rešenja, već i za organizovanje procesa razvoja i dodeljivanje odgovarajućih uloga učesnicima . U sledećem odeljku, videćemo ulogu sistemskog prilaza u pozitivnoj praksi softverskog inženjerstva.

1.5 SISTEMSKI PRILAZ

Projekti koje razvijamo ne egzistiraju u vakuumu. Hardver i softver koje sastavljamo, zajedno su u interakciji sa korisnicima, sa drugim softverima, sa drugim delovima hardvera, postojećim bazama podataka (tj. sa pažljivo definisanim skupovima podataka i odnosima između podataka), ili čak sa drugim računarskim sistemima. Stoga je važno da obezbedimo kontekst projekta koji nazivamo **granice (opseg)** projekta, tj. šta je uključeno u projekat, a šta ne. Na primer, uzmimo da vaš pretpostavljeni od vas zahteva da napišete program za štampanje platnih listića za radnike vašeg preduzeća. Morate da znate pri tome da li vaš program treba da čita radne sate iz drugog sistema i da jednostavno štampa rezultate, ili možda mora takođe da izračunava informacije o plati. Dalje, potrebno je znati da li program treba da izračunava poreze, doprinose za penzije i razne dodatke, odnosno da li i to treba da bude prikazano na svakom platnom listiću. U stvari, postavlja se pitanje gde započinje projekat i gde se završava? Ovo pitanje važi za svaki sistem. Sistem predstavlja skup objekata i aktivnosti, i veza koje međusobno povezuju objekte i aktivnosti. Obično, definicija sistema obuhvata, za svaku aktivnost, spisak obaveznih ulaznih informacija, radnji koje se preduzimaju, kao i izlaznih informacija koje će se proizvesti. Stoga, da bismo uopšte mogli početi, moramo da znamo da li je neki objekat ili aktivnost uključen u sistem ili ne.

Elementi sistema

Opisaćemo sistem imenujući njegove delove i identifikujući kako se sastavni delovi međusobno odnose. Ta identifikacija je prvi korak u analizi problema koji nam je predstavljen.

Aktivnosti i objekti. Najpre, između aktivnosti i objekata postoji razlika. Aktivnost je nešto što se događa u sistemu. Obično se opisuje kao događaj iniciran nekim okidačem; aktivnost transformiše jednu stvar u drugu menjajući joj osobine. Ta transformacija može da znači da se podatak prenosi sa jedne lokacija na drugu, ili se jedna vrednost menja u drugu, ili se kombinuje sa drugim podacima da bi se napravila ulazna informacija za neku drugu aktivnost. Na primer, pojedinačni podatak može da se prenese iz jedne datoteke u drugu. U tom slučaju, karakteristika koja je izmenjena jeste njegova lokacija.

Vrednost podataka može da se uveća. Na kraju, adresa podataka može da se uključi u listu parametara sa adresama više drugih podataka u cilju poziva neke druge rutine koja obrađuje sve navedene podatke odjednom.

Elementi uključeni u aktivnosti zovu se **objekti** ili **entiteti**. Obično, ti objekti su povezani na neki način. Na primer, objekti mogu da budu uređeni u tabelu ili matricu. Često su grupisani u zapise, gde je svaki zapis uređen u propisanom formatu. Na primer, zapis o istoriji rada nekog radnika može da sadrži objekte (koje zovemo polja) za svakog radnika, kao na primer:

Ime	Poštanski broj
Srednje slovo	Plata po času
Prezime	Dodaci po času
Ulica i broj	Obračunati slobodni sati
Grad	Obračunato bolovanje
Država	

Ne samo da je svako polje u zapisu definisano, već je navedena veličina svakog polja i njihov međusobni odnos. Stoga, opis zapisa u svakom polju mora da se sastoji od tipa podatka, početne pozicije u zapisu i dužine polja. Dalje, kako postoji zapis za svakog radnika, zapisi se kombinuju u datoteku, koja takođe ima svoje devinisane karakteristike (na primer, maksimalni broj zapisa).

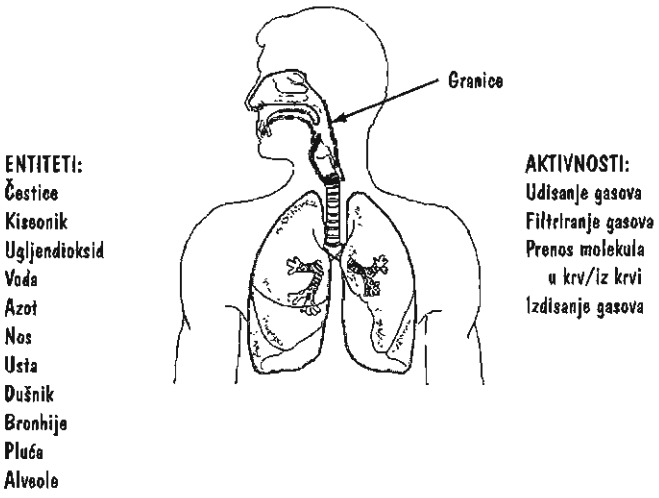
Ponekad, objekti su definisani na neznatno različit način. Umesto da posmatramo svaku stavku kao polje u većem zapisu, objekti se posmatraju kao nezavisni. Opis objekta sadrži spisak karakteristika svakog objekta, kao i spisak svih akcija koje mogu da se odigraju upotrebom objekta ili koji utiču na objekat. Na primer, razmotrićemo objekat „poligon". Opis objekta može da kazuje da taj objekat ima karakteristike kao što je broj stranica i njihova dužina. Akcije mogu da uključuju izračunavanje površine ili obima. Može da postoje karakteristika „tip poligona", koja omogućuje indentifikovanje svakog primerka „poligona", kao na primer „romb" ili „četvorougaonik". Opis objekta može biti sadržan u samom tipu. Na primer, „četvorougaonik" može da bude sastavljen od tipova „kvadrat" i „nije kvadrat". Objasnićemo ove koncepte u poglavlju 4, kada budemo istraživali analizu zahteva, a detaljno u poglavlju 6 kada budemo razmatrali objektno orijentisani razvoj.

Odnosi i granice sistema. Jednom kad definišemo entitete i aktivnosti, onda uparujemo entitete sa odgovarajućim aktivnostima. Odnosi između entiteta i aktivnosti jasno su i pažljivo definisani. Definicija entiteta sadrži opis odakle potiče entitet. Neke stavke se nalaze u postojećim datotekama, dok se druge stvaraju tokom same aktivnosti. Odredište entiteta je takođe važno. Neke stavke koristi samo jedna aktivnost, dok su druge namenjene kao ulaz za druge sisteme. To jest, neke stavke iz jednog sistema se koriste od strane aktivnosti izvan opsega sistema koji ispitujemo. To znači da je posmatrani sistem ograničen i ima ograničeni domet. Neke stavke prelaze granice da bi ušle u naš sistem, a druge su proizvodi našeg sistema i putuju napolje da bi ih koristili drugi sistemi.

Koristeći te koncepte, možemo da definišemo **sistem** kao zbirku stvari: skup entiteta, skup aktivnosti, opis odnosa između entiteta i aktivnosti, i definiciju granica sistema. Ova definicija sistema važi ne samo za računarske sisteme već i za sve drugo gde su jedni objekti na neki način u interakciji sa drugim objektima.

Primeri sistema. Da bismo videli definiciju sistema na delu, posmatraćemo respiratorni sistem, tj. delove koji omogućavaju udisanje kiseonika i izbacivanje ugljen-dioksida i vode. Lako možemo da definišemo granice sistema. Ako imenujemo pojedinačni organ tela, možemo da kažemo da li on jeste ili nije deo respiratornog sistema. Molekuli kiseonika i ugljen-dioksida su entiteti ili objekti koji se kreću kroz sistem na način koji je jasno definisan. Možemo takođe da opišemo aktivnosti u sistemu zavisno od interakcija entiteta. Ako je neophodno, možemo da ilustrujemo sistem prikazujući šta ulazi u njega i šta iz njega izlazi. Možemo takođe da obezbedimo tabele koje opisuju sve entitete i aktivnosti u koje su one uključene. Slika 1.8 ilustruje respiratorni sistem. Napomenimo da svaka aktivnost uključuje entitete i može biti definisana entitetima koji predstavljaju ulaz, načinom obrade entiteta i finalnim proizvodom (izlazom).

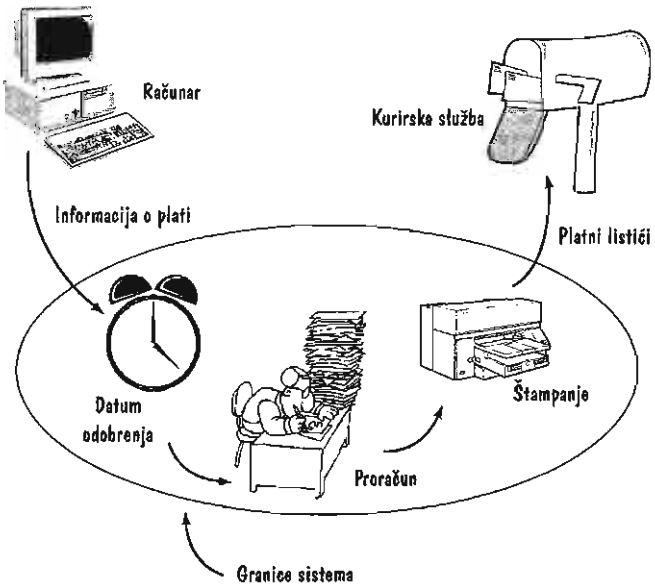
SLIKA 1.8 Respiratorni sistem



Takođe, moramo da jasno opišemo računarski sistem. Sa potencijalnim korisnikom radimo na definisanju granica sistema: Gde naš rad započinje i gde prestaje? Pored toga, potrebno je da znamo čime je ograničen sistem i tako odredimo poreklo ulaza i odredište izlaza. Na primer, u sistemu koji štampa platne listiće, informacija o plati može da dolazi iz računara kompanije. Izlaz sistema može biti skup platnih listića poslat u kurirsku službu da bi se otpremio odgovarajućim primaocima. U sistemu prikazanom na slici 1.9, možemo da vidimo granice i shvatimo entitete, aktivnosti i njihove odnose.

Međusobno povezani sistemi

Koncept razgraničenja je važan jer postoji vrlo malo sistema koji su nezavisni od drugih sistema. Na primer, respiratorni sistem je u interakciji sa probavnim sistemom, sistemom krvotoka, nervnim sistemom i drugima. Respiratorni sistem ne bi mogao da funkcioniše bez nervnog sistema, čak ni krvotok ne bi mogao da funkcioniše bez respiratornog sistema. Međusobne zavisnosti su složene. Činjenica je da su mnogi problemi u okruženju nastali i intenzivirali se jer se nije cenila kompleksnost eko sistema. Ipak, kada su opisane granice sistema lakše je videti šta je unutra a šta van, kao i šta prelazi granicu.



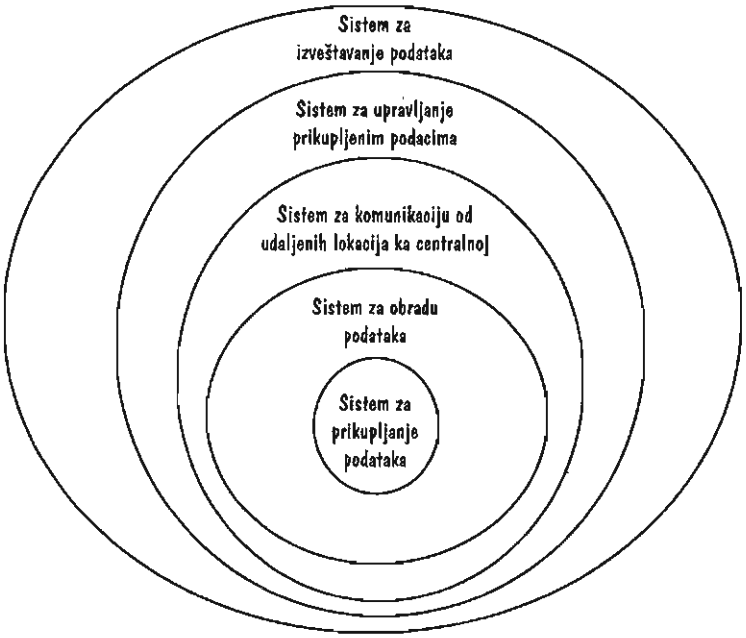
SLIKA 1.9 Definicija sistema za pravljenje platnih listića

Dalje, moguće je da jedan sistem postoji unutar drugog sistema. Kada opisujemo računarski sistem, često se bavimo samo malim delom onoga što je zaista nadređeni sistem. Takvo usredsređivanje omogućava nam da definišemo i pravimo manje kompleksan sistem nego što je onaj koji ga okružuje. Ako pažljivo dokumentujemo interakcije između sistema koji utiče na naš sistem, ne gubimo ništa koncentrišući se na taj manji deo većeg sistema.

Pogledaćemo primer kako to možemo da uradimo. Pretpostavimo da razvijamo sistem za nadgledanje vodotoka gde se podaci sakupljaju na mnogim tačkama duž rečne doline. U sabirnim centrima se radi nekoliko proračuna, a rezultati se šalju na centralnu lokaciju radi sveobuhvatnog izveštavanja. Takav sistem može biti organizovan tako što računar na centralnoj lokaciji komunicira sa više desetina manjih računara na udaljenim lokacijama. Pri tome je potrebno da razmotrimo više sistemskih aktivnosti, uključujući način kako se prikupljaju podaci o vodi, proračuni koji se izvode na udaljenim lokacijama, prenos informacija do centralne lokacije, skladištenje primljenih podataka u bazi podataka ili zajedničkoj datoteci, kao i pravljenje izveštaja od dobijenih podataka. Sistem možemo da posmatramo kao skup sistema, svaki sa posebnom namenom. Posebno, možemo da posmatramo samo komunikacione aspekte velikog sistema i razvijemo komunikacioni sistem za prenos podataka od skupa udaljenih lokacija do centralne lokacije. Ako pažljivo definišemo granicu između komunikacija i velikog sistema, projektovanje i razvoj komunikacionog sistema može se realizovati nezavisno od celokupnog sistema.

Kompleksnost čitavog sistema za nadgledanje vodotoka mnogo je veća nego kompleksnost komunikacionog sistema, tako da tretiranje odvojenih, manjih delova, čini posao mnogo jednostavnijim. Ako je definicija razgraničenja detaljna i ispravna, pravljenje velikog sistema od manjih sistema je relativno lako. Možemo da opišemo proces gradnje posmatrajući veliki sistem kroz slojeve, kao što je prikazano na slici 1.10, koja prikazuje primer nadgledanja vodotoka. Svaki sloj predstavlja sistem, ali svaki sloj sa

svojim podslojevima takođe formira sistem. Kružnice na slici predstavljaju razgraničenja odgovarajućih sistema, a čitav skup kružnica se udružuje u celovit sistem za nadgledanje vodotoka.



SLIKA 1.10 Slojevi sistema za nadgledanje vodotoka

Uočavanje da jedan sistem sadrži drugi sistem je važno, jer odražava činjenicu da je objekat ili aktivnost u jednom sistemu deo svakog sistema predstavljenog spoljašnjim slojevima. Kako svaki sloj unosi dodatnu složenost, razumevanje pojedinačnih objekata ili aktivnosti postaje sve teže. Stoga fokusiranjem u startu na najmanji mogući sistem vršimo maksimalno pojednostavljivanje, čime se poboljšava naše dalje razumevanje sistema.

Tu ideju koristimo kada pravimo sistem kojim menjamo stariju verziju, bilo ručno ili automatski. Želimo što bolje da razumemo kako rade stari i novi sistemi. Često, što je veća razlika između dva sistema, to su projektovanje i razvoj teži. Ova poteškoća nije prouzrokovana samo time što ljudi imaju otpor prema promenama, nego i time što razlike otežavaju učenje. Tokom gradnje ili sinteze našeg velikog sistema, od velike pomoći je pravljenje novog sistema koji predstavlja inkrementalni niz međusistema. Umesto da idemo od sistema A ka sistemu B, biće moguće da idemo od A preko A' i preko A'' do B. Na primer, pretpostavimo da je A ručni sistem koji se sastoji od tri glavne funkcije, pri čemu je B automatizovana verzija sistema A. Možemo da definišemo sistem A' kao novi sistem sa automatizovanom funkcijom 1, pri čemu su funkcije 2 i 3 još uvek ručne. Zatim A'' ima automatizovane funkcije 1 i 2, dok je 3 još uvek ručna. Na kraju, B ima sve

tri automatizovane funkcije. Deleći „rastojanje" od A do B na trećine, imamo niz malih problema sa kojima se lakše možemo izboriti nego sa celinom.

U našem primeru, dva sistema su veoma slična, funkcije su iste, ali se razlikuje stil kako su implementirane. Međutim, ciljni sistem je često veoma različit od postojećeg. Povrh svega, obično je poželjno da ciljni sistem nema ograničenja koja postavljaju postojeći hardver ili softver. Pristup **inkrementalnog razvoja** uključuje niz stanja, gde svako sledeće oslobađa prethodno od još jednog takvog ograničenja. Na primer, stanje 1 dodaje novi hardverski deo, stanje 2 vrši zamenu softvera koji izvodi određeni skup funkcija itd. Sistem se polako udaljava od starog hardvera i softvera, sve dok ne bude odražavao novi dizajn sistema.

Stoga, razvoj sistema može prvo da inkorporira skup izmena nad stvarnim sistemom, a zatim ih dopuni nizom izmena koje generišu celokupnu šemu dizajna, umesto pokušaja da se put od sadašnjeg do budućeg stanja pređe u jednom koraku. Sa takvim pristupom, moramo istovremeno da posmatramo sistem na dva različita načina: statički i dinamički. Statički pogled nam govori kako sistem sada radi dok dinamički pogled prikazuje kako se sistem menja u nešto što će na kraju i da postane. Jedan pogled nije potpun bez drugog pogleda.

1.6 INŽENJERSKI PRISTUP

Jednom kada shvatimo prirodu sistema, spremni smo da započnemo njegovo konstruisanje. Na ovom mestu, „inženjerski" deo softverskog inženjerstva postaje relevantan i komplementaran sa onim šta smo do sada uradili. Podsetimo se da smo počeli ovo poglavlje priznanjem da je pisanje softvera jednako umetnost kao i nauka. Umetnost proizvodnje sistema uključuje zanat proizvodnje softvera. Kao umetnici, razvijamo tehnike i alate za koje je dokazano da pomažu u proizvodnji korisnih, visokokvalitetnih proizvoda. Na primer, možemo da upotrebimo prevodilac sa optimizacijom kao alat za generisanje programa koji se brzo izvršavaju na mašinama koje koristimo. Ili možemo da uključimo specijalne rutine za sortiranje ili pretraživanje, kao tehnike za uštedu vremena ili prostora na sistemu. Ove softverske tehnike koriste se na isti način na koji se tehnike i alati koriste u zanatskom delu izrade lepih delova nameštaja ili tokom gradnje kuće. Zaista, popularna zbirka programerskih alata se zove „programerska tezga" (*Programmer's Workbench*), jer programeri koriste te alate kao što stolar koristi stolarsku tezgu.

Shodno činjenici da je gradnja sistema slična gradnji kuće, u graditeljstvu ćemo potražiti još primere koji će pokazati zašto je „umetnički" pristup u razvoju softvera važan.

Gradnja kuće

Pretpostavimo da su Chuck i Betsy Howell unajmili nekoga da im sagradi kuću. Zbog svoje veličine i kompleksnosti kuća, obično je neophodno više ljudi u građevinskom timu. Prema tome, Howellovi su gradnju ugovorili sa građevinskim preduzećem McMullen. Prvi događaj uključen u gradnju kuće je sastanak između Howellovih i McMullena, tako da Howellovi mogu da objasne šta žele. Taj sastanak istražuje ne samo kako Howellovi žele da kuća izgleda, već i koja svojstva treba da ima. Zatim je preduzeće McMu-

llen iscrtalo tlocrte, kao i arhitektonsko rešenje kuće. Nakon detaljne diskusije između Howellovih i McMullena, napravljene su određene izmene. Kada su Howellovi dali svoj pristanak McMullenu, građevinski radovi su započeli.

Tokom procesa gradnje, Howellovi će verovatno posećivati gradilište, razmišljajući o željenim izmenama. Tokom gradnje može doći da više takvih izmena, ali na kraju kuća će biti dovršena. Tokom gradnje i pre nego što se Howellovi usele, proveravane su razne komponente kuće. Na primer, električari su proveravali strujna kola, vodoinstalateri su se uverili da cevi ne cure, a stolari su sredili drvenariju i podove. Na kraju, Howellovi su se uselili. Ako ima nečega što nije ispravno sagrađeno, oni će zvati preduzeće McMullen da to popravi, ali na kraju Howellovi preuzimaju potpunu odgovornost za kuću.

Pogledaćemo detaljnije šta je sve bilo obuhvaćeno procesom. Prvo, mnogo ljudi radi na kući u isto vreme, pa je neophodna dokumentacija. Ne samo tlocrti i arhitektonski crteži, već i detalji moraju biti zapisani, tako da specijalisti, kao što su vodoinstalateri i električari mogu da montiraju svoje proizvode kako bi kuća postala jedna celina.

Drugo, nerazumno je očekivati da Howellovi opišu svoju kuću na početku procesa i jednostavno ušetaju kada je kuća kompletirana. Umesto toga, Howellovi mogu da modifikuju projekat kuće više puta tokom građevinskih radova. Te modifikacije mogu biti rezultat više situacija:

- materijali koji su bili specifikovani na početku nisu više raspoloživi. Na primer, neke vrste krovnih pokrivača više se ne proizvode.
- Howellovi mogu da dobiju nove ideje kada vide kuću koja dobija oblik. Na primer, oni mogu da shvate da mogu da dodaju svetlarnik na kuhinju za malu dodatnu cenu.
- Raspoloživost ili finansijska ograničenja mogu da zahtevaju od Howellovih da izmene zahteve u cilju zadovoljavanja roka ili budžeta. Na primer, posebni prozori koje su Howellovi hteli da naruče neće biti spremni na vreme da bi kuća bila dovršena do zime, tako da će biti zamenjeni prozorima kojih ima na skladištu.
- Pokazuje se da se ne mogu ispoštovati stavke ili zahtevi dizajna o kojima je na početku postignut dogovor. Na primer, ispitivanje vodopropustljivosti tla možda otkrije da zemljište koje okružuje kuću ne može izdržati onaj broj kupatila koje su Howellovi na početku zahtevali.

Preduzeće McMullen takođe može da preporuči neke izmene po započetim građevinskim radovima, možda zbog boljeg rešenja ili zato što ključni član konstrukcionog tima nije bio raspoloživ pre toga. Takođe, i McMullen i Howellovi mogu zajedno da promene mišljenje o nekom svojstvu kuće, čak i nakon što je to svojstvo ugrađeno.

Treće, McMullen mora da obezbedi šematske planove, dijagrame električnih instalacija i vodovoda, uputstva za kućne uređaje i svu drugu dokumentaciju koje će omogućiti Howellovima da naprave modifikacije ili popravke nakon što se usele.

Sumiraćemo ovaj proces gradnje na sledeći način:

- određivanje i analiziranje zahteva;
- izrada i dokumentovanje celokupnog projekta kuće;
- izrada detaljne specifikacije kuće;

- identifikovanje i određivanje komponenti;
- gradnja svake komponente kuće;
- testiranje svake komponente kuće;
- integrisanje komponenti i pravljenje finalnih izmena nakon što se stanari usele;
- nastavljanje održavanja u ime stanara kuće.

Videli smo kako učesnici moraju da ostanu fleksibilni i omoguće izmene prvobitne specifikacije u različitim trenucima tokom izgradnje.

Važno je zapamtiti da se kuća gradi u određenim socijalnim, ekonomskim i pravnim okvirima. Kao što su kod sistema za nadgledanje vodotoka na slici 1.10 opisane međuzavisnosti podsistema, i o kući moramo da razmišljamo kao o podsistemu u okviru neke veće šeme. Na primer, konstrukcija kuće se radi u kontekstu građevinskih propisa i regulativa grada ili okruga. Radnici preduzeća McMullen imaju dozvole od grada ili okruga, i od njih se očekuje da svoje poslove izvode prema građevinskim standardima. Takode, gradilište obilazi građevinski inspektor, koji proverava da li se poštuju standardi. Građevinski inspektori postavljaju standarde u pogledu kvaliteta, dok inspekcije služe kao kontrolni mehanizam koji se stara da građevinski projekat zadovolji određeni kvalitet. Mogu da postoje i socijalna ili običajna ograničenja koja sugerišu uobičajeno ili prihvatljivo ponašanje. Na primer, nije običaj da ulazna vrata direktno vode u kuhinju ili spavaću sobu.

U isto vreme, moramo da priznamo i da ne možemo precizno da propišemo aktivnosti građenja kuće. Moramo da ostavimo mesta za odluke zasnovane na iskustvu, za bavljenje neočekivanim ili nestandardnim situacijama. Na primer, mnoge kuće sastavljene su od već postojećih komponenti. Vrata se isporučuju sa gotovim okvirom, za kupatila se koriste gotove kabine za tuširanje itd. Međutim, standardni proces gradnje kuće može biti izmenjen da bi se prihvatilo neko neuobičajeno svojstvo ili zahtev. Pretpostavimo da je kostur kuće postavljen, kao i suvi zidovi i međuspratovi, a sledeći korak je postavljanje pločica na pod kupatila. Zidari sa zgražanjem otkrivaju da zidovi i pod nisu tačno pod pravim uglom. Taj problem možda nije rezultat lošeg procesa. Kuće se grade iz delova koji imaju svoje prirodno ili fabričko odstupanje, tako da može doći do problema u vezi sa nepreciznošću. Pravougaone podne pločice istaći će nepreciznost ako se postavljaju na uobičajeni način. Međutim, to je mesto gde dolaze na red umetnost i iskustvo. Zidar će postavljati pločice jednu po jednu, praveći mala podešavanja kod svake od njih, tako da sveukupna varijacija bude neprimetna za sve osim najpronicljivijih.

Stoga, gradnja kuće je kompleksan zadatak sa mnogim mogućnostima za usputne izmene u procesima, proizvodima ili resursima, ublažen zdravom dozom umetnosti i iskustva. Proces gradnje kuće može biti standardizovan, ali uvek postoji potreba za ekspertskim procenama i kreativnošću.

Gradnja sistema

Softverski projekti se odvijaju na sličan način kao i proces gradnje kuće. U našem primeru, Howellovi su u ovom slučaju bili i kupci i korisnici, a preduzeće McMullen je sprovodilo razvoj. Da su Howellovi tražili od McMullana da sagradi kuću da bi u njoj živeli roditelji gospodina Howella, tada bi korisnici, kupci i graditelji (razvijaoci) bili

različiti. Po analogiji, razvoj softvera uključuje korisnike, kupce i razvojni tim. Ako se od nas traži da razvijemo softverski sistem za kupca, prvi korak je sastajanje sa kupcem radi određivanja zahteva. Ti programski zahtevi, kao što smo ranije videli, opisuju sistem. Bez poznavanja granica, entiteta i aktivnosti, nemoguće je da opišemo softver, kao i to kako će on biti u interakciji sa okruženjem.

Jednom kada definišemo zahteve, pravimo projekat sistema radi zadovoljavanja zadatih zahteva. Kao što ćemo videti u poglavlju 5, projekat sistema prikazuje kako će sistem izgledati iz perspektive kupca. Stoga, kao što su Howellovi gledali tlocrte i arhitektonske planove, ovde treba da kupcu predstavimo snimke ekrana koji će se koristiti, izveštaje koji će se generisati i sve ostalo što će opisati interakciju u kojoj će biti korisnici sa kompletiranim sistemom. Ako sistem ima ručne rezervne procedure ili zamenske procedure, one se takođe opisuju. U početku, Howellovi su bili zainteresovani samo za izgled i funkcionalnost svoje kuće. To je bilo tako sve dok nisu morali da donesu odluke o pojedinostima, na primer da li žele bakarne ili plastične cevi. Slično tome, faza projektovanja softverskog sistema opisuje samo izgled i funkcionalnost.

Zatim kupac pregleda projekat. Kada ga odobri, projekat kompletnog sistema se koristi za generisanje pojedinačnih potprojekata. Napomenimo da se programi pominju tek u ovom koraku. Sve dok se ne odrede funkcionalnost i izgled, najčešće nema smisla da razmatramo kodiranje. U našem primeru kuće, sada bismo bili spremni da govorimo o tipovima cevi ili kvalitetu električne instalacije. Odluku o plastičnim ili bakarnim cevima možemo da donesemo jer sada znamo gde voda treba da teče. Slično tome, kada su svi odobrili projekat sistema, spremni smo da vodimo raspravu o programima. Osnova za našu raspravu jeste dobro definisani opis softverskog projekta kao sistema. Projekat sistema uključuje kompletan opis funkcija i interakcija.

Kada se programi napišu, pre nego što se povežu u celinu oni se testiraju kao individualni delovi koda. Prva faza testiranja zove se testiranje modula ili testiranje jedinica. Kada se uverimo da ti delovi rade kako želimo, sastavljamo ih u celinu, a zatim treba da proverimo da li ti delovi ispravno rade posle povezivanja sa drugim delovima. Druga faza testiranja često se zove integrativno testiranje, jer smo naš sistem gradili dodajući deo na deo, sve dok čitav sistem nije bio spreman za rad. Finalna faza testiranja, koja se zove testiranje sistema, podrazumeva testiranje celokupnog sistema da bismo se uverili da li su pravilno implementirane funkcije i interakcije koje su zadate na početku. U ovoj fazi, sistem se poredi sa navedenim zahtevima. Razvojni tim, kupac i korisnici proveravaju da li sistem služi svojoj nameni.

Na kraju se finalni proizvod isporučuje. Tokom upotrebe, otkrivaju se neslaganja i problemi. Ako se radi o sistemu „ključ u ruke", kupac nakon isporuke preuzima odgovornost za sistem. Mnogi sistemi nisu tipa „ključ u ruke", tako da razvojni tim ili neka druga organizacija obezbeđuju održavanje, ako nešto krene naopako ili se javi potreba da se nešto izmeni.

Stoga, razvoj softvera obuhvata sledeće aktivnosti:

- analizu i definisanje zahteva;
- projektovanje sistema;
- projektovanje programa;

- pisanje programa (implementaciju programa);
- testiranje jedinica;
- integrativno testiranje;
- testiranje sistema;
- isporuku sistema;
- održavanje.

U idealnoj situaciji, aktivnosti se izvode vremenski jedna po jedna. Kada stignemo do kraja spiska, tada je softverski projekat kompletiran. Međutim, u praksi, mnogi koraci se ponavljaju. Na primer, prilikom pregledanja projekta sistema, zajedno sa kupcem možemo da otkrijemo da neki zahtevi još nisu dokumentovani. Možemo da radimo sa kupcem na dodavanju zahteva i mogućoj izmeni projekta sistema. Slično tome, kada pišemo i testiramo kôd, možemo da otkrijemo da uređaj ne funkcioniše kao što je opisano u dokumentaciji. Možda će biti potrebno preprojektovanje koda, ponovno razmatranje projekta sistema, ili čak ponovo razgovarati sa kupcem o tome kako zadovoljiti konkretan zahtev. Zbog svega toga, **proces razvoja softvera** definišemo kao svaki opis razvoja softvera koji sadrži neku od prethodno nabrojanih devet aktivnosti, organizovane tako da zajedno proizvode proveren kôd. U poglavlju 2, istražićemo nekoliko različitih razvojnih procesa koji se koriste za izradu softvera. Sledeća poglavlja istražiće svaki potproces i njihove aktivnosti, od analize zahteva do održavanja. Pre nego što to uradimo, pogledajmo ko razvija softver i kako se izazov razvoja softvera menjao tokom godina.

1.7 ČLANOVI RAZVOJNOG TIMA

U ovom poglavlju smo već videli da kupac, korisnik i razvojni tim igraju glavne uloge u definisanju i stvaranju novog proizvoda. Razvojni tim se sastoji od softverskih inženjera, ali svaki inženjer može da se specijalizuje za poseban aspekt razvojnog procesa. Videćemo detaljnije uloge članova razvojnog tima.

Prvi korak u svakom razvojnom procesu je utvrđivanje šta korisnik želi i dokumentovanje tih zahteva. Kao što smo videli, analiza je postupak razbijanja stvari na sastavne delove, tako da možemo bolje da ih shvatimo. Stoga, razvojni tim uključuje jednog ili više *analitičara zahteva* u rad sa kupcem, da bi se ono što kupac hoće razložilo u pojedinačne zahteve.

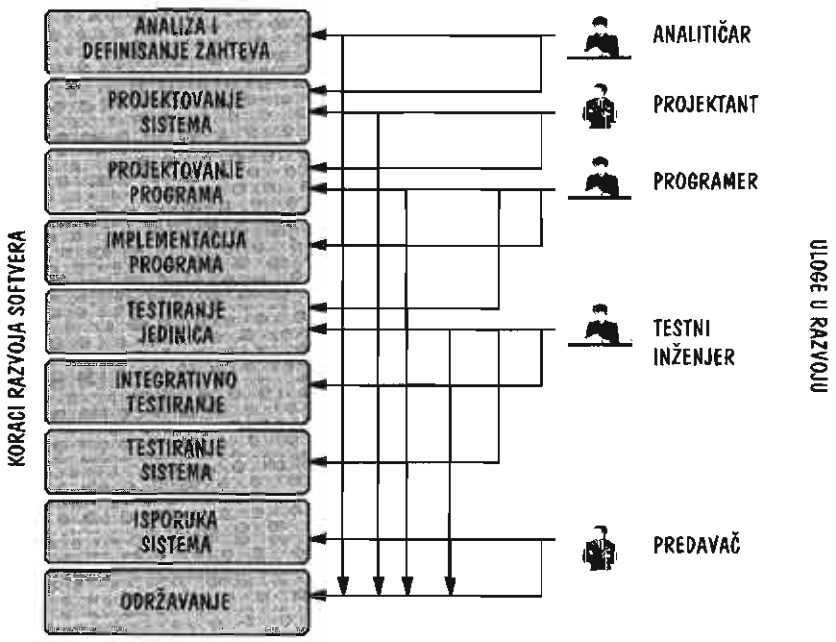
Kada zahtevi postanu poznati i dokumentovani, analitičar radi sa *projektantima* na generisanju opisa funkcija sistema. Dalje, projektanti rade sa programerima na predstavljanju sistema na takav način da *programeri* mogu da pišu kôd koji implementira ono šta je formulisano u zahtevima.

Posle generisanja programskog koda, on mora da se testira. Prvo testiranje često obavljaju sami programeri. Ponekad se koriste drugi *testeri* kao pomoć u otkrivanju grešaka koje previde sami programeri. Kada se jedinice koda integrišu u funkcionalne grupe, timovi za testiranje rade zajedno sa implementacionim timom na verifikovanju ispravnosti rada u skladu sa specifikacijom.

Kada je razvojni tim zadovoljan sa funkcionalnošću i kvalitetom sistema, pažnja se okreće *kupcu*. Tim za testiranje i kupac rade zajedno na verifikovanju celokupnog sistema, poređenjem stvarnog funkcionisanja sistema sa specifikacijom zahteva. Nakon toga, *instruktori* obučavaju korisnike za operativno korišćenje sistema.

U slučaju mnogih softverskih sistema, tehnički prijem od strane kupca ne znači i kraj razvoja. Ako se otkriju greške posle prijema sistema, njih ispravlja *tim za održavanje*. Pored toga, potrebe kupca se mogu menjati tokom vremena, pri čemu na sistemu moraju da se urade odgovarajuće izmene. Stoga, u održavanju mogu učestvovati analitičari koji određuju koje su nove ili izmenjene potrebe, projektanti koji određuju šta u projektu sistema treba izmeniti, programeri za implementiranje izmena, inženjeri za testiranje radi provere da li izmenjeni sistem još uvek ispravno radi, kao i instruktori koji objašnjavaju korisnicima kako se izmena odražava na upotrebu sistema. Slika 1.11 ilustruje u kojim koracima razvoja učestvuju koji pripadnici razvojnog tima.

Studenti često rade samostalno ili u malim grupama, kao razvojni tim za školski projekat. Dokumentacija koju zahteva predavač je minimalna. Od studenata se obično ne traži da pišu korisnička uputstva ili dokumente za obuku. Pored toga, zadaci su prilično nepromenljivi jer se zahtevi ne menjaju tokom životnog veka projekta. Na kraju, sistem koji je napravio student verovatno će biti obrisan na kraju godine. Njegova uloga bila je da demonstrira problem koji se javlja kod stvarnog kupca. Stoga, veličina programa, kompleksnost sistema, potreba za dokumentacijom i potreba za održavanjem praktično ne postoje u školskim projektima.



SLIKA 1.11 Uloge razvojnog tima

Međutim, u slučaju stvarnog korisnika, dimenzije sistema i kompleksnost mogu biti veliki, kao i potreba za dokumentacijom i održavanjem. Kod projekta koji ima više hiljada redova koda i mnogo interakcije između članova razvojnog tima, prilično je teško kontrolisati različite aspekte projekta. Više različitih osoba može biti uključeno u sistem na početku i tokom razvoja, radi podrške svim učesnicima u razvojnom timu.

Bibliotekari pripremaju i skladište dokumentaciju koja se koristi tokom života sistema, uključujući specifikacije zahteva, projektne opise, programsku dokumentaciju, uputstva za obuku, podatke sa testiranja, rokove itd. Sa bibliotekarima rade članovi *tima za upravljanje konfiguracijom*. Zadaci upravljanja konfiguracijom su održavanje korespondencije između zahteva, projekta, implementacije i testova. Unakrsne reference kazuju razvojnom timu koji program treba da se modifikuje ako se vrši izmena u zahtevima, kao i na koje delove programa se odražavaju neke predložene izmene. Osoblje koje upravlja konfiguracijama takođe koordiniše različite verzije sistema koji treba sagraditi i podržavati. Na primer, softverski sistem može biti implementiran na različitim platformama ili se isporučuje u nizu verzija. Upravljanje konfiguracijom obezbeđuje konsistentnu funkcionalnost pri prelasku sa jedne platforme na drugu, kao i očuvanje stepena funkcionalnosti sa novom verzijom.

Razvojne uloge mogu da podrazumevaju jednu osobu ili više njih. Kod malih projekata, dva ili tri čoveka mogu da podele sve uloge. Međutim, kod većih projekata, razvojni tim se često deli na jasno odvojene grupe zasnovane na njihovim funkcijama u razvoju. Ponekad, oni koji održavaju sistem nisu isti oni koji su inicijalno projektovali i napisali sistem. Kod velikog razvojnog projekta, kupac može čak da unajmi jednu kompaniju da uradi inicijalni razvoj, a drugu da vrši održavanje. Kada budemo razmatrali aktivnosti razvoja i održavanja u kasnijim poglavljima, videćemo koje su sposobnosti potrebne za svaku vrstu razvojne uloge.

1.8 KAKO SE IZMENILO SOFTVERSKO INŽENJERSTVO

Uporedili smo pravljenje softvera sa gradnjom kuće. Svake godine, širom zemlje se sagrade stotine kuća u koje se useljavaju zadovoljni kupci. Svake godine, razvojni timovi prave stotine softverskih projekata, ali su kupci vrlo često nezadovoljni rezultatom. Gde je tu razlika? Kad je tako lako nabrojati korake u razvoju sistema, zašto onda softverski inženjeri imaju problema prilikom pravljenja kvalitetnog softvera?

Prisetimo se našeg primera gradnje kuće. Tokom procesa gradnje, Howellovi su neprestano pregledali planove. Oni su takođe imali mnogo prilika da promene mišljenje o onome što su želeli. Na isti način, razvoj softvera omogućava kupcu da pregleda planove na svakom koraku i da napravi izmene u projektu. Nakon svega, ako razvojni tim proizvede odličan proizvod koji ne zadovoljava korisnikove potrebe, svi su uzalud utrošili i vreme i trud na rezultujući sistem.

Stoga je od izuzetnog je značaja da se alati i tehnike za softversko inženjerstvo koriste prilično fleksibilno. U prošlosti, tokom razvoja pretpostavili smo da kupac zna od samog početka šta želi. Ta nepromenljivost nije uobičajena. Kako se dostižu razna stanja projekta, ograničenja se pojavljuju tamo gde na početku nisu bila očekivana. Na primer, pošto smo izabrali hardver i softver za neki projekat, možda shvatimo da su promenjeni zahtevi onemogućili da se naprave meniji na osnovu određenog sistema za upravljanje

bazom podataka tačno onako kako je obećano korisniku. Ili možda otkrijemo da je drugi sistem, sa kojim naš treba da ima vezu, promenio proceduru ili format očekivanih podataka. Može se čak dogoditi da hardver i softver ne rade baš onako kako je obećano u dokumentaciji proizvođača. Zato ne treba zaboraviti da je svaki projekat priča za sebe i da alati i tehnike moraju biti odabrani tako da odražavaju ograničenja koja postavlja konkretan projekat.

Takođe, treba imati u vidu da većina sistema ne radi samostalno već u vezi sa drugim sistemima; bilo primajući od njih informacije, ili ih za njih obezbeđujući. Razvoj takvih sistema je kompleksan jednostavno zato što zahteva veliku koordinaciju između sistema koji međusobno komuniciraju. Ta kompleksnost se posebno iskazuje kod sistema koji se paralelno razvijaju. U prošlosti, razvoj nije lako obezbeđivao tačnost i zaokruženost dokumentacije za međusistemski interfejs. U sledećim poglavljima bavićemo se pitanjem kontrolisanja interfejsa.

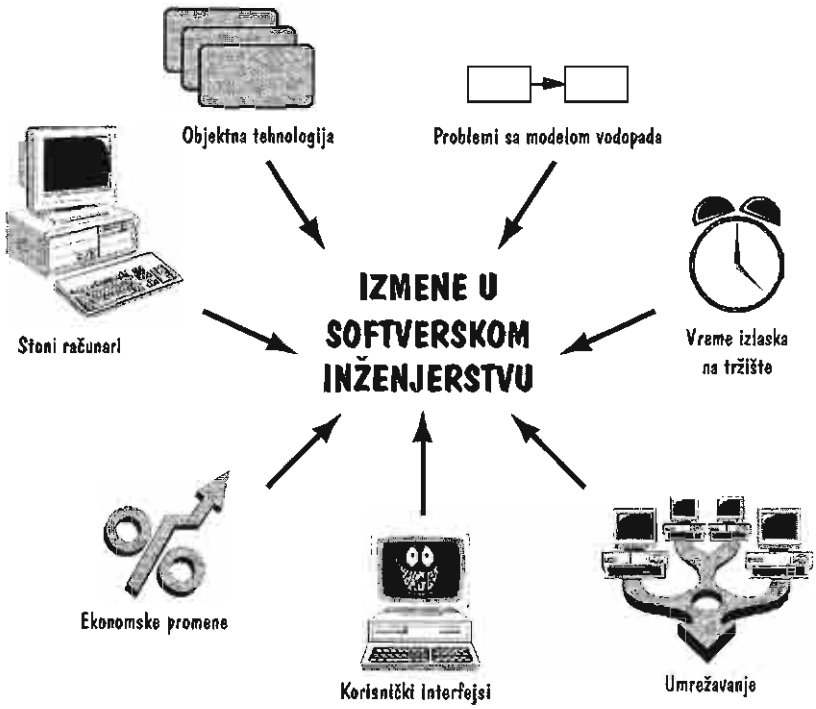
Priroda izmene

Ovi problemi su jedni od mnogih koji utiču na uspeh razvoja softverskog projekta. Koji god pristup da odaberemo, moramo da gledamo i unazad i unapred. Moramo da gledamo unazad, na ranije razvijene projekte, da bismo videli šta smo naučili, ne samo o obezbeđivanju kvaliteta softvera, već i o efikasnosti tehnika i alata. Unapred gledamo da bismo anticipirali verovatni razvoj i korišćenje softverskih proizvoda u budućnosti, što će izmeniti naš način rada. Wasserman (1995) ukazuje na to da su izmene od 1970-ih postale dramatične. Na primer, prve aplikacije su razvijane sa ciljem da se izvršavaju na jednoprocesorskom, obično velikom, računaru. Ulaz je bio linijski, obično preko kartica ili trake, dok je izlaz bio alfanumerički. Sistem je bio projektovan na jedan od dva osnovna načina: kao **transformacija**, gde je ulaz konvertovan u izlaz, ili kao **transakcija**, gde je ulaz određivao koja će funkcija biti izvršena. Današnji sistemi zasnovani su na softveru, veoma su raznovrsni i znatno složeniji. Po pravilu se izvršavaju na više sistema, ponekad su konfigurisani u klijent-server arhitekturi, sa distribuiranom funkcionalnošću. Softver ne obavlja samo primarne funkcije namenjene korisniku, već upravlja mrežom, zaštitom, prezentacijom i obradom korisničkog interfejsa, kao i podacima ili objektima. Tradicionalni pristup razvoju po modelu „vodopada" (naivni pristup, gde razvoj liči na mirnu plovidbu od tačke do tačke), pretpostavlja linearnu progresiju razvojnih aktivnosti, gde jedna počinje samo kada se prethodna završi (proučićemo u poglavlju 2), nefleksibilan je i nije pogodan za današnje sisteme.

Wasserman (1996) je rezimirao te izmene identifikujući sedam ključnih faktora koji su izmenili praksu softverskog inženjerstva, što je ilustrovano na slici 1.12:

1. vitalni značaj vremena izlaska na tržište za komercijalne proizvode;
2. ekonomske promene u računarskoj industriji: niže cene hardvera a viši troškovi razvoja i održavanja;
3. raspoloživost moćnih stonih računara;
4. intenzivna upotreba lokalnih i rasprostranjenih mreža;
5. raspoloživost i prihvatanje objektno orijentisane tehnologije;

6. grafički korisnički interfejs oslonjen na prozore, ikone, menije i pokazivače;
7. nepredvidivost modela vodopada,



SLIKA 1.12 Ključni faktori koji su izmenili razvoj softvera

Na primer, tržišni pritisak podrazumeva da preduzetnici moraju plasirati nove proizvode i usluge pre nego što to uradi konkurencija. U protivnom, sudbina poslovanja može biti dovedena u pitanje. Stoga tradicionalne tehnike za pregledanje i testiranje nisu korisne ako zahtevaju veliki utrošak vremena, a su zastoji zbog nedostatka i otkaza retki. Slično tome, vreme koje je ranije trošeno na optimizaciju koda u cilju poboljšanja brzine ili smanjenja zauzeća prostora možda više nije pametna investicija, jer dodatni disk ili memorijski modul mogu da budu značajno jeftinije rešenje problema.

Pored toga, stoni računari snagu razvoja stavljaju u ruke korisnicima, koji svoj sistem koriste za razvoj aplikacija zasnovanih na radnim tabelama i bazama podataka, malih programa pa čak i specijalizovanih korisničkih interfejsa i simulacija. Ovo premeštanje odgovornosti za razvoj znači da ćemo, kao softverski inženjeri, verovatno morati graditi još kompleksnije sisteme nego pre. Slično tome, ogromne mogućnosti umrežavanja raspoložive većini korisnika i učesnika u razvoju, pojednostavljuju pronalaženje željenih informacija i bez posebnih aplikacija. Na primer, pretraživanje World Wide Weba je brzo, lako i efikasno. Korisnik više nema potrebe da piše aplikacije za bazu podataka da bi pronašao ono što mu je potrebno.

Takođe je unapređen i posao projektanata. Objektno orijentisana tehnologija, povezana sa mrežama i repozitorijumima višekratno upotrebljivih komponenti, stavlja pro-

jektantima na raspolaganje mnoštvo takvih modula, koji se trenutno ili vrlo brzo mogu uključiti u nove aplikacije. Takođe i grafički korisnički interfejsi, koji se često razvijaju pomoću posebnih alata, pomažu da komplikovane aplikacije dobiju poznato lice. Zahvaljujući tome što smo postali sofisticirani u načinu na koji analiziramo probleme, sada možemo da podelimo sistem tako da paralelno razvijamo podsisteme, što zahteva model procesa razvoja koji je potpuno drugačiji od modela vodopada. Videćemo u poglavlju 2 da nam je na rasplaganju mnogo opcija u okviru takvog procesa, uključujući one koje omogućavaju izradu prototipova (kako bismo proverili da li su zahtevi korektni i ocenili da li je projekat isplativ) i aktivnosti koje primenjujemo korak po korak. Ti koraci nam pomažu da osiguramo da u zahtevima i projektantskim rešenjima bude što manje nedostataka pre njihove transformacije u programski kôd.

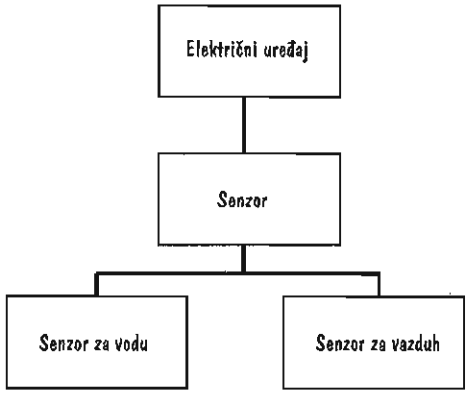
Wassermanova disciplina softverskog inženjeringa

Wasserman (1996) ukazuje na to da svaka od sedam tehnoloških izmena značajno utiče na proces razvoja softvera. Međutim, svih sedam u svojoj ukupnosti promenile su naš način rada. U svojim prezentacijama, DeMarco opisuje taj radikalni pomak tvrdnjom da smo do sada rešili lake probleme, što znači da je one koji su preostali mnogo teže rešiti sada, nego što je to bilo pre. U odgovoru na taj izazov, Wasserman predlaže osam fundamentalnih pojmova u softverskom inženjerstvu koji istinski utemeljuju disciplinu softverskog inženjerstva. Sa njima ćemo se ovde upoznati samo u kratkim crtama, a vratićemo im se u kasnijim poglavljima, da bismo ukazali gde i kako oni mogu da se praktično primene.

Apstrakcija. Ponekad je razmatranje problema u njegovom „prirodnom stanju" (tj. kako ga izražava kupac ili korisnik) prilično obeshrabrujući zadatak. U tom trenutku ne vidimo jasan način da se efikasno ili uopšte uhvatimo u koštac sa problemom. **Apstrakcija** predstavlja opis problema na nekom nivou opštosti koji omogućava usredsređivanje na njegove ključne aspekte, bez bavljenja detaljima. Taj pojam je različit od **transformacije**, gde problem prevodimo u drugo okruženje koje bolje razumemo. Transformacija se često koristi za prevođenje problema iz realnog sveta u matematički svet, tako da do rešenja problema dolazimo proračunima.

Apstrakcija po pravilu obuhvata identifikovanje klasa objekata, koje nam omogućavaju grupisanje pojmova sa ciljem da se usredsredimo na zajedničke osobine klase i time smanjimo broj razmatranih elemenata. Možemo da govorimo o osobinama ili atributima članova klase i da ispitamo odnose između osobina i klasa. Na primer, pretpostavimo da se od nas traži da napravimo sistem za nadgledanje prirodne sredine neke velike i složene reke. Opremu za nadgledanje čine senzori za kvalitet vazduha i zakvalitet vode, za merenje temperature, brzine i drugih karakteristika životne sredine. Ali, za naše potrebe, definisaćemo klasu „senzor", kod koje svaki član klase ima određene osobine, nezavisne od parametra koji nadzire: visinu, težinu, zahteve napajanja, plan održavanja i sl. Bavimo se klasama, umesto elementima, učeći o kontekstu problema, smišljajući rešenje. Na taj način, klase nam pomažu da pojednostavimo problem i pažnju usmerimo na njegove suštinske elemente ili osobine.

Moguće je uspostaviti i hijerarhiju apstrakcija. Na primer, senzor je tip električnog uređaja, pri čemu imamo dve vrste senzora: senzore za vodu i senzore za vazduh.



SLIKA 1.13 Jednostavna hijerarhija opreme za nadgledanje

Na osnovu toga možemo formirati jednostavnu hijerarhiju prikazanu na slici 1.13. Skrivajući neke detalje, možemo da se usredsredimo na suštinsku prirodu objekata kojima moramo da se bavimo, i na taj način dođemo do jednostavnih i elegantnih rešenja. U poglavljima 5, 6 i 7 detaljnije ćemo razmotriti apstrakciju i skrivanje informacija.

Metodi i notacije za analizu i dizajn. Kada projektujemo program koji predstavlja školski zadatak, obično radimo samostalno. Dokumentacija koja se pri tome generiše predstavlja formalni opis vaših zabeleški, koje su namenjene samo vama, o tome zašto ste izabrali određeni pristup, šta predstavljaju imena promenljivih i koji su algoritmi implementirani. Međutim timski rad podrazumeva komunikaciju sa drugim učesnicima u procesu razvoja. Mnogi inženjeri, bez obzira na to kojom vrstom inženjerstva se bave, koriste standardnu notaciju koja im pomaže u komunikaciji i dokumentovanju donetih odluka. Na primer, arhitekta crta nacrte i planove koje onda drugi arhitekti mogu da razumeju. Još važnije je to da zajednička notacija omogućava građevinskim izvođačima da shvate ideje i namere arhitekte. Kao što ćemo videti u poglavljima 4, 5, 6 i 7, u softverskom inženjerstvu ne postoji sličan standard, a pogrešno tumačenje predstavlja jedan od ključnih problema današnjeg softverskog inženjerstva.

Metode analize i dizajna su više od medijuma za komunikaciju. Omogućavaju nam izgradnju modela i proveru njihove sveobuhvatnosti i konsistentnosti. Uz njih mnogo lakše možemo ponovno koristiti zahteve ili komponente iz ranijih projekata i time relativno lako povećati produktivnost i kvalitet.

Međutim, ima mnogo otvorenih pitanja koja treba razrešiti, pre nego što pređemo na zajednički skup metoda i alata. Kao što ćemo videti u kasnijim poglavljima, različiti alati i tehnike odnose se na različite aspekte problema pa je neophodno identifikovati osnovne elemente modelovanja, koji će nam omogućiti da primenom jedne tehnike obuhvatimo sve važne aspekte problema. U suprotnom, moramo razviti tehniku predstavljanja koja se, uz moguća prilagođavanja, može koristiti u svim metodima.

Pravljenje prototipa korisničkog interfejsa. Izrada prototipa podrazumeva realizaciju umanjene verzije sistema, obično sa ograničenom funkcionalnošću, u cilju da:

- korisniku ili kupcu pomognu u identifikaciji ključnih zahteva;
- demonstriraju izvodljivost projekta ili pristupa.

Postupak izrade prototipova je često iterativan. Realizujemo prototip, ocenjujemo ga (pomoću povratne informacije od korisnika i kupca), razmatramo kako izmene mogu da unaprede proizvod ili način projektovanja, a zatim pravimo sledeći prototip. Te iteracije završavaju se kada kupac zaključi da imamo zadovoljavajuće rešenje za problem koji je pred nama.

Izrada prototipa često se koristi za projektovanje dobrog **korisničkog interfejsa**, tj. dela sistema sa kojim korisnik ima interakciju. Međutim, postoje druge mogućnosti za upotrebu prototipova, čak i u **ugrađenim sistemima**, tj. sistemima u kojima softverske funkcije nisu neposredno vidljive korisnicima. Prototip može da prikaže korisniku koje će funkcije biti raspoložive, bez obzira na to da li su implementirane u softveru ili u hardveru. Kako je korisnički interfejs na neki način spona između domena primene i softverskog razvojnog tima, izrada prototipa može da na površinu izbaci pitanja i pretpostavke koje možda nisu bile jasne primenom drugih pristupa u analizi zahteva. Uloga izrade prototipa korisničkog interfejsa biće razmotrena u poglavljima 4 i 5.

Arhitektura softvera. Sveobuhvatna arhitektura sistema je važna ne samo za njegovo lakše implementiranje i testiranje, već i za brzinu i efikasnost održavanja i unošenja izmena. Kvalitet arhitekture može ili da stvori ili da uništi sistem. Shaw i Garlan (1996) su predstavili arhitekturu kao samostalnu disciplinu, čiji se efekti osećaju kroz celokupan proces razvoja. Arhitektonska struktura sistema treba da odražava principe dobrog projektovanja, koje ćemo proučiti u poglavljima 5 i 7.

Arhitektura sistema opisuje sistem oslanjajući se na kategorije koje pripadaju nekom skupu arhitektonskih celina i način na koji se one međusobno povezuju. Što su te celine nezavisnije, to je arhitektura modularnija, pa je nezavisno projektovanje i realizacija tih delova jednostavnija. Wasserman (1996) ukazuje na to da postoji najmanje pet načina kako možemo neki sistem dekomponovati na manje celine:

1. modularna dekompozicija: zasnovana na funkcionalnoj modularizaciji;
2. dekompozicija zasnovana na podacima: zasniva se na spoljašnjim strukturama podataka;
3. dekompozicija zasnovana na događajima: zasniva se na događajima koje sistem mora da prihvati i obradi;
4. dizajn spolja ka unutra: zasnovan na elementima koje korisnik spolja unosi u sistem;
5. objektno orijentisani dizajn: zasnovan na identifikovanju klasa objekata i njihovih veza.

Ovi pristupi se uzajamno ne isključuju. Na primer, možemo da projektujemo korisnički interfejs sa dekompozicijom orijentisan na događaje, dok za projektovanje baze podataka možemo koristiti objektno orijentisani pristup ili pristup koji pokreću podaci. Istražićemo detaljnije te tehnike u sledećim poglavljima. Značaj ovih pristupa je u tome što sumraju naša projektantska iskustva, omogućavajući nam da profitiramo ponovo upotrebljavajući ono što smo ranije uradili, kao i ono što smo tokom tog rada naučili.

Softverski proces. Do kasnih 1980-ih, mnogi softverski inženjeri polagali su posebnu pažnju na *proces* razvoja softvera, kao i na proizvode koji su rezultat tog procesa. Znalo

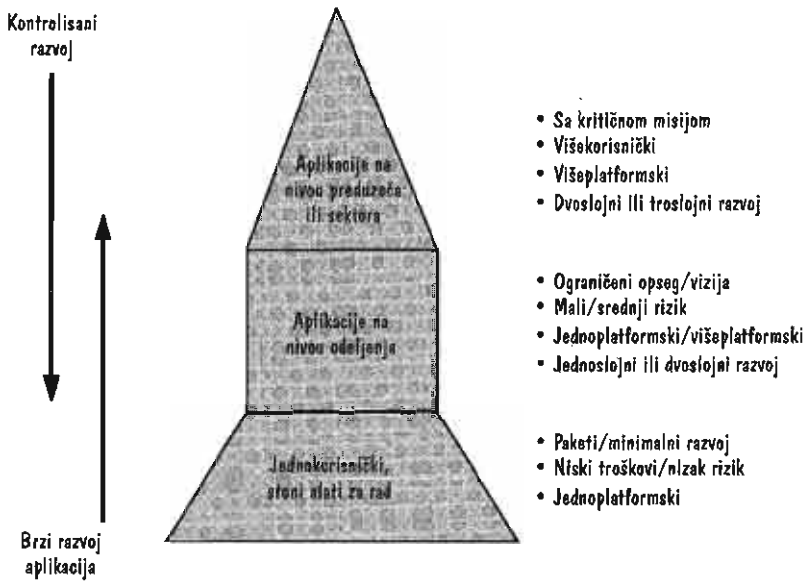
se da organizacija i disciplina u tim aktivnostima doprinose kvalitetu softvera i brzini kojom se softver razvija. Međutim, Wasserman napominje da

> velika raznolikost tipova aplikacija i kultura organizacionih sistema čine nemogućim striktno propisivanje procesa. Stoga se čini da softverski proces za softversko inženjerstvo nema jednako fundamentalan značaj kao što ga imaju apstrakcija i modularizacija (Wasserman 1996).

Umesto toga, on sugeriše da različite vrste softvera zahtevaju različite procese. Wasserman posebno sugeriše da aplikacije koje se primenjuju na nivou celog preduzeća zahtevaju visok stepen kontrole, dok nezavisne aplikacije i aplikacije za potrebe pojedinih odeljenja mogu koristiti prednosti brzog razvoja, kao što je prikazano na slici 1.14.

Koristeći savremene alate, mnoge male i srednje sisteme mogu izgraditi jedan ili dva projektanta, pri čemu svaki od njih mora da preuzme više uloga. U njihove alate spadaju editor teksta, programersko okruženje, podršku za testiranje, možda i mala baza za skladištenje ključnih podataka o proizvodima i procesima. Kako je stepen rizika takvih projekata relativno mali, potrebna je mala podrška u upravljanju i nadzoru.

Međutim, veliki, kompleksni sistemi zahtevaju više uređivanja, provera i usaglašavanja. Takvi sistemi često uključuju veliki broj kupaca i korisnika, a njihov razvoj je dugotrajan. Pored toga, projektanti nemaju uvek potpunu kontrolu nad procesom razvoja, jer neke važne podsisteme mogu isporučiti drugi, ili oni mogu biti implementirani u hardver. Takav tip visoko rizičnog sistema zahteva alate za analizu i projektovanje, upravljanje projektom, upravljanje konfiguracijom, sofisticirane alate za testiranje, kao i još rigorozniji sistem nadzora i analize uzroka. U poglavlju 2 pažljivo ćemo razmotriti nekoliko alternativnih procesa i videti u kakvom su odnosu promene u procesu i različiti ciljevi. Zatim, u poglavljima 12 i 13, procenićemo efikasnost nekih procesa, kao i načine njihovog unapređenja.



SLIKA 1.14 Razlike u razvoju (Wasserman 1996)

РЕПУБЛИКА СРПСКА

Višekratna upotreba. U razvoju i održavanju softvera, često koristimo prednost zajedničkih elemenata u sklopu aplikacija iskorišćavajući ranije razvijene elemente. Na primer, u više razvojnih projekata koristimo isti operativni sistem ili sistem za upravljanje bazama podataka, umesto da svaki put pravimo novi. Kada gradimo sisteme koji su slični, ali ne i identični onima koje smo ranije realizovali, tada možemo iskoristiti postojeće programske zahteve, delove projekta, grupe skriptova za testiranje ili test podataka. Barnes i Boolinger (1991) ukazuju na to da višekratna upotrebljivost nije nova ideja, pri čemu obezbeđuju mnogo zanimljivih primera kako višekratno koristimo ne samo programski kôd, već i mnogo više.

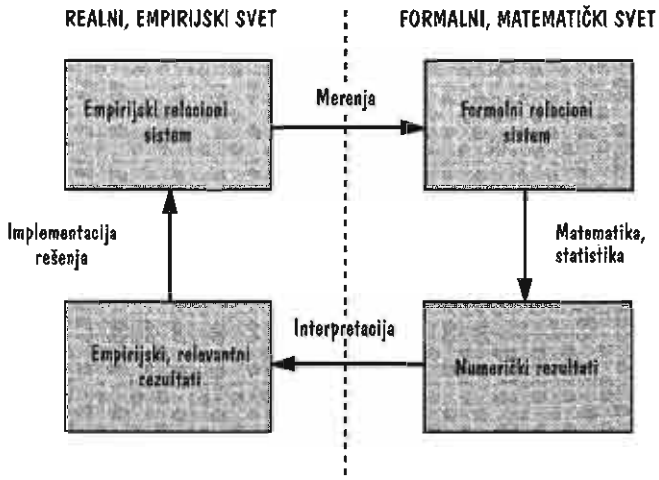
Prieto-Diaz (1991) je uveo pojam višekratno upotrebljivih komponenti kao prednost u poslovanju. Kompanije i organizacije ulažu u elemente koji su višekratno upotrebljivi, a zatim dobijaju dokazanu dobit, u trenucima kada se te stavke iznova koriste u sledećim projektima. Međutim, uspostavljanje dugoročnog i efikasnog programa višekratne upotrebe nije uvek lako, jer postoji nekoliko prepreka:

- Nekada je brže izgraditi malu komponentu, nego tražiti postojeću u skladištu višekratnih komponenti.
- Potrebno je dodatno vreme da bi se komponenta načinila dovoljno opštom da bi bila višekratno upotrebljiva.
- Teško je dokumentovati postignut kvalitet i obavljena testiranja, u takvoj meri da potencijalni korisnik potojeće komponente stekne poverenje u njen kvalitet.
- Nije jasno ko je odgovoran ako komponenta ne radi dobro ili ako je treba prepraviti.
- Potrebno je puno vremena da bi se shvatilo kako radi komponenta koju je neko drugi napisao, što može biti skupo.
- Često postoji sukob između opštih i pojedinačnih osobina.

U poglavlju 12 ponovo ćemo se pozabaviti višekratnom upotrebljivošću i proučiti nekoliko primera uspešnih višekratno upotrebljivih komponenti.

Merenje. Usavršavanje predstavlja pokretačku snagu istraživanja u softverskom inženjerstvu: poboljšanje procesa, korišćenja resursa i metoda u cilju izrade i održavanja boljih proizvoda. Međutim, ponekad se ciljevi usavršavanja izražavaju uopšteno, bez kvantitativne predstave o tome gde smo i kuda idemo. Iz tog razloga, merenje je u softveru postalo ključni element pravilnog rada u softverskom inženjerstvu. Kvantifikujući kad god možemo i šta god možemo, opisujemo akcije i njihove ishode pomoću uobičajenog matematičkog jezika koji nam omogućava da vrednujemo napredak. Pored toga, kvantitativni pristup omogućava poređenje napretka između potpuno različitih projekata. Na primer, dok je John Young bio rukovodilac u Hewlett-Packardu, on je postavio cilj od „10X", tj. desetostruko poboljšanje kvaliteta i produktivnosti svakog projekta u Hewlett-Packardu, bez obzira na tip aplikacije ili oblast primene (Grady i Caswell 1987).

Na nižem nivou apstrakcije, merenja omogućavaju da se vide specifične karakteristike procesa i proizvoda. Često je korisno da naše razumevanje realnog, empirijskog sveta transformišemo u elemente i odnose u formalnom, matematičkom svetu, gde sa njima možemo da manipulišemo radi daljeg razumevanja. Kao što je prikazano na slici 1.15, matematiku i statistiku koristimo za rešavanje problema, posmatranje trendova, ili opisivanje situacije (srednja vrednost i standardna devijacija). Ta nova informacija može da se preslika na realni svet i primeni kao deo rešenja empirijskog problema. U ovoj knjizi videćemo primere kako se merenje koristi za podršku analizi i donošenju odluka.



SLIKA 1.15 Merenja kao pomoć prilikom traženja rešenja

Alati i integrisana okruženja. Godinama su prodavci softvera reklamirali CASE (*Computer-Aided Software Engineering*) alate, kod kojih standardizovano i integrisano razvojno okruženje unapređuje razvoj softvera. Međutim, videli smo kako različiti učesnici u razvoju koriste različite procese, metode i resurse. Što se tiče jedinstvenog pristupa, lakše je reći nego uraditi.

S druge strane, istraživači su predložili nekoliko okvira koji nam omogućavaju da poredimo i suprotstavljamo postojeća i predložena razvojna okruženja. Ti okviri nam dozvoljavaju da istražimo usluge koje pružaju razvojna okruženja za softversko inženjerstvo, i da odlučimo koje je okruženje najpogodnije za dati problem ili razvoj aplikacije.

Jedna od glavnih poteškoća prilikom poređenja alata je činjenica da se njihovi proizvođači retko bave celokupnim razvojnim ciklusom. Oni se obično bave malim skupom aktivnosti, kao što su dizajn ili testiranje, a na samom korisniku je da integriše odabrane alate u kompletno razvojno okruženje. Wasserman (1990) je identifikovao pet pitanja kojima svaki postupak integrisanja alata mora da se pozabavi:

1. integracija platforme: sposobnost alata da radi u heterogenoj mreži;
2. integracija prezentacije: zajednički korisnički interfejs;
3. integracija procesa: povezivanje alata i procesa razvoja;
4. integracija podataka: način na koji alati dele podatke;
5. integracija kontrole: mogućnost da jedan alat nešto saopšti drugom alatu ili inicira neku njegovu akciju.

U svakom od sledećih poglavlja ove knjige, istražićemo alate koji podržavaju aktivnosti i koncepte opisane u ovom poglavlju.

Osam ovde opisanih koncepata mogu biti shvaćeni kao osam niti pomoću kojih je prošivena ova knjiga, i koje povezuju potpuno različite aktivnosti u sklopu discipline koju zovemo softversko inženjerstvo. Kada budemo naučili nešto više o softverskom inženjerstvu, podsetićemo se na te ideje, da bismo videli kako one ujedinjuju i uzdižu softversko inženjerstvo kao naučnu disciplinu.

1.9 PRIMER IZ INFORMACIONOG SISTEMA

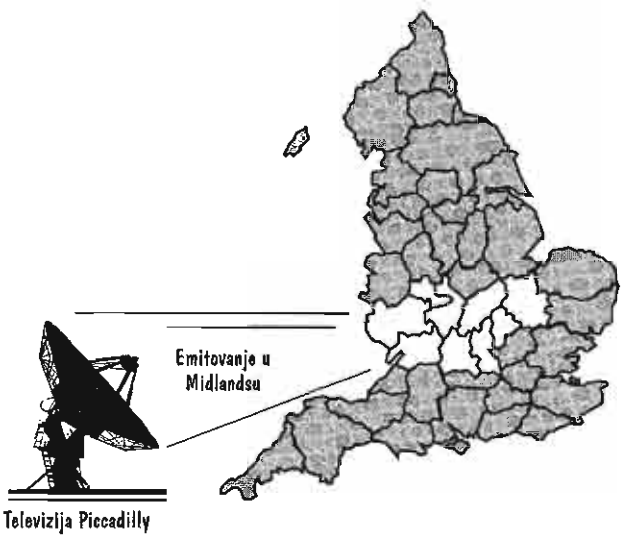
U sklopu ove knjige, videćemo da se svako poglavlje završava sa dva primera, i to prvo primerom iz informacionog sistema, a zatim primerom iz sistema koji radi u realnom vremenu. Koncepte koji su opisani u datom poglavlju, primenićemo na neke aspekte svakog primera, tako da možemo videti šta koncepti znače ne samo teoretski već i u praksi.

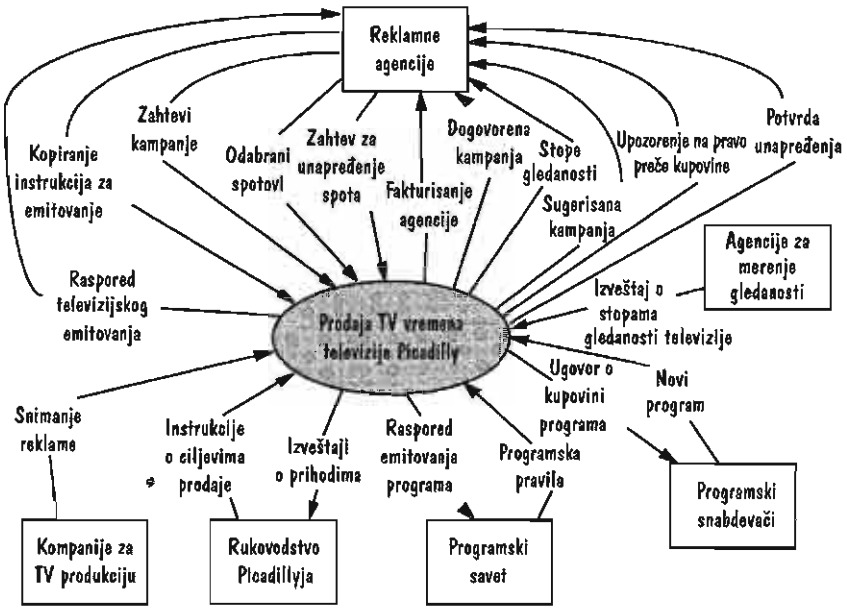
Naš primer iz informacionog sistema je preuzet (sa dozvolom) iz knjige *Complete Systems Analysis: The workbook, the Textbook, the Answers*, autora Jamesa i Suzanne Robertson (Robertson i Robertson 1994). Primer obuhvata razvoj sistema za samooglašavanje stanice Picadilly Television, vlasnika regionalne licence britanske televizije. Slika 1.6 ilustruje oblasti vidljivosti televizije Picadilly. Kao što ćemo videti, ograničenja koja se odnose na cenu televizijskog vremena su mnoga i promenljiva što problem čini zanimljivim i teškim. U ovoj knjizi naglašavamo aspekte problema i rešenje, dok knjiga Robertsonovih detaljno prikazuje metode za prikupljanje i analizu zahteva sistema.

U Velikoj Britaniji radiodifuzna agencija izdaje osmogodišnje licence komercijalnim televizijskim kompanijama, dajući im ekskluzivna prava da emituju svoje programe u okviru pažljivo definisanog regiona ili okruga. Zauzvrat, nosilac licence mora da emituje dramski program, komedije, sport, dečije i ostale programe, u propisanoj ravnoteži. Pored toga, postoje ograničenja o tome koji programi mogu da se emituju u koje vreme, kao i pravila o sadržaju programa i komercijalnog oglašavanja.

Komercijalni oglašivač ima nekoliko izbora da stigne do publike regiona Midlands, a to su: Picadilly, kablovski kanali i satelitski kanali. Međutim, Picadilly privlači većinu gledalaca. Prema tome, Picadilly mora da dostigne određenu stopu gledanosti da bi privukao deo budžeta oglašivača. Jedan od načina da privuče pažnju oglašivača je pomoću indeksa gledanosti koji odražava broj i tip gledalaca u različitim trenucima dana. Indeksi se daju zavisno od tipa programa, tipa publike, vremena u okviru dana, televizijske kompanije itd. Ali stopa oglašavanja ne zavisi samo od stope gledanosti. Na primer, cena po času može biti niža ako oglašivač kupi veći broj časova. Pored toga, postoje ograničenja tipa oglašavanja u određeno vreme i za određene programe. Na primer:

SLIKA 1.16 Oblast za koju televizija Picadilly ima dozvolu emotovanja





SLIKA 1.17 Kontekstni dijagram televizije Picadilly sa prikazanim granicama sistema (Robertson i Robertson 1994)

- Oglasi za alkoholna pića mogu da se prikazuju samo posle 9 uveče.
- Ako neki glumac učestvuje u emisiji, tada reklama sa istim glumcem ne sme da se emituje u periodu do 45 minuta posle emisije.
- Ako je reklama za neku klasu proizvoda (kao što je automobil) raspoređena u određenom reklamnom bloku, onda nijedna druga reklama nečega iz te klase ne može da se prikazuje tokom tog reklamnog bloka.

Kada budemo detaljnije ispitali ovaj primer, napomenućemo dodatna pravila i propise o oglašavanju i ceni. Kontekstni dijagram sistema na slici 1.17 prikazuje granice sistema i u kakvom je odnosu sistem sa datim pravilima. Zasenčena elipsa, koja predstavlja na slici Picadillyjev sistem, je upravo naš primer iz informacionog sistema, pri čemu je granica obvojnica elipse. Strelice i pravougaonici prikazuju stavke koje utiču na rad Picadillyjevog sistema, i posmatraćemo ih kao skup ulaza i izlaza, svaki sa svojim izvorom i odredištem.

U kasnijim poglavljima otkrićemo aktivnosti i elemente koji se nalaze unutar zasenčene elipse (tj. unutar granica sistema). Ispitaćemo izgled projekta i razvoj ovog sistema, koristeći tehnike softverskog inženjerstva koje će biti opisane u sledećim poglavljima.

1.10 PRIMER IZ PRAKSE

Naš primer sistema sa radom u realnom vremenu zasniva se na softveru ugrađenom u svemirskoj raketi Ariane-5, koja pripada evropskoj svemirskoj agenciji ESA (*European Space Agency*). Na letu prvencu, koji se odigrao 4. juna 1996., raketa Ariane-5 bila je

lansirana i ponašala se izvrsno nekih 40 sekundi. Zatim je počela da menja kurs. Prema naredbi zemaljske kontrole Ariane, raketa je uništena daljinskom komandom. Uništenje neosigurane rakete bio je gubitak ne samo rakete, već i četiri satelita koji su se u njoj nalazili. Ukupna šteta bila je oko 500 miliona dolara (Newsbytes matična stranica 1996; Lions i dr. 1996).

Softver je bio uključen u skoro sve aspekte sistema, od vođenja rakete, do internog rada njenih sastavnih delova. Otkazivanje rakete i zatim njeno uništavanje, postavljaju mnoga pitanja o kvalitetu softvera. Kao što ćemo videti u sledećim poglavljima, istražna komisija koja je ispitivala uzroke problema svu pažnju je usmerila na kvalitet softvera i njegovo obezbeđivanje. U ovom poglavlju, posmatrali smo kvalitet zavisno od poslovne vrednosti rakete.

Bilo je mnogo organizacija koje su bile stubovi uspeha Ariane-5. To je pre svega ESA, zatim francuska svemirska agencija CNES (*Centre National d'Etudes Spatiales*), koja je bila zadužena za sveopšte upravljanje programom Ariane, kao i 12 drugih evropskih zemalja. Gubitak rakete bio je samo još jedan događaj u nizu kašnjenja i problema koji su pogodili program razvoja rakete Ariane, kao recimo 1995., kada se dogodilo curenje kiseonika tokom testiranja motora i koje je usmrtilo dva inženjera. Međutim, incident iz juna 1996. je bio prvi koji je imao direktno veze sa otkazom softvera.

Posledice tog incidenta daleko su nadmašile pomenutih 500 miliona dolara izgubljene opreme. U 1996. godini, raketa Ariane-4 i prethodne varijante prednjačile su u lansiranju, ispred američkih, ruskih i kineskih. Stoga su u pitanje dovedeni kredibilitet programa i potencijalni budući poslovi u vezi sa raketama Ariane.

Budući poslovi bili su delom zasnovani na sposobnosti nove rakete da u orbitu nosi teži teret nego što su to mogli prethodni lansirni sistemi. Ariane-5 je bila projektovana da nosi jedan satelit do 6,8 tona, ili dva satelita sa ukupnom težinom od 5,9 tona. U daljem radu na razvoju do 2002. godine, očekivalo se dodavanje još jedne tone lansirnog kapaciteta. Uvećani kapacitet u pogledu nosivosti bio je čista poslovna prednost. Satelitski operatori često smanjuju troškove zajedničkim lansiranjima, što znači da je na raketi Ariane mogao da se lansira teret više kompanija.

Razmotrićemo šta kvalitet znači u kontekstu ovog primera. Ispostavilo se da je uništenje Ariane-5 rezultat loše specifikacije programskih zahteva koje je dao kupac. U ovom slučaju, razvojni tim može da tvrdi da je sistem visokog kvaliteta, ali da je sagrađen po pogrešnoj specifikaciji. Zaista, istražna komisija, formirana radi ispitivanja uzroka i uklanjanja posledica nesreće konstatuje sledeće:

> Komisijski nalazi zasnovani su na temeljnim i otvorenim prezentacijama projektnog tima Ariane-5, kao i na dokumentaciji koja je pokazala visok kvalitet programa Ariane-5 u pogledu inženjerskog rada, u opštem smislu i zaokruženosti i mogućnosti praćenja dokumenata (Lions i dr. 1996).

Međutim, sa stanovišta korisnika i kupaca, postupak izrade specifikacije je trebao da bude dovoljno dobar da prepozna nedostatke u specifikaciji i da natera kupca da ispravi specifikaciju pre nego što je nastala šteta. Istražna komisija potvrdila je sledeće:

> Isporučilac sistema SRI (podsistema u kojem je na kraju lokalizovan uzrok problema), sledio je isključivo specifikaciju koja mu je bila data, a koja je uslovljavala da u slučaju svakog otkrivenog izuzetka procesor treba da se zaustavi. Izuzetak koji se dogodio nije bio zbog slučajnog otkaza, već zbog greške u projektu. Izuzetak je bio prepoznat, ali nije bio pravilno prihvaćen i obrađen, zbog zauzetog stanovišta da se taj softver smatra ispravnim sve dok ne pokaže da ima nedostatak. Komisija ima razloga da veruje da je to stanovište zauzeto i u drugim oblastima softverskog projekta za raketu Ariane-5. Komisija zastupa suprotno stanovište, da za softver treba pretpo-

stavljati da ima nedostatke, sve dok se primenom najboljih postojećih metoda prihvaćenih u praksi ne dokaže da je softver ispravan (Lions i dr. 1996).

Ovim primerom ćemo se detaljnije pozabaviti u kasnijim poglavljima, gde ćemo videti kako odluke koje donose razvojni tim ili kupac utiču na izgled projekta, testiranje i održavanje. Videćemo kako loše sistemsko inženjerstvo na početku razvoja dovodi do niza pogrešnih odluka, koje dalje dovode do katastrofe. S druge strane, otvorenost svih zainteresovanih, uključujući agenciju ESA i istražnu komisiju, uz visokokvalitetnu dokumentaciju i ozbiljnu želju da se što pre dođe do istine, imali su za posledicu brzo rešenje iskrslog problema, uz donošenje efikasnog plana da se takvi problemi u budućnosti spreče.

Sistemski pogled omogućio je istražnoj komisiji, u saradnji sa razvojnim timom, da gleda na raketu Ariane-5 kao na skup podsistema. Taj skup je izraz analize problema, prema načelima opisanim u ovom poglavlju, što znači da različiti učesnici u razvoju mogu da rade na odvojenim podsistemima, sa izrazito odvojenim funkcijama. Na primer:

> Položaj rakete nosača i njeno pokretanje u svemiru mereni su pomoću sistema SRI (*Inertial Reference System*). Taj sistem ima sopstveni računar, gde se uglovi i brzine računaju na osnovu informacija od inercijalne „*strap-down*" platforme, sa laserskim žiroskopom i meračem ubrzanja. Podaci dobijeni od sistema SRI se prenose preko magistrale podataka ka računaru OBC (*On-Board Computer*), na kome se izvršava letački program i kontrolišu mlaznice potisnih motora na čvrsto gorivo, kao i kriogenski Vulcain motor, preko servo ventila i hidrauličkih pokretača (Lions i dr. 1996).

Međutim, sinteza rešenja mora da obuhvati pregled svih sastavnih delova, pri čemu se delovi posmatraju zajedno, radi određivanja da li je „lepak", koji sve te delove drži zajedno, dovoljan i odgovarajući. U slučaju rakete Ariane-5, istražna komisija sugerisala je da kupac i razvojni tim moraju da rade zajedno na pronalaženju kritičnog softvera, kao i na proverama da li softver može da pravilno reaguje ne samo na predviđeno, već i na nepredviđeno ponašanje.

> To znači da se kritični softver (tj. gde otkaz softvera dovodi misiju u pitanje) mora identifikovati na vrlo detaljnom nivou, da se izuzetno ponašanje mora ograničiti, kao i da razumna politika pravljenja rezervnih sistema mora da uzme u obzir i softverske otkaze (Lions i dr. 1996).

1.11 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju uvedeni su mnogi koncepti koji predstavljaju suštinu dobrog softverskog inženjerstva, kako u praksi, tako i u istraživanjima. Od strane pojedinačnih učesnika u razvoju softvera, ti koncepti se mogu koristiti na sledeće načine:

- Kada nam se predoči problem koji treba da rešimo (bez obzira na to da li rešenje uključuje ili ne uključuje softver), možemo ga analizirati dekomponovanjem na sastavne delove i veze između njih. Nakon toga, sintetizujemo rešenje, tako što rešavamo zasebne potprobleme i njihova rešenja integrišemo u jedinstvenu celinu.
- Morate shvatiti da su zahtevi promenljivi, čak i ako ista osoba analizira i rešava problem. Stoga rešenje treba da bude dobro dokumentovano i fleksibilno, a dokumentovati treba sve pretpostavke i algoritme koji se koriste (koje je onda lako kasnije izmeniti).

- Na kvalitet treba gledati iz nekoliko različitih uglova, shvatajući pri tom da tehnički kvalitet i poslovni kvalitet mogu da budu veoma različiti.
- Apstrakcija i merenje su pomoć za identifikovanje suštinskih aspekata problema i rešenja.
- Treba imati na umu i granice sistema, tako da se rešenja ne preklapaju sa sistemima koji su u interakciji sa onim koji mi gradimo.

1.12 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Pretežan deo rada pojedinac obično obavlja kao član većeg razvojnog tima. Kao što smo videli u ovom poglavlju, razvoj obuhvata analizu zahteva, projektovanje, implementaciju, testiranje, upravljanje konfiguracijom, osiguranje kvaliteta i drugo. Pojedinci u timu mogu imati više uloga, pri čemu uspeh projekta u velikoj meri zavisi od komunikacije i koordinacije između članova tima. Videli smo u ovom poglavlju da uspehu projekta možemo da doprinesemo odabiranjem:

- procesa razvoja koji je pogodan za tim određene veličine, nivo rizika i za oblast primene;
- alata koji su dobro integrisani i podržavaju tip komunikacije koji se zahteva za dati projekat;
- alata za merenje i podršku, koji pružaju veći uvid i razumevanje.

1.13 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Mnoga pitanja koja su razmatrana u ovom poglavlju, dobar su predmet za dalja istraživanja. Napomenuli smo neka otvorena pitanja, uključujući potrebu da pronađemo:

- odgovarajuće nivoe apstrakcije da bismo olakšali rešavanje problema;
- odgovarajuća merenja da bismo otkrili suštinsku prirodu problema i rešenja, i da bi oni bili korisni;
- odgovarajuću dekompoziciju problema, gde je svaki potproblem rešiv;
- zajednički okvir ili notaciju za omogućavanje lake i efikasne integracije alata, kao i za maksimizovanje komunikacije između učesnika u projektu.

U kasnijim poglavljima opisaćemo mnoge tehnike. Neke su već korišćene i predstavljaju potvrđenu praksu u razvoju softvera, dok su druge samo predložene i demonstrirane na malim „igračkama" ili studentskim projektima. Nadamo se da ćemo vam pokazati kako da unapredite ono što radite sada i u isto vreme da vas inspirišemo da budete kreativniji i da u budućnosti razmišljate o isprobavanju novih tehnologija i procesa.

1.14 SEMINARSKI RAD

Nemoguće je učiti softversko inženjerstvo bez učešća u razvoju softverskog projekta zajedno sa vašim kolegama. Zato u svakom poglavlju ove knjige opisujemo seminarski rad koji možete da uradite sa timom školskih drugova. Projekat, zasnovan na realnom sistemu u realnoj organizaciji, omogućiće vam da se pozabavite vrlo realnim izazovima koje pred vas postavljaju analiza, projektovanje, implementacija, testiranje i održavanje. Pored toga, radeći u timu bavićete se i pitanjima razlika unutar tima i upravljanja projektom.

Seminarski rad predstavlja utvrđivanje vrste zajma koji možete ugovoriti sa bankom kada želite da kupite kuću. Banke generišu prihod na mnoge načine, često pozajmljujući novac od svojih ulagača po niskoj kamatnoj stopi, a zatim dajući na pozajmicu isti novac po višoj kamatnoj stopi. Međutim, dugoročni krediti za nekretnine, kao što su hipotekarni krediti, obično imaju rok otplate od 15, 25 ili čak 30 godina. To jest, morate 15, 25 ili 30 godina da otplaćujete kredit: glavnicu (novac koji ste prvobitno pozajmili) plus kamatu po određenoj stopi. Iako je prihod od kamata na takve kredite profitabilan, pozajmice dugoročno vezuju novac, sprečavajući banke da koriste svoj novac za druge transakcije. Zbog toga banke često prodaju svoje pozajmice organizacijama za konsolidaciju, uzimajući manji dugoročni profit uzamenu za oslobađanje kapitala koji mogu koristi za druge transakcije.

Aplikacija seminarskog rada zove se „Aranžer pozajmica" (*Loan Arranger*). Oblikovana je na način kako (nepostojeća) organizacija FCO (*Financial Consolidation Organization*) rukuje sa pozajmicama koje kupuje od banaka. Konsolidaciona organizacija pravi novac kupujući pozajmice od banaka i prodajući ih investitorima. Banka prodaje pozajmicu FCO organizaciji, dobijajući glavnicu nazad. Zatim, kao što ćemo videti, FCO prodaje pozajmicu investitorima koji su radi da čekaju duže nego banka da bi im se uloženo vratilo.

Da bismo videli kako ta transakcija radi, razmotrićemo kako se dobija pozajmica za kuću (hipotekarni kredit). Kuću od 150 000 dolara možete da kupite plaćajući 50 000 dolara kao učešće i uzimajući pozajmicu za preostalih 100 000 dolara. Uslovi koje daje First National Bank mogu biti, na primer, pozajmica na 30 godina sa 5% kamate. To znači da First National Bank daje rok od 30 godina (rok pozajmice) za otplatu pozajmljenog iznosa (glavnice) i kamate na sve ono što nije odmah otplaćeno. Na primer, 100 000 dolara može da se otplati uplaćujući jedanput mesečno u roku od 30 godina (tj. 360 rata), sa kamatom na neotplaćeni dug. Ako je početno stanje 100 000 dolara, banka izračunava mesečnu ratu koristeći veličinu glavnice, kamatne stope, dužine vremena u kome treba otplatiti pozajmicu, uz pretpostavku da sva mesečna plaćanja treba da predstavljaju isti iznos.

Na primer, pretpostavimo da banka kaže da mesečna rata treba da bude 536,82 dolara. Kamata za prvi mesec je (1/12) × (0,05) × (100 000), ili 416,67 dolara. Ostatak rate (536,82 - 416,67) je uplata za umanjenje glavnice, a to je 120,15 dolara. U drugom mesecu dug je 100 000 dolara manje 120,15, tako da je kamata umanjena na (1/12) × (0,05) × (100 000 -120,15), ili 416,17 dolara. Tako, tokom drugog meseca, samo 416,17 dolara mesečne rate predstavlja kamatu, a ostatak od 102,65 se odnosi na preostalu glavnicu. Tokom vremena, što se više smanjuje glavnica, plaća se manja kamata, sve dok se ne otplati čitava glavnica a vi postanete vlasnik nekretnine, neopterećene dugom banci.

First National Bank može da proda vašu pozajmicu organizaciji FCO u nekom trenutku tokom perioda otplaćivanja. First National i FCO pregovaraju o ceni. Dalje, FCO može da proda vašu pozajmicu korporaciji ABC Investment. Vi još uvek mesečno otplaćujete hipotekarni kredit, ali uplata ide u ABC, a ne u First National. Obično, FCO prodaje svoje pozajmice „u paketu", a ne pojedinačno, tako da investitor kupuje skup pozajmica, na osnovu rizika, veličine glavnice i očekivane stope povraćaja. Drugim rečima, investitor kao što je ABC može da kontaktira FCO i navede koliko novca želi da investira, na koji period, koji rizik želi da preuzme (na osnovu biografija ljudi ili organizacija koje otplaćuju pozajmicu), i koliki profit očekuje.

Loan Arranger je aplikacija koja omogućava FCO analitičaru da odabere pakete pozajmica koje odgovaraju željama investitora. Aplikacija pristupa informacijama o pozajmicama koje je kupio FCO od raznih kreditnih institucija. Kada investitor odredi investicione kriterijume, sistem odabira optimalne pakete pozajmica koje zadovoljavaju kriterijum. Sistem treba da omogućava neke napredne optimizacije, kao što je biranje najboljih paketa pozajmica iz podskupa raspoloživih (na primer, od svih pozajmica u Massachusettsu, umesto od svih raspoloživih pozajmica). Pored toga, sistem treba ipak da dozvolianalitičaru datog klijenta da ručno bira pakete pozajmica. Sistem takođe automatizuje aktivnosti oko upravljanja informacijama, kao što su ažuriranje informacija o bankama, ažuriranje informacija o pozajmicama, kao i dodavanje novih pozajmica, kada banka jednom mesečno obezbedi takvu informaciju.

Da zaključimo, sistem Loan Arranger omogućava analitičaru pozajmica da pristupi informacijama o hipotekarnim kreditima (za kuće, ovde ih zovemo prosto „pozajmice") koje je FCO kupio od više kreditnih institucija, sa namerom prepakivanja pozajmica radi prodaje drugom investitoru. Pozajmice koje je FCO kupio radi investiranja i dalje prodaje, zbirno se zovu kreditni portfelj. Sistem Loan Arranger zapisuje te hartije od vrednosti u repozitorijumu informacija o pozajmicama. Analitičar pozajmica može da doda, gleda, ažurira ili obriše informaciju o kreditorima skupu pozajmica u portfoliju. Pored toga, sistem omogućava analitičaru da pravi pakete pozajmica radi prodaje investitorima. Korisnik sistema Loan Arranger je analitičar pozajmica, koji vodi računa o hipotekarnim kreditima koje je FCO kupio.

U kasnijim poglavljima istražićemo detaljnije sistemske zahteve. Za sada, ako je potrebno da se podsetite znanja o glavnici i kamati, nije loše da pogledate stare knjige iz matematike, ili da pogledate stranicu http://www.interest.com/hugh/calc/formula.html.

1.15 KLJUČNE REFERENCE

Informacije o softverskim nedostacima i otkazima mogu se pronaći na Risks Forumu, čiji je moderator Peter Neumann. Papirna verzija informacija koje se tamo nalaze štampa se u svakom izdanju *Software Engineering Notes*, čiji je izdavač SIGSOFT (*Special Interest Group on Software Engineering*) u okviru udruženja Association for Computer Machinery. Arhive rizika su raspoložive na ftp.sri.com, u odeljku risks. Konferencijska grupa na Risks Forumu je na internetu, na comp.risks, ili možete da se učlanite preko servera za automatske liste na risks-request@CSL.sri.com.

O projektu Ariane-5 više može da se pronađe na veb lokaciji organizacije European Space Agency, na adresi http://www.esrin.esa.it/htdocs/esa/ariane. Primerak zajedničke izjave za štampu ESA/CNES povodom neuspeha misije nalazi se na adresi http://www.esrin.esa.it/htdocs/tidc/Press/Press96/press19.html (na engleskom) . Francuska verzija izjave za štampu je na adresi http://www.cnes.fr/Acces/Espace/Vol_50x.html. Elektronska kopija dokumenta Ariane-5 Flight 501 Failure Report je na http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

Leverson i Turner (1993) opisali su detaljno softverski projekat i problematiku testiranja mašine Therac.

Izdanje časopisa IEEE Software iz januara 1996. posvećeno je kvalitetu softvera. Naročito u uvodnom članku autori Kitchenham i Pfleeger (1996) opisuju i kritikuju više scenarija za postizanje kvaliteta, dok autor Dromey (1996) u svom članku izlaže o definisanju kvaliteta na merljiv način.

U cilju dobijanja više informacija o primeru televizije Picadilly, pogledajte referencu (Robertson i Robertson 1994), ili istražite pristup Robertsonovih prema utvrđivanju zahteva, na adresi www.systemsguild.com.

1.16 VEŽBE

1. Sledeći članak pojavio se u *Washington Postu* (Associated Press 1996):

KAO RAZLOG PADA AVIONA NAVEDENA JE PILOTOVA RAČUNARSKA GREŠKA. AMERIČKI AVIOPREVOZNIK KAŽE DA JE JEDNO SLOVO U OZNACI BILO RAZLOG UDARA MLAZNJAKA U PLANINU U KOLUMBIJI.

Dalas, 23. avgust – Kapetan mlaznjaka kompanije American Airlines, koji se srušio u Kolumbiji prošlog decembra, uneo je pogrešnu jednoslovnu računarsku komandu koja je poslala avion u planinu, rekao je danas avioprevoznik.

U avionu je bilo 163 putnika, od kojih je samo četvoro preživelo pad.

Istražitelji American Airlinesa zaključili su da je kapetan Boinga 757 očigledno mislio da je uneo koordinate Calija, nameravanog odredišta. Međutim, na većini južnoameričkih aeronautičkih karti, jednoslovni kod za Cali je ista kao i za Bogotu, 132 milje u suprotnom smeru.

Koordinate za Bogotu uputile su avion prema planini, kako kaže u pismu Cecil Ewell, glavni pilot American Airlinesa i potpredsednik zadužen za letenje. Oznake za Bogotu i Cali su različite u mnogim računarskim bazama podataka, rekao je Ewell.

Portparol American Airlinesa, John Hotard potvrdio je da je Ewellovo pismo, prvi put objavljeno u listu *Dallas Morning News*, podeljeno ove nedelje svim pilotima ovog avioprevoznika, kao upozorenje na problem sa kodovima.

Otkriće American Airlinesa podstaklo je Federal Aviation Administration da izda proglas svim avioprevoznicima, upozoravajući ih na neusklađenosti između računarske baze podataka i aeronautičkih karti, kaže se u pomenutom listu.

Računarska greška ne predstavlja poslednju reč o tome šta je uzrok pada. Kolumbijska vlada je vršila istraživanja, a očekuje se da do oktobra objavi svoja saznanja.

Pat Carisco, portparol National Transportation Safety Boarda, kaže da kolumbijski istražitelji ispituju faktore kao što su obuka letačkog osoblja i kontrola letenja.

Računarsku grešku pronašli su istražitelji American Airlinesa, kada su poredili podatke sa navigacionog računara mlaznjaka, sa informacijama iz dubine, kaže Ewell.

Pogrešni podaci ostali su neotkriveni tokom 66 sekundi, dok je posada zbunjeno pratila naredbe kontrole letenja, da bi zauzela direktniji prilaz aerodromu Cali.

Tri minuta kasnije, dok se avion još uvek spuštao, a posada pokušavala da shvati zašto se avion okrenuo, on se srušio.

Ewell kaže da je pad predstavljao dve važne lekcije za pilote.

„Pre svega, nije bitno koliko mnogo puta ste išli u Južnu Ameriku ili neko drugo mesto – Stenovate planine – ne možete nikada, nikada i nikada bilo šta da pretpostavljate", rekao je on u izjavi za štampu. Drugo, dodao je on, piloti moraju da shvate da ne mogu da dozvole da automatizacija preuzme odgovornost za let aviona.

Da li ovaj članak predstavlja dokaz da imamo softversku krizu? Da li je avijacija bogatija zbog softverskog inženjerstva? Kojim pitanjima treba da se bavimo tokom razvoja softvera tako da se problemi poput ovoga spreče u budućnosti?

2. Dajte primer analize problema gde su komponente problema relativno jednostavne, ali poteškoća u rešavanju problema leži u međusobnim vezama između potproblemskih komponenti.
3. Objasnite razliku između greške, nedostatka i otkaza. Dajte primer greške koja dovodi do nedostatka u programskim zahtevima, u načinu projektovanja i u programskom kodu. Dajte primer nedostataka u programskim zahtevima koji dovode do softverskog otkaza, zatim nedostatka u projektovanju koji dovodi do otkaza, kao i nedostatka u podacima za test koji dovodi do otkaza.
4. Zašto broj grešaka može da zavede prilikom merenja kvaliteta proizvoda?
5. Mnogi koji rade u razvoju izjednačavaju tehnički kvalitet sa ukupnim kvalitetom proizvoda. Dajte primer proizvoda čiji je tehnički kvalitet visok, ali ga kupac ne smatra kvalitetnim. Da li postoje etički problemi kada se pitanje kvaliteta sužava samo na tehnički kvalitet? Ilustrujte svoje stanovište na primeru sistema Therac-25.
6. Mnoge organizacije kupuju komercijalni softver, misleći da je tako jeftinije nego da razviju i održavaju softver u okviru kuće. Opišite prednosti i mane korišćenja COTS softvera. Na primer, šta se događa ako proizvođači COTS-ovih proizvoda više ne pružaju podršku? Šta moraju kupac, korisnik i razvojni tim da predvide kada projektuju proizvod koji koristi COTS softver u većem sistemu?
7. Koje su pravne i etičke implikacije upotrebe COTS softvera? Ili saradnje sa podizvođačima? Na primer, ko je odgovoran za ispravljanje problema kada glavni sistem ne radi zbog nedostatka u COTS softveru? Ko je odgovoran kada takav otkaz prouzrokuje štetu korisnicima, bilo direktnu (kao kada otkažu automatske kočnice na kolima), ili indirektnu (kada se pogrešna informacija prosleđuje drugom sistemu, kao što smo videli u vežbanju 1). Kakve su provere i odnosi potrebni za osiguravanje kvaliteta COTS softvera, pre nego što se integriše u veći sistem?
8. Primer televizije Picadilly, kao što je ilustrovano na slici 1.17, sadrži veliki broj pravila i ograničenja. Razmotrite tri od njih i objasnite prednosti i mane činjenice što se nalaze van granica sistema.
9. Kada je uništena raketa Ariane-5, ta vest je došla na prve stranice štampe u Francuskoj i drugde. Francuski list *Liberasion*, nazvao je to na naslovnoj strani „vatrometom od 37 milijardi franaka". U stvari, eksplozija je bila vest sa prvih stranica u skoro svim evropskim novinama, a bila je i glavna vest večernjih dnevnika većine evropskih TV mreža. Nasuprot tome, hakerska invazija na Panix, njujorškog dobavljača internetskih usluga, prouzrokovala je višečasovni ispad i nedostupnost Panix sistema. Vest koja je opisivala ovaj događaj pojavila se samo na naslovnoj stranici privredne rubrike *Washington Posta*. Kakva je odgovornost novinara kada izveštavaju o softverskim incidentima? Kako treba da se ocenjuje potencijalni uticaj softverskih otkaza i da se o njima izveštava?

2 Modelovanje procesa i životnog ciklusa



Videli smo u poglavlju 1 da je softversko inženjerstvo stvaralački i postupan proces, koji okuplja velik broj ljudi na poslovima izrade različitih vrsta proizvoda. U ovom poglavlju, detaljnije ispitujemo korake razvoja i razmatramo načine kako se mogu organizovati aktivnosti u cilju koordinacije poslova koji predstoje. Poglavlje počinjemo definisanjem značenja reči proces, da bismo razumeli o čemu se mora voditi računa prilikom modelovanja razvojnog procesa softvera. Dalje, ispitujemo različite modele tog razvojnog procesa. Nakon upoznavanja sa raznim modelima, detaljnije razmatramo dve tehnike modelovanja: statičko i dinamičko. Na kraju, primenjujemo neke od opisanih tehnika na primere iz informacionog sistema i iz sistema koji radi u realnom vremenu.

2.1 ZNAČENJE TERMINA PROCES

Prilikom pružanja usluga ili izrade proizvoda, bilo da je reč o razvoju softvera, pisanju izveštaja, ili odlasku na službeni put, uvek sledimo niz koraka da bismo izvršili skup zadataka. Zadaci se obično izvršavaju u istom redosledu. Na primer, obično ne malterišemo zidove pre postavljanja električnih instalacija ili ne pečemo kolač pre nego što smo mu dodali sve sastojke. Uređeni skup zadataka možemo smatrati **procesom**: nizom koraka koji obuhvataju aktivnosti, ograničenja i resurse, a rezultiraju željenim ostvarenjem.

Proces obično uključuje skup alata i tehnika, kao što je definisano u poglavlju 1. Svaki proces poseduje sledeće karakteristike:

- Proces propisuje sve glavne aktivnosti.
- Proces koristi resurse, koji podležu skupu ograničenja (kao što je raspored), i rezultira međuproizvodima i finalnim proizvodima.
- Proces može da se sastoji od međusobno povezanih potprocesa. Proces može biti definisan kao hijerarhija drugih procesa, organizovanih tako da je svaki potproces zasnovan na vlastitom modelu.
- Svaka aktivnost u sklopu procesa poseduje uslove za pokretanje i terminiranje.
- Aktivnosti su organizovane u sekvence, tako da je jasno kada se jedna aktivnost izvodi u odnosu na druge.
- Svaki proces poseduje skup uputstava koja objašnjavaju ciljeve svake aktivnosti.
- Ograničenja ili kontrole mogu da se primene na aktivnost, resurse ili proizvod. Na primer, budžet ili plan rada mogu da unesu ograničenja vezana za trajanje neke aktivnosti, kao što korišćeni alat može da ograniči način korišćenja resursa.

Kada se radi o procesu izrade nekog proizvoda, takav proces ponekad nazivamo **životni ciklus**. Zbog toga se nekada softverski razvojni proces naziva i **životni ciklus softvera** jer opisuje „život" softverskog proizvoda od formulisanja, preko implementacije do isporuke, upotrebe, operativnog korišćenja i održavanja.

Procesi su važni jer omogućavaju strukturisanje i konsistentno predstavljanje skupa aktivnosti. To je posebno korisno u situaciji kada posedujemo znanja da nešto uradimo dobro, a želimo da i drugi to urade na isti način. Na primer, ako je Sem dobar zidar, on proces zidanja može da opiše na takav način da na osnovu opisa Sara može da nauči taj posao. On može da uzme u obzir i različite načine na koji ljudi žele da rade iste poslove. Na primer, uputstva mogu biti napisana tako da Sara može da postavlja cigle bez obzira da li to radi levom ili desnom rukom. I proces razvoja softvera može se opisati fleksibilno, tako da omogućava projektovanje i izradu softvera uz oslonac na tehnike i alate koje pojedincima više odgovaraju. Modelovanje procesa može da zahteva projektovanje pre kodiranja, ali i da dozvoli korišćenje širokog spektra različitih tehnika projektovanja. Otud proces pomaže da se održi nivo konsistentnosti i kvaliteta proizvoda ili usluga koje različiti ljudi realizuju.

Proces je složeniji pojam od postupka. U poglavlju 1 smo videli da postupak liči na recept: uređen način kombinovanja alata i tehnika u cilju izrade proizvoda. Proces predstavlja skup postupaka, organizovanih tako da rezultujući proizvod zadovolji unapred postavljeni skup ciljeva i standarda. Proces može da sugeriše izbor između više različitih postupaka, sve dok se time postiže postavljeni cilj. Na primer, proces može da zahteva da proverimo delove projekta pre početka procesa kodiranja. Provera može da se izvede neformalnim pregledom ili formalnom inspekcijom, od kojih je svaki postupak za sebe, ali im je cilj isti.

Struktura procesa upravlja akcijama i omogućava: ispitivanje, razumevanje, kontrolu i poboljšanje aktivnosti koje su njegov sastavni deo. Poslužićemo se primerom pravljenja čokoladnog kolača sa čokoladnim prelivom. Taj proces može da sadrži više postupaka, kao što su: kupovina sastojaka i pronalaženje odgovarajućeg posuda. Recept opisuje postupak za mešenje i pečenje kolača. Recept sadrži aktivnosti (na primer: „ulupati jaje pre mešanja sa ostalim sastojcima"), ograničenja (kao što je „pre mešanja sa šećerom ugrejati čokoladu do tačke topljenja") i resurse (kao što su šećer, brašno, jaja i čokolada).

Pretpostavimo da Chuck peče čokoladni kolač prema ovom receptu. Kada je kolač gotov, on proba uzorak i zaključuje da je kolač suviše sladak. Analizira recept da bi utvrdio koji to sastojci utiču na slatkoću? Šećer! Nakon toga, on peče drugi kolač, ali u novom receptu umanjuje količinu šećera. Ponovo proba kolač, ali on sada nema dovoljno „čokoladni" ukus. U sledećoj verziju dodaje jednu mericu kakaoa u prahu i ponovo proba. Nakon nekoliko iteracija, svaki put menjajući neki sastojak ili akciju (kao što je duže pečenje kolača, ili puštanje da se čokoladna smeša ohladi pre mešanja sa ulupanim jajetom), Chuck postiže da kolač bude po njegovom ukusu. Bez recepta koji dokumentuje proces, Chuck ne bi mogao jednostavno da napravi izmene i proceni njihove efekte.

Procesi su takođe važni jer omogućavaju evidentiranje iskustava i njihovo prosleđivanje drugima. Kao što glavni kuvar prosleđuje svojim kolegama i prijateljima svoje omiljene recepte, tako i vodeći projektanti mogu da prosleđuju dokumentovane procese i postupke. U stvari, pojmovi kao što su podučavanje i mentorstvo zasnivaju se na ideji deljenja iskustva, i omogućavaju prenošenje veština sa iskusnijih inženjera na početnike.

To znači da želimo da učimo na iskustvima ranije razvijenih projekata, dokumentujemo najdelotvorniju izradu softvera visokog kvaliteta i pratimo proces razvoja softvera, da bismo bolje razumeli, kontrolisali i poboljšali ono što se događa u fazi izrade naručenih proizvoda. U poglavlju 1 smo videli da razvoj softvera obično obuhvata sledeće faze:

- analizu i definisanje zahteva;
- projektovanje sistema;
- projektovanje programa;
- pisanje programa (implementacija programa);
- jedinično testiranje;
- testiranje integrisanog rešenja;
- testiranje sistema;
- isporuku sistema;
- održavanje.

Svaka faza sama po sebi predstavlja proces (ili skup procesa) koji se može opisati skupom aktivnosti pri čemu svaka aktivnost uključuje ograničenja, izlaze i resurse. Na primer, prva faza analize i definisanja zahteva kao početni ulaz koristi iskaze, koje je formulisao korisnik na neki od mogućih načina, vezane za tražene funkcije i osobine. Finalni produkt ove faze je skup zahteva, ali to može biti i međuproizvod, jer dijalog između korisnika i razvojnog tima generiše izmene i različite varijante. Postoje i ograničenja, kao što su budžet ili vremenski rok za izradu dokumenta o zahtevima, kao i standardi vezani za tipove uključenih zahteva i, eventualno, notaciju za njihovo iskazivanje.

Ova knjiga se bavi svakom od pomenutih faza. Kod svake faze, detaljno ćemo analizirati procese koji joj pripadaju, resurse, aktivnosti i izlaze, i utvrditi njihov uticaj na kvalitet finalnog proizvoda. Svaka faza razvoja može da se posmatra na više načina. Svaki skup aktivnosti, resursa i izlaza čine proces, a skup procesa definiše šta se u svakoj fazi dešava. Na primer, proces projektovanja može da zahteva izradu prototipa na kome se unapred mogu proveriti mnoge projektantske odluke. Na taj način projektni tim može da

se opredeli za odgovarajući pristup i ponovno korišćenje ranije generisanih komponenti u sklopu aktuelnog projekta.

Svaki proces može da se opiše na različite načine, pomoću teksta, slika, ili njihovom kombinacijom. Istraživači u oblasti softverskog inženjerstva sugerišu različite forme takvih opisa, odnosno organizovane modele koji sadrži ključne odlike procesa. U ostatku ovog poglavlja, razmatramo različite modele procesa razvoja softvera, sa ciljem da utvrdimo način na koji organizovanje aktivnosti unutar procesa može da donese efikasnije projektovanje.

2.2 MODELI PROCESA IZRADE SOFTVERA

U literaturi iz domena softverskog inženjerstva opisani su mnogi modeli procesa. Neki predstavljaju *recepte* za sprovođenje procesa razvoja softvera, dok drugi predstavljaju *opis* načina na koji se razvoj softvera stvarno odvija. Teoretski gledano, ova dva modela bi trebalo da budu ista ili slična, no u praksi, oni to nisu. Izrada modela procesa i razmatranje njegovih potprocesa pomažu projektnom timu da shvati jaz između toga šta bi trebalo da bude i onoga što stvarno jeste.

Postoje i drugi razlozi za modelovanje procesa:

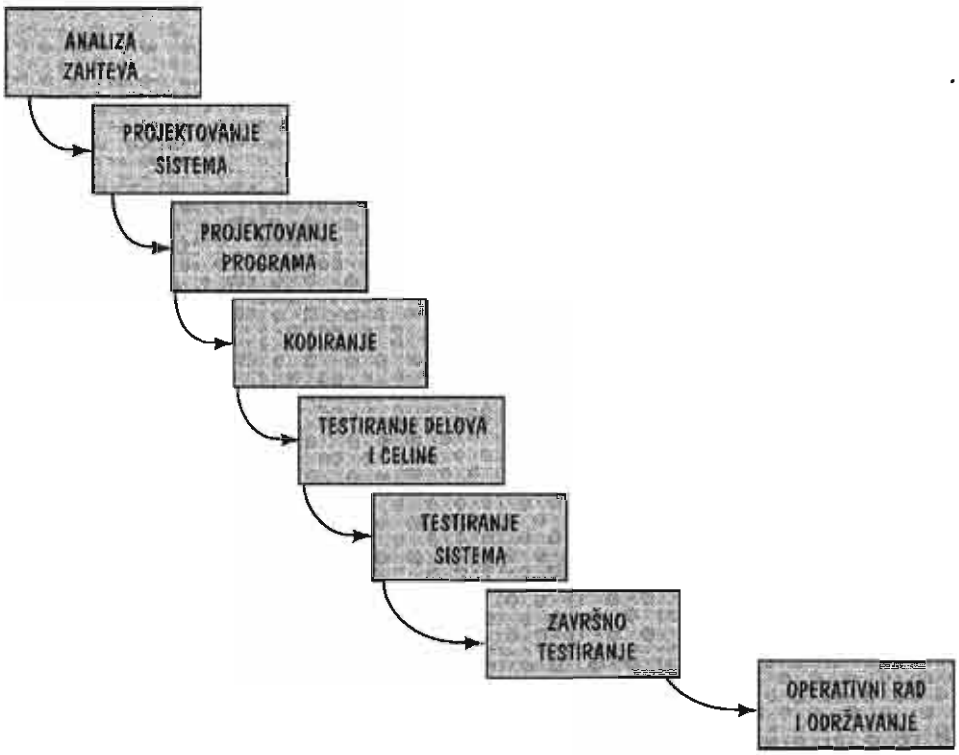
- Kada grupa opiše proces projektovanja koji primenjuje, opis postaje zajedničko shvatanje aktivnosti, resursa i ograničenja uključenih u razvoj softvera.
- Modelovanje procesa pomaže razvojnom timu da pronađe nedoslednosti, viškove i izostavljene elemente u samom procesu ili njegovim delovima. Uočavanjem i otklanjanjem tih nedostataka, proces postaje efikasniji i usmereniji na gradnju finalnog proizvoda.
- Model treba da odražava ciljeve razvoja, kao što je izrada softvera visokog nivoa kvaliteta, otkrivanje grešaka u ranim fazama razvoja i ostajanje u okvirima budžeta i formulisanih ograničenja vezanih za rokove realizacije. Nakon izrade modela, projektni tim ocenjuje aktivnosti sa aspekta njihove usklađenosti sa postavljenim ciljevima. Na primer, tim može da uključi pregled zahteva da bi se uočili i eliminisali problemi, koji su posledica zahteva, pre početka faze izrade dizajna.
- Svaki proces treba da bude „skrojen" prema posebnoj situaciji u kojoj će se primenjivati. Modelovanje procesa pomaže da projektni tim utvrdi mesta na kojima su ta prilagođavanja potrebna.

Svaki model procesa razvoja softvera kao ulaz koristi specifikaciju zahteva, a kao izlaz isporučeni proizvod. Tokom niza godina, predloženi su mnogi takvi modeli. Razmotrimo nekoliko najpopularnijih da bismo bolje razumeli njihove sličnosti i razlike.

Model vodopada

Jedan od prvih predloženih je model vodopada, prikazan na slici 2.1, gde su faze kaskadno prikazane (Royce 1970). Kao što se vidi na slici, jedna faza razvoja treba da se u potpunosti okonča pre nego što započne sledeća. Zbog toga tek kada su svi zahtevi kupca iskazani, analizirani sa aspekta potpunosti i doslednosti, dokumentovani u dokumentu koji sadrži tehničke zahteve, razvojni tim može da nastavi sa aktivnostima u vezi sa di-

zajnom sistema. Model vodopada predstavlja veoam visok nivo apstrakcije razvojnog procesa, pri čemu se projektnom timu sugeriše redosled očekivanih događaja.



SLIKA 2.1 Model vodopada

Model vodopada se koristio za propisivanje aktivnosti u procesu razvoja softvera u različitim kontekstima. Na primer, ovaj model, opisan u sklopu standarda Defence Standard 2167-A, je dugi niz godina bio osnova ugovora za razvoj i isporuku softvera u Ministarstvu odbrane SAD (*U.S. Department of Defence*). Uz svaku aktivnost u sklopu procesa definisani su miljokazi i međuproizvodi da bi rukovodioci projekata mogli na osnovu modele da mere stepen gotovosti projekta. Na primer, „testiranje delova i integrisanosti" u modelu vodopada završava se miljokazom „napisani, testirani i integrisani moduli", dok je međuproizvod primerak testiranog koda. Nakon toga, kôd se predaje inženjerima koji testiraju sistem, gde se on prvo združuje sa ostalim komponentama sistema (hardver ili softver), a zatim testira u sklopu celine.

Model vodopada je veoma koristan kao pomoćno sredstvo za izražavanje onoga što projektni tim treba da uradi. Njegova jednostavnost olakšava pružanje objašnjenja kupcima koji nisu upoznati sa procesom razvoja softvera jer eksplicitno ukazuje na međuproizvode koji su neophodni za započinjanje sledeće faze. Mnogi drugi, složeniji modeli, u stvari predstavljaju „ulepšani" model vodopada, kroz uključivanje povratnih sprega i dodatnih aktivnosti.

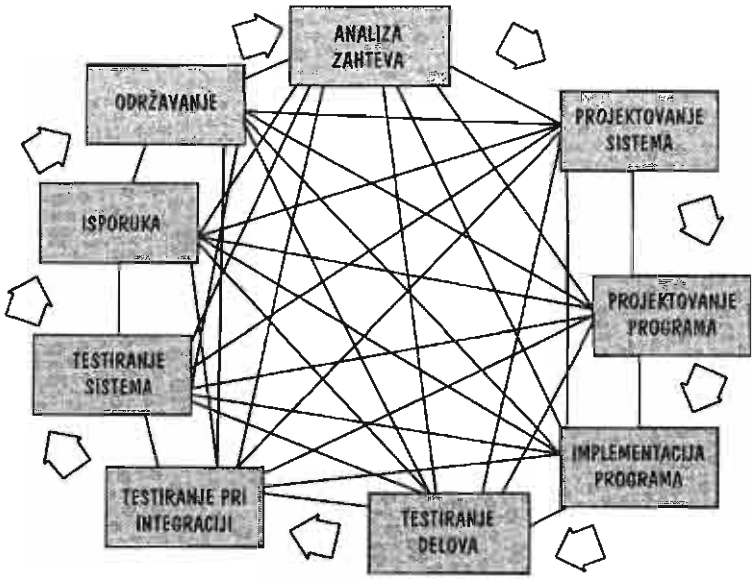
Mnogi problemi koje unosi model vodopada razmatrani su u literaturi, od kojih su dva sumirana u podsetniku 2.1. Najveći problem sa modelom vodopada je u tome što on ne odražava stvarni način na koji se kôd razvija. Osim možda za vrlo jasne probleme, softver se obično razvija kroz veliki broj iteracija. Softver se često koristi za rešavanje problema koji nikada ranije nije bio rešen, ili čije rešenje mora da se unapredi da bi odražavalo promene nastale u poslovnom ili radnom okruženju. Na primer, proizvođač aviona može da zahteva softver za novi avion, koji će biti veći ili brži nego postojeći modeli, tako da se pojavljuju novi izazovi kojima se treba baviti, čak i ako tim koji radi na razvoju softvera ima mnogo iskustva u domenu aeronautičkog softvera. Ni razvojni tim ni korisnici ne znaju sve ključne faktore koji utiču na željeni ishod, pa se značajno vreme u analitičkoj fazi (kao što ćemo pokazati u poglavlju 4) troši na razumevanje elemenata i procesa na koje sistem i njegov softver utiču, kao i odnosa koje sistem uspostavlja sa radnim okruženjem. Stoga, ako stvarni proces razvoja softvera nije pod kontrolom, lako može izgledati kao na slici 2.2. Učesnici u razvoju mogu da skaču sa jedne aktivnosti na drugu i vraćaju se nazad, u pokušajima da upoznaju problem i način na koji se on rešava.

PODSETNIK 2.1 NEDOSTACI MODELA VODOPADA

Još od samog uvođenja, model vodopada doživeo je mnogo kritika. Na primer, Cracken i Jackson (1981) su ukazali da ovaj model nije ništa drugo do upravljačka struktura namenjena procesu razvoja sistema. „Tvrditi da se bilo koja šema životnog ciklusa, čak i sa varijacijama, može primeniti na razvoj svih sistema, predstavlja ili sukob sa realnošću ili pretpostavku da je životni ciklus do te mere jednostavan da je besmislen."

Uočimo da model vodopada pokazuje da se svaka glavna faza razvoja okončava izradom nekog artefakta (kao što su zahtevi, dizajn ili kod). Pri tome ne postoji uvid u to kako svaka aktivnost jedan artefakt transformiše u drugi, npr. zahteve u dizajn. Stoga, model ne obezbeđuje smernice za rukovodioce i projektni tim o tome kako da upravljaju promenama na proizvodu ili u aktivnostima, do kojih će sasvim verovatno doći dok traje razvoj. Na primer, kada dođe do promene zahteva u fazi kodiranja, model vodopada ne konstatuje izmene koje time nastaju u dizajnu i programskom kodu.

Curtis, Krasner, Shen i Iscoe (1987) napominju da je glavna mana modela vodopada nemogućnost da se softver tretira kao proces rešavanja problema. Model vodopada je izveden iz sveta hardvera i odražava proizvođački pogled na proces razvoja softvera. Industrijska proizvodnja podrazumeva izradu pojedinačnih proizvoda i njihovu neograničenu replikaciju. Softver se ne proizvodi na taj način; on evoluira sa razumevanjem problema i očenom varijanti rešenja. Prema tome, proces izrade softvera je stvaralački a ne proizvođački. Model vodopada nam ništa ne govori o ponavljanim aktivnostima „korak napred, korak nazad" koje dovode do izrade finalnog proizvoda. Proces stvaranja obično podrazumeva da se proba malo od ovoga malo od onoga, kao i razvoj i ocenjivanje prototipova, procenu izvodljivosti zahteva, poređenje različitih projektnih rešenja, dobijanje novih saznanja po otkazu sistema, i na kraju zaustavljanje kada se nađe zadovoljavajuće rešenju razmatranog problema.

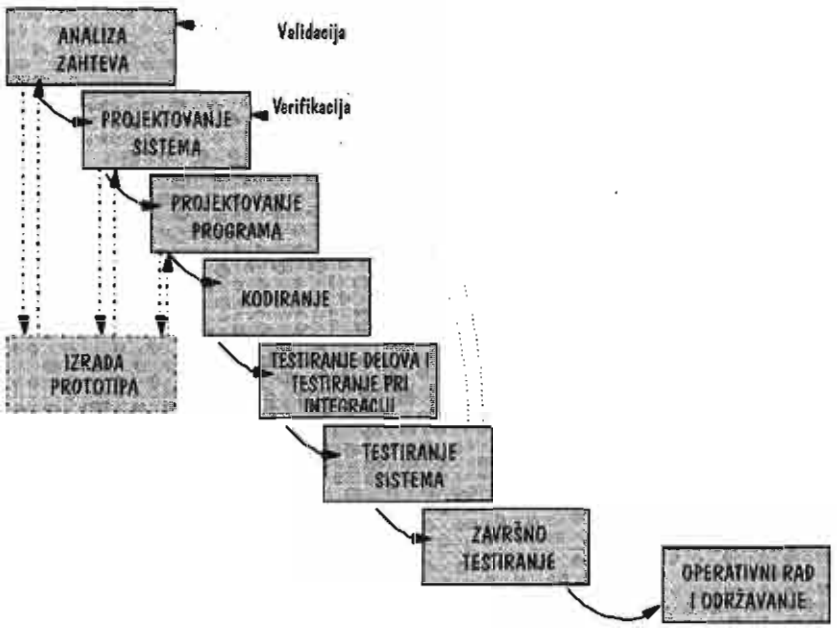


SLIKA 2.2 Proces razvoja softvera u stvarnosti

Proces razvoja softvera, kroz uključenje aktivnosti i potprocese koji doprinose boljem razumevanju, pomaže u procesu kontrolisanja „škarta". Izrada prototipova je jedan takav potproces. Prototip predstavlja delimično razvijen proizvod koji omogućava naručiocima i projektantima da ispitaju neke aspekte predloženog sistema i odluče da li je on pogodan ili potreban u sklopu gotovog proizvoda. Na primer, projektni tim može da izgradi sistem koji implementira jedan deo ključnih zahteva, da bi se ustanovila njihova doslednost, izvodljivost i praktična vrednost. Ako se ispostavi da zahtevi nisu dosledni, izvodljivi, itd., oni se revidiraju, i time se izbegavaju mnogo skuplje revizije u fazi testiranja. Slično tome, od delova projekta mogu da se naprave prototipovi, kao što je prikazano na slici 2.3. Izrada prototipova u fazi projektovanja pomaže projektnom timu da oceni alternativne strategije projektovanja i odluči koja je najbolja za određeni projekat. Kao što ćemo videti u poglavlju 5, projektanti mogu da dizajniraju zahteve na potpuno različite načine, da bi videli koji od njih poseduje najbolje osobine. Na primer, mreža može da se, u jednom prototipu, projektuje u topologiji prstena a u drugom u topologiji zvezde, pri čemu se ocenjuju performanse u cilju utvrđivanja koja je topologija primerenija za postizanje postavljenih ciljeva u datim ograničenjima.

Često se korisnički interfejs gradi i testira u formi prototipa, sa ciljem da korisnici shvate izgled novog sistema a projektanti steknu bolji utisak o tome kakvu interakciju sa sistemom korisnici žele. Prema tome, osnovne poteškoće u zahtevima uočavamo i ispravljamo znatno pre nego što se one zvanično validiraju u fazi testiranja sistema. **Validacija** obezbeđuje potpunost implementacije zahteva u sklopu sistema, tako da se svaka funkcija sistema može propratiti i povezati sa određenim zahtevom definisanim u specifikaciji zahteva. Testiranjem sistema zahtevi se verifikuju; **verifikacija** osigurava ispravan rad svake funkcije. U sklopu validacije proveravamo da li je razvoj rezultirao „pravim" sistemom (usklađenim sa specifikacijom), dok verifikacija proverava kvalitet implementacije.

Izrada prototipa je korisna za verifikaciju i validaciju, ali te aktivnosti mogu da se vrše i u drugim fazama procesa razvoja softvera, kao što ćemo videti u sledećim poglavljima.

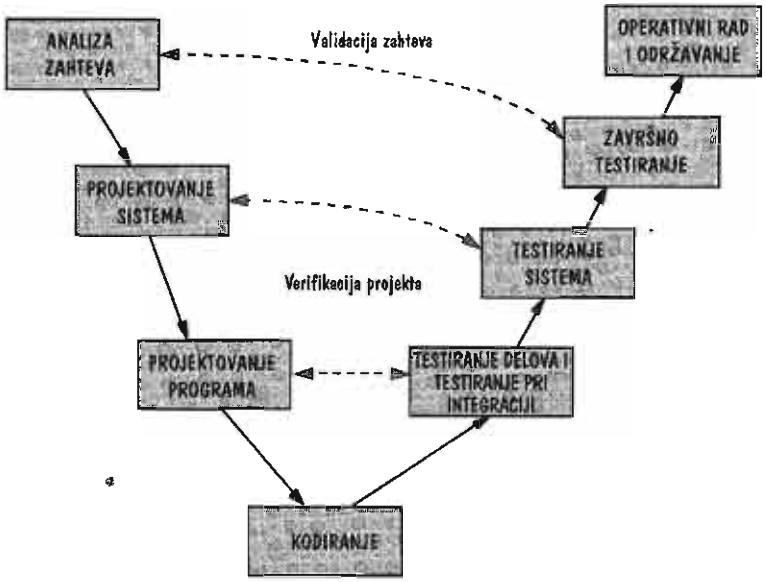


SLIKA 2.3 Model vodopada

V model

V model je modifikovani model vodopada, koji pokazuje odnos testiranja i faza analize i dizajna (prema namačkom Ministarstvu odbradne, 1992). Kao što je prikazano na slici 2.4, kodiranje predstavlja ishodište V modela, pri čemu su analiza i dizajn na levoj, a testiranje i održavanje na desnoj strani. Testiranje delova i testiranje pri integrisanju bave se ispravnošću programa, kao što ćemo videti u kasnijim poglavljima. V model sugeriše da testiranje delova i testiranje pri integraciji mogu da se koriste za verifikovanje dizajna programa. To jest, tokom testiranja delova i testiranja pri integrisanju, programeri i članovi testerskog tima treba da obezbede ispravno implementirani u kōdu svih dizajnerskih aspekata programa . Slično tome, testiranje sistema treba da verifikuje dizajn sistema, uz potvrdu da su svi aspekti projektovanog sistema ispravno implementirani. Završni test, koji češće sprovode naručioci nego projektni tim, služi validaciji zahteva. Pri tome se svakom elementu iz specifikacije zahteva dodeljuje po jedan korak u procesu testiranja. Završnim testiranjem se proverava da li su svi zahtevi potpuno implementirani, pre nego tehničkog prijema sistema i plaćanja.

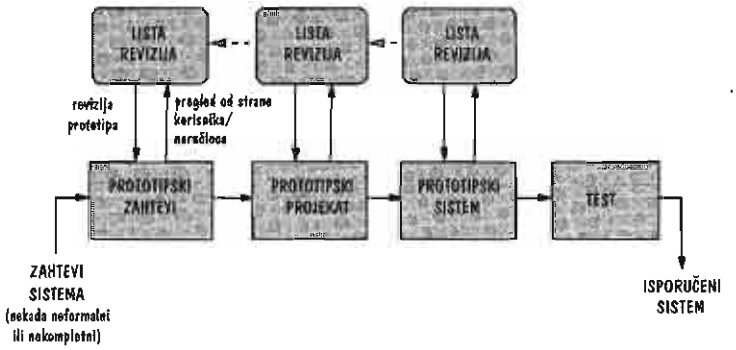
Veze leve strane sa desnom stranom u sklopu V modela ukazuju da, ako se pronađu problemi tokom verifikacije i validacije, tada se aktivnosti sa leve strane V modela mogu ponoviti radi popravke i poboljšanja zahteva, dizajna ili koda, pre nego što se ponove koraci testiranja definisani na desnoj strani. Drugim rečima, V model čini još eksplicitnijim neke iteracije i prepravke koje su sakrivene u modelu vodopada. U središtu pažnje modela vodopada često su dokumenti i artefakti, dok se V model usredsređuje na aktivnosti i ispravan rad.



SLIKA 2.4 V model

Prototipski model

Videli smo kako model vodopada može da se izmeni dodavanjem aktivnosti izrade prototipova u funkciji boljeg razumevanja zahteva. Međutim, izrada prototipa ne treba samo da bude dopuna modela vodopada. Ona sama po sebi može da bude osnov za efikasno modelovanje procesa, kao što je prikazano na slici 2.5. Kako prototipski model omogućava da se kompletan sistem ili delovi sistema brzo konstruišu radi razjašnjenja ili boljeg razumevanja otvorenih pitanja, ovaj model ima isti cilj kao inženjerski prototip, gde zahtevi ili dizajn iniciraju ponovne analize da bi se osiguralo da projektni tim, korisnik i naručilac postignu zajednički stepen razumevanja stvarnih potreba i predložene funkcionalnosti. Jedna povratna sprega ili više njih kod izrade prototipa zahteva, dizajna ili sistema mogle bi da se eliminišu, zavisno od ciljeva zbog kojih se prototip izrađuje. Međutim, ukupni cilj ostaje isti: smanjenje rizika i neodređenosti prilikom projektovanja.



SLIKA 2.5 Prototipski model

Na primer, razvoj sistema može da započne sa nominalnim skupom zahteva podnetih od strane naručilaca i korisnika. Zatim se istražuju alternative, tako što zainteresovane strane analiziraju moguće ekrane, tabele, izveštaje i druge izlaze iz sistema koje direktno koriste korisnici i kupci. Kako korisnik i naručilac definišu svoje želje, dolazi do revizije zahteva. Kada se postigne dogovor o zahtevima, projektni tim prelazi na dizajn. Zatim se istražuju varijante u dizajnu, često uz konsultaciju sa korisnikom i naručiocem.

Inicijalni dizajn se revidira sve dok projektni tim, korisnici i naručilac nisu zadovoljni rezultatom. Razmatranje varijanti dizajna ponekad ume da otkrije problem koji potiče od zahteva. Tada se projektni tim vraća na izrađene zahteve, ponovo ih razmatra i menja specifikaciju zahteva. Najzad, sistem se kodira uz razmatranje varijanti, sa mogućim ponovnim iteracijama kroz zahteve i dizajn.

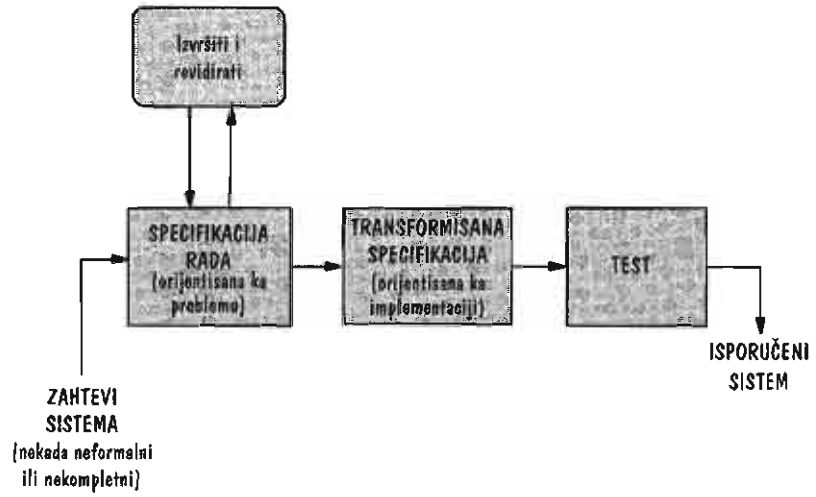
Specifikacija rada

Kod mnogih sistema, neodređenost vezana za zahteve dovodi do izmena i problema u kasnijem razvoju. Zave (1984) sugeriše takav procesni model koji omogućava projektnom timu i naručiocu da u prvim fazama projektovanja ispitaju zahteve i njihove implikacije, pri čemu se diskutuju i moguće razrešavaju neke neodređenosti. U **modelu specifikacije rada** sistemski zahtevi se ocenjuju ili izvršavaju na način koji pokazuje prirodu sistema. To jest, kad zahtevi postoje, oni se mogu aktivirati oslanjajući se na programski paket, da bi se ocenilo njihovo potencijalno značenje pre početka dizajna. Na primer, ako prema specifikaciji predloženim sistemom treba da rukuje 24 korisnika, izvršni oblik specifikacije će pomoći analitičarima da odrede da li taj broj korisnika možda preopterećuje sistem.

Ovaj model je vrlo različit od modela vodopada i ostalih tradicionalnih modela. Model vodopada razdvaja funkcionalnost sistema od dizajna (tj. ono što sistem treba da radi, odvojeno je od načina na koji sistem to radi), sa namerom da potrebe naručioca čuva odvojene od implementacije. Međutim, model specifikacije rada dopušta da se funkcionalnost i dizajn objedine. Slika 2.6 ilustruje kako model specifikacije funkcioniše. Primetimo da je model specifikacije rada sličan izradi prototipa. Proces omogućava korisniku i projektnom timu da ispitaju zahteve u ranim fazama projektovanja.

Transformacioni model

Balzerov **transformacioni model** pokušava da redukuje mogućnost greške eliminacijom nekoliko ključnih koraka u razvoju. Uz pomoć automatske podrške, u ovom modelu primenjuje se niz transformacija kojima se specifikacija pretvara u sistem koji može da se isporuči (Balzer 1981a).



SLIKA 2.6 Model specifikacije rada

Tipične transformacije su:

- izmena predstavljanja podataka;
- izbor algoritama;
- optimizacija;
- prevođenje.

Od specifikacije do isporučenog sistema ima više puteva, te se stoga sekvence transformacija i odluka koje ga predstavljaju čuvaju kao formalni zapis u sklopu dizajna.

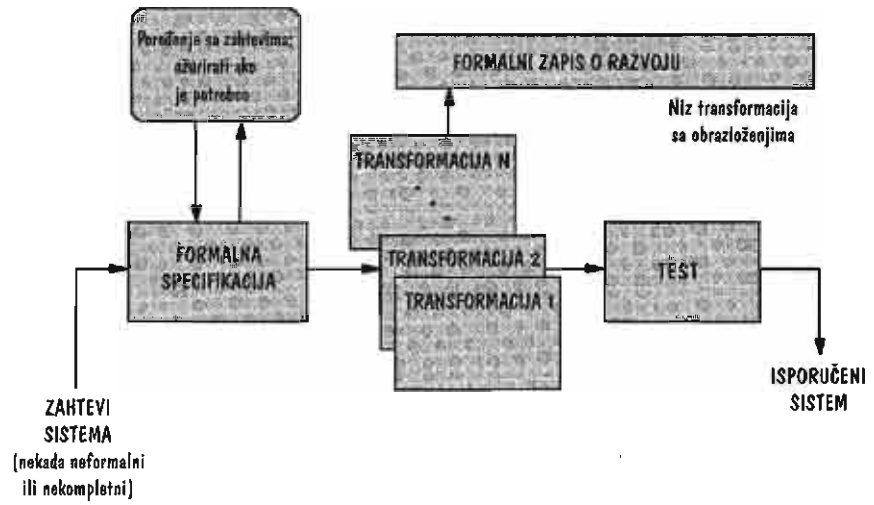
Transformacioni pristup veoma mnogo obećava. Međutim, glavna smetnja njegovoj upotrebi je neophodnost precizne formalne specifikacije da bi transformacije mogle da se nad njom izvršavaju, kao što je prikazano na slici 2.7. Sa sve većom popularnošću formalnih specifikacijskih modela, transformacioni model bi mogao da bude šire prihvaćen.

Fazni razvoj: Inkrementi i iteracije

U početnim godinama razvoja softvera naručioci su bili spremni da duže čekaju na završetak razvoja softverskih sistema. Nekada su prolazile godine od trenutka pisanja dokumenta sa zahtevima, do isporuke sistema, što je nazivano **trajanje ciklusa**. Međutim, današnja poslovna okruženja više ne tolerišu duga čekanja. Softver pomaže da se proizvodi razlikuju na tržištu, a naručioci uvek traže nov kvalitet i funkcionalnost. Na primer, u 1996. godini, 80% prihoda Hewlett-Packarda dobijeno je od proizvoda uvedenih u prethodne dve godine. Stoga se razvijaju novi procesni modeli, koji treba da pomognu da se smanji trajanje ciklusa.

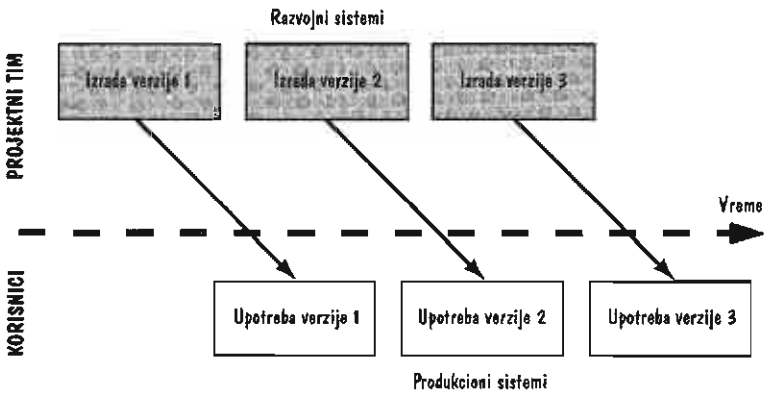
Jedan način za smanjenje trajanja ciklusa jeste fazni razvoj, kao što je prikazano na slici 2.8. Sistem se projektuje tako da može da se isporučuje u delovima, omogućavajući korisnicima da imaju deo funkcija, dok je ostatak još u razvoju. Tako obično postoje dva sistema koja uporedo funkcionišu: produkcioni sistem i razvojni sistem. **Operativni**

ili **produkcioni sistem** je onaj koji trenutno koriste naručilac i korisnik. **Razvojni sistem** predstavlja sledeću verziju koja se priprema da zameni postojeći produkcioni sistem. Sisteme često nazivamo prema rednom broju njihove verzije (izdanja): razvojni tim gradi verziju 1, testira je, a zatim predaje korisnicima kao prvu produkcionu verziju. Zatim, dok korisnici koriste verziju 1, projektni tim gradi verziju 2. Tako, projektni tim uvek radi na verziji $n + 1$, dok je verzija n u fazi operativnog korišćenja.

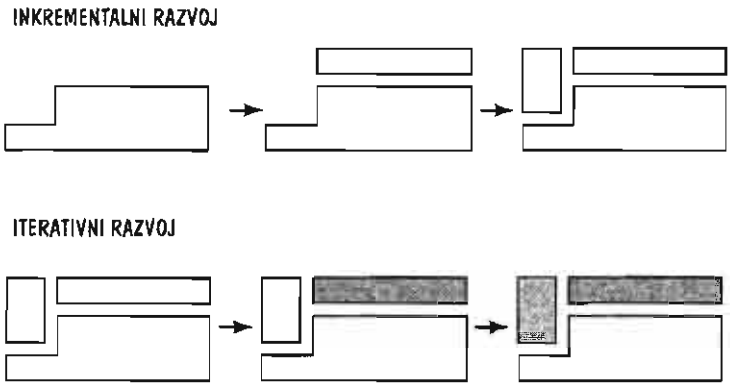


SLIKA 2.7 Transformacioni model

Postoje mnogi načini da projektni tim odluči kako da organizuje razvoj pojedinih verzija. Dva najpopularnija pristupa su inkrementalni razvoj i iterativni razvoj. Kod **inkrementalnog razvoja**, sistem, onako kako je specifikovan u specifikaciji zahteva, se deli na podsisteme, prema funkcijama. Verzije se definišu na samom početku kao mali, funkcionalni podsistemi, a zatim se svakoj novoj verziji priključuju nove funkcije. Na vrhu slike 2.9 prikazano je kako se inkrementalnim razvojem svako novo izdanje sistema polako dograđuje, sve do potpune funkcionalnosti.



SLIKA 2.8 Model faznog razvoja



SLIKA 2.9 Inkrementalni i iterativni modeli

Međutim, prilikom **iterativnog razvoja** isporučuje se odmah, na samom početku, potpun sistem, a zatim se menjaju funkcije svakog podsistema, u svakoj novoj verziji. Donji deo slike 2.9 prikazuje tri verzije u iterativnom razvoju.

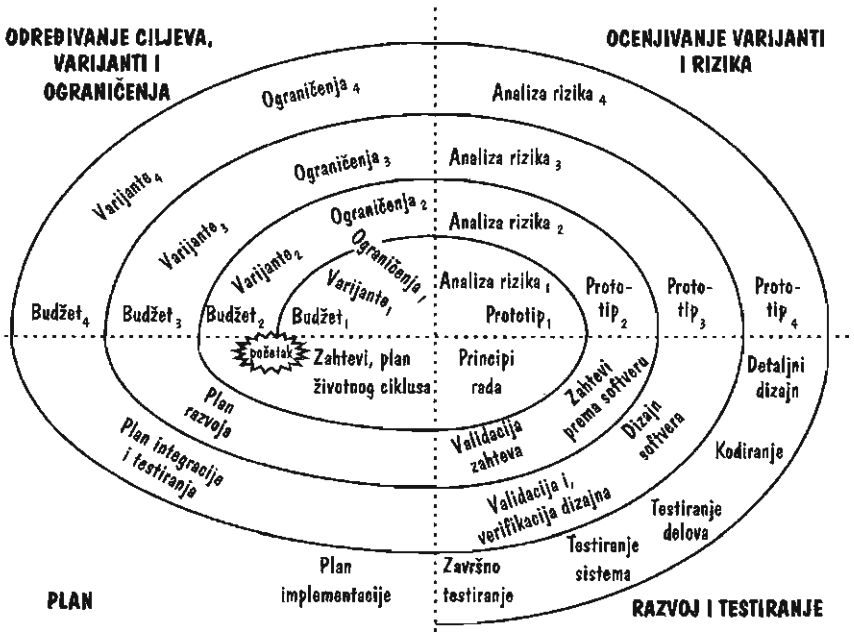
Da bismo razumeli razliku između inkrementalnog i iterativnog razvoja, razmotrićemo programski paket za obradu teksta. Pretpostavimo da paket treba da isporuči tri vrste funkcionalnosti: unošenje teksta, organizovanje (tj. isecanje i umetanje), kao i formatiranje teksta (upotreba različitih veličina i stilova). Za gradnju sistema inkrementalnim razvojnim putem, u verziji 1 možemo da obezbedimo samo funkcije za unos teksta, zatim u verziji 2 i funkcije za unos i za organizaciju, a na kraju u verziji 3 funkcije za unos teksta, organizaciju i formatiranje. Međutim, koristeći iterativni razvoj, mogli bismo da obezbedimo primitivne oblike sva tri tipa funkcionalnosti već u verziji 1. Na primer, možemo da unesemo tekst, a zatim da isecamo i umećemo, pri čemu je isecanje i umetanje nezgodno i sporo. Tako da u sledećoj iteraciji, tj. verziji 2, imamo istu funkcionalnost, ali poboljšan kvalitet, tako da je isecanje i umetanje lako i brzo. Svaka verzija unapređuje na neki način prethodnu. Ovakvi oblici faznog razvoja poželjni su iz sledećih razloga:

1. Obuka može da započne sa prvom verzijom, čak i ako neke funkcije nedostaju. Proces obuke omogućava projektnom timu da posmatra kako se izvršavaju neke funkcije, i time sugeriše poboljšanja u kasnijim verzijama. Na isti način, projektni tim je vrlo otvoren za pitanja korisnika.
2. Na tržište može da se izađe vrlo rano, ako se radi o funkcionalnostima koje nikada ranije nisu bile nuđene.
3. Česte verzije omogućavaju projektnom timu da brzo i globalno otkloni nepredviđene probleme, čim se o tome dobije izveštaj od produkcionog sistema.
4. Projektni tim može da se usredsredi na različite oblasti usavršavanja u različitim verzijama. Na primer, u jednoj verziji može da izmeni korisnički interfejs sistema sa komandno orijentisanog na zadavanje komandi mišem, na osnovu stručnog znanja tima za korisnički interfejs. Sledeća verzija može da se usredsredi na poboljšanje performansi sistema.

Spiralni model

Boehm (1988) je posmatrao razvoj softvera u svetlu prisutnih rizika, sugerišući da spiralni model može da kombinuje aktivnosti razvoja sa upravljanjem rizicima, da bi ih bilo manje i lakše se kontrolisali. Spiralni model, prikazan na slici 2.10, na neki način je sličan modelu iterativnog razvoja datom na slici 2.9. Započinjući za zahtevima i početnim planom razvoja (uključujući budžet, ograničenja i alternative u pogledu osoblja, dizajna i razvojnog okruženja), ovaj proces ubacuje korak procenjivanja rizika i prototipske alternative, pre nego što se proizvede dokument „principi rada", sa ciljem opisivanja funkcionisanja sistema na visokom nivou apstrakcije. Iz tog dokumenta, definiše se i nadgleda skup zahteva da bi se potvrdili kompletnost i doslednost zahteva. Stoga je princip rada proizvod prve iteracije, dok su zahtevi glavni proizvod iteracije druge po redu. U trećoj iteraciji razvoj sistema produkuje dizajn, dok četvrta iteracija omogućava testiranje.

U svakoj iteraciji, analiza rizika određuje različite varijante sa aspekta zahteva i ograničenja, dok se izradom prototipova verifikuje izvodljivost ili poželjnost, pre izbora odgovarajuće varijante. Nakon identifikacije rizika, rukovodioci projekta moraju da odluče kako da ih eliminišu ili minimizuju. Na primer, projektanti možda nisu sigurni da li će korisnicima više da se dopada jedan korisnički interfejs ili drugi. Da bi se izbegao rizik izbora interfejsa koji bi onemogućio produktivnu upotrebu novog sistema, projektanti mogu da izrade prototip oba interfejsa i testiraju ih da bi zaključili koji je poželjniji, ili čak da oba interfejsa uključe u projekat kako bi korisnici mogli da, nakon prijavljivanja, odaberu interfejs. Ograničenja kao što su budžet i vreme isporuke pomažu kod izbora strategije upravljanja rizicima. Upravljanje rizicima biće detaljno razmotreno u poglavlju 3.



SLIKA 2.10 Spiralni model

Agilne metode

Mnogi modeli razvojnog procesa koji su predloženi i korišćeni u periodu od 1970-ih do 1990-ih pokušavali su da nametnu neki oblik discipline vezane za način na koji se softver osmišljava, dokumentuje, razvija i testira. Krajem 1990-ih, grupa projektanata koji su se odupirali toj disciplini, formulisala je sopstvene principe, pokušavajući da naglasi uloge koje fleksibilnost može da igra u spretnom brzom razvoju softvera. Oni su kodifikovali svoja razmišljanja u „agilnom manifestu", koji se usredsređuje na četiri principa alternativnog načina razmišljanja o procesu razvoja softvera (Agile Alliance 2001):

- Za njih pojedinci i interakcije imaju veću vrednost nego procesi i alati. Ova filozofija podrazumeva da je projektnom timu potrebno obezbediti neophodne resurse i imati poverenje u to da će oni svoje poslove uraditi dobro. Timovi se samoorganizuju i komuniciraju direktno licem u lice, umesto posredstvom dokumentacije.
- Oni više vole da ulože vreme u izradu softvera koji radi, nego u izradu sveobuhvatne dokumentacije. To jest, primarno merilo uspeha je stepen do koga softver radi ispravno.
- Oni se usredsređuju na zajednički rad sa naručiocem, umesto na proces ugovaranja i pregovaranja, čime se naručilac uključuje u ključne aspekte razvojnog procesa.
- Oni se usredsređuju da odgovore na promene, umesto na planiranje i praćenje plana, jer veruju da je nemoguće da se svi zahtevi predvide na početku razvojnog procesa.

Opšti cilj agilnog razvoja je zadovoljavanje naručioca uz oslonac na „pravovremenui stalnu isporuku korisnog softvera" (prema Agile Alliance, 2001). Mnoge poslovne potrebe naručioca se menjaju tokom vremena, što je ne samo posledica novih zahteva, već i potrebe da se odgovori promenama na tržištu. Na primer, nakon što se softver projektuje i izgradi, konkurencija može da objavi novi proizvod koji zahteva izmene u planiranoj funkcionalnosti softvera, ili vladina agencija ili telo za standardizaciju može da nametne nove propise ili standarde koji utiču na projekat ili ograničenja vezana za softver. Postoji mišljenje da uvođenjem fleksibilnosti u proces razvoja agilne metode pružaju mogućnost da naručioci dodaju ili menjaju zahteve u kasnim fazama ciklusa razvoja.

Ima mnogo primera agilnih procesa u savremenoj literaturi. Svaki je zasnovan na skupu principa koji implementiraju principe iz agilnog manifesta. Primeri obuhvataju:

- **Ekstremno programiranje, XP** (*eXtreme Programming*), detaljno opisano u nastavku, predstavlja skup tehnika za nivelisanje kreativnosti projektnog tima uz minimizovanje prekomernog administriranja.
- **Crystal** predstavlja skup pristupa zasnovanih na činjenici da svaki projekat zahteva različit skup politika, dogovora i metodologija. Cockburn (2002), tvorac Crystala, veruje da ljudi imaju glavni uticaj na kvalitet softvera, pa se time kvalitet projekata i procesa poboljšava sa poboljšanjem kvaliteta ljudi koji u njemu učestvuju. Produktivnost se uvećava kroz bolju komunikaciju i česte isporuke, jer je potreba za međuproizvodima manja.
- **Scrum**, stvoren 1994. od strane grupacije Object Technology, a kasnije su ga komercijalizovali Schwaber i Bedle (2002). Ovaj model se zasniva na iterativnom razvoju, gde se svaka 30-dnevna iteracija naziva „sprint" a služi za implementiranje zaostalih

zahteva najvišeg prioriteta. Više samoorganizovanih i autonomnih timova paralelno implementira inkremente proizvoda. Koordinacija se vrši na kratkim dnevnim statusnim sastancima pod nazivom *scrum* (termin iz ragbija).

- **Adaptivni razvoj softvera** (Adaptive Software Development, **ASD**) se zasniva na šest osnovnih principa. Postoji misija koja igra ulogu smernice, postavlja cilj, ali ne propisuje kako se do njega dolazi. Svojstva predstavljaju suštinsku vrednost za naručioca, tako da se projekat organizuje oko izgradnje komponenti koje ih implementiraju. Iteracije su bitne, tako da su ponovljene radnje jednako važne kao i početne. Izmene su dobrodošle, tako da se izmene ne posmatraju kao korekcije grešaka već kao prilagođavanje realnostima u procesu razvoja softvera. Utvrđeno vreme isporuke primorava projektni tim da redukuje obim esencijalnih zahteva u sklopu svake radne verzije. Pri tome je rizik nešto što se prihvata tako da se projektni tim prvo hvata u koštac sa najtežim problemom.

Često se izraz „ekstremno programiranje" koristi za opisivanje opštijeg koncepta agilnih metoda. U stvari, XP je poseban oblik agilnog procesa, zasnovan na principima koji odražavaju opštija načela agilnog manifesta. Pobornici XP-a naglašavaju četiri karakteristike agilnosti: komunikaciju, jednostavnost, odvažnost i povratnu spregu. *Komunikacija* obuhvata neprestanu razmenu informacije između naručioca i projektnog tima. *Jednostavnost* ohrabruje projektni tim da odabere najjednostavniji dizajn ili implementaciju, koji odgovaraju potrebama naručilaca. *Odvažnost* stvaraoci XP-a opisuju kao posvećenost blagovremenim i čestim isporukama funkcija. *Povratne sprege* se ugrađuju u različite aktivnosti tokom procesa razvoja. Na primer, programeri rade zajedno da bi jedni drugima dali povratne informacije o najboljem načinu za implementaciju projektnih rešenja, a naručioci rade sa razvojnim timom u cilju planiranja.

Te karakteristike ugrađene su u takozvanih *dvanaest faktora XP-a*:

- *Igra planiranja*: Ovaj XP-ov aspekt se sastoji u tome da naručilac, koji se nalazi na licu mesta, definiše šta se podrazumeva pod pojmom „vrednost" na takav način da se svaki zahtev može oceniti prema vrednosti koju njegova implementacija donosi sistemu. Korisnici pišu scenarije o tome kako bi sistem trebalo da radi, a projektni tim procenjuje neophodne resurse za realizaciju tih scenarija. Scenariji opisuju učesnike i akcije koji se sprovode, slično „slučajevima korišćenja" koje ćemo detaljnije opisati u poglavljima 4 i 6. Svaki scenario odnosi se na jedan zahtev: dve ili tri rečenice su sve što je potrebno da se razvojnom timu dovoljno detaljno objasne vrednosti zahteva, radi specifikovanja test-slučajeva i za procenu resursa potrebnih za implementiranje zahteva. Nakon pisanja scenarija, potencijalni korisnici dodeljuju prioritete zahtevima, razdvajaju ih i spajaju, sve dok se ne postigne konsenzus o tome šta je stvarno potrebno, šta može da se testira, kao i šta može da se uradi sa raspoloživim resursima. Planeri onda generišu mapu svake verzije, dokumentujući šta ta verzija sadrži i kada će biti isporučena.
- *Male verzije*: Sistem je projektovan tako da funkcionalnost može da se isporuči što je pre moguće. Funkcije su dekomponovane na male delove, sa ciljem da se neka funkcionalnost isporuči u ranoj fazi, a zatim poboljšava ili proširuje u kasnijim verzijama. Male verzije zahtevaju fazni pristup razvoju, sa inkrementalnim ili iterativnim ciklusima.

- *Metafora*: Projektni tim se usaglašava oko zajedničke vizije načina na koji će sistem raditi. Kao podršku zajedničkoj viziji tim bira zajedničku nomenklaturu i sporazumeva se oko zajedničkog načina tretiranja ključnih pitanja.
- *Jednostavan dizajn*: Jednostavnost dizajna se održava tako što se tretiraju jedino aktuelne potrebe. Ovaj pristup se zasniva na gledištu da predviđanje potencijalnih budućih potreba dovodi do nepotrebnih funkcija. Ako je neki deo sistema vrlo složen, tim može da realizuje tzv. „klin", tj. brzu i usmerenu implementaciju, koja pomaže u procesu donošenja odluke o tome kako će se dalji rad nastaviti.
- *Pisanje testova pre kodiranja*: Da bi se osiguralo da potrebe naručioca budu glavni pokretač razvoja, prvo se pišu scenariji testova, kao način kojim se naručilac primorava da specifikuje zahteve koji se, po izradi softvera, mogu testirati i verifikovati. U XP-u se koriste dve vrste testova. Prva vrsta su funkcionalni testovi koje specifikuje naručilac, a koje sprovode projektni tim i korisnici. Druga vrsta su testovi delova koje piše i izvršava razvojni tim. U XP-u su funkcionalni testovi automatizovani, a u idealnom slučaju oni se izvršavaju svakog dana. Funkcionalni testovi se smatraju delom specifikacije sistema. Testovi delova se pišu i pre i posle kodiranja, radi verifikovanja da svaki modul implementacije radi prema očekivanjima. Funkcionalni testovi i testovi delova opisani su detaljnije u poglavlju 8.
- *Refaktorisanje (prerađivanje koda)*: Tokom izgradnje sistema, postoji verovatnoća da dođe do promene zahteva. Kako je glavna karakteristika XP filozofije dizajn jedino aktuelnih zahteva, pojava novih zahteva često primorava projektni tim da preispitaju postojeća projektna rešenja. **Refaktorisanje** (prerađivanje koda) se odnosi na preispitivanje zahteva i dizajna, i njihovo preformulisanje u skladu sa novim i postojećim potrebama. Nekada se refaktorisanje bavi prestrukturisanjem koda, bez izmena spoljašnjeg ponašanja sistema. Refaktorisanje se vrši u malim koracima, uz testiranje delova i programiranje u paru, sa jednostavnošću kao idejom vodiljom. Poteškoće pri refaktorisanju biće razmotrene u poglavlju 5.
- *Programiranje u paru*: Kao što je napomenuto u poglavlju 1, postoje dva oprečna stanovišta o tome šta je softversko inženjerstvo, umetnost ili nauka. Programiranje u paru se bavi „umetničkom" stranom razvoja softvera, priznajući činjenicu da je metafora majstor-šegrt korisna u fazi u kojoj novopečeni programeri uče kako da razviju instinkt majstora. Koristeći jednu tastaturu, dva programera koji rade u paru razvijaju sistem na osnovu specifikacija i dizajna. Jedna osoba je odgovorna za kodiranje, no rad u paru poseduje fleksibilnost, jer jedan programer može da ima više partnera u istom danu. U poglavlju 7 ćemo uporediti programiranje u paru sa tradicionalnim pristupima, gde pojedinci rade odvojeno do okončanja jediničnog testiranja njihovih modula.
- *Kolektivna svojina*: U XP-u svaki učesnik u razvoju može da izmeni bilo koji deo sistema, dok je on u fazi razvoja. U poglavlju 11 bavićemo se poteškoćama upravljanja promenama, uključujući greške koje se generišu kada dve osobe pokušavaju da istovremeno izmene isti modul.
- *Neprekidna integracija*: Brza isporuka funkcionalnosti znači da isporuka sistema koji funkcionišu može da se obeća naručiocu u roku od jednog dana ili čak jednog sata. Naglasak je na malim inkrementima ili poboljšanjima, umesto na velikim skokovima od jedne revizije do druge.

- *Održiv korak*: Naglasak na ljudima u XP-u uzima u obzir i činjenicu da umor može proizvesti greške. Tako pobornici XP-a sugerišu da je cilj 40 radnih časova nedeljno. Ako projektni tim treba da utroši ogromno vreme da bi sustigao rokove, to je siguran znak da su rokovi nerealni ili da postoji nedostatak resursa neophodnih za njihovo poštovanje.
- *Naručilac raspoloživ na terenu*: U idealnom slučaju, naručilac bi trebalo da bude raspoloživ na terenu, tj. trebalo bi da radi sa razvojnim timom na definisanju zahteva i obezbeđivanju povratnih informacija o tome kako treba da se vrši testiranje.
- *Standardi kodiranja*: Mnogi posmatrači vide XP i druge agilne metode kao okruženje bez ikakvih ograničenja, gde je sve moguće. Međutim, XP u stvari zastupa jasno definisanje standarda kodiranja, sa ciljem da se članovi tima osposobe da razumeju neophodne izmene u produktima rada drugih članova tima. Ti standardi podržavaju i ostali praktičan rad, kao što je testiranje i refaktorisanje. Rezultat treba da bude kôd koji izgleda kao da ga je pisala jedna osoba, konsistentan sa aspekta pristupa i izražavanja.

Ekstremno programiranje i agilni metodi relativno su novi. Pouzdani dokazi efikasnosti tih metoda su mali, ali sve brojniji. U kasnijim poglavljima vraćaćemo se na mnoge agilne metode i koncepte i njihovu empirijsku evaluaciju, uz razmatranje aktivnosti koje su povezane sa njima.

Modeli procesa predstavljeni u ovom poglavlju jesu samo neki od onih koji se koriste ili razmatraju. Kao što je napomenuto u podsetniku 2.3, potrebno je da proces razvoja shvatimo kao skup procesnih modela, a ne da se bavimo samo jednim modelom ili pogledom.

PODSETNIK 2.2 KADA JE EKSTREMNO SUVIŠE EKSTREMNO?

Kao i kod većine pristupa razvoju softvera, i agilne metode su podložne kritici. Na primer, Stephens i Rosenberg (2003) ukazuju da su mnogi praktični aspekti ekstremnog programiranja međusobno zavisni tako da izmena nekog od njih čini pristup ranjivim. Da bismo videli zašto je to tako, pretpostavićemo da se neki ljudi osećaju neugodno ako programiraju u paru. Više koordinacije i dokumentacije je potrebno za bavljenje zajedničkom vizijom nego o samostalnom radu. Slično tome, mnogi projektanti više vole da malo rade na dizajnu pre nego što počnu da pišu kôd. Scrum se bavi ovim stvarima kroz mesečne sprintove. Elssamadissy i Schalliol (2002) napominju da se u ekstremnom programiranju zahtevi izražavaju kao skup test scenarija koje softver mora uspešno da prebrodi. Taj pristup može da dovede do toga da se predstavnici kupca usredsrede na test scenarije umesto na zahteve. Kako test scenariji predstavljaju detaljne specifikacije zahteva i mogu biti zasnovani na rešenju, insistiranje na njima može da udalji kupce od osnovnih ciljeva projekta i rezultira situacijom u kojoj sistem prolazi sve testove, ali naručioci smatraju da to nije ono što su oni platili. Kao što ćemo videti u poglavlju 5, preradivanje koda može da predstavlja Ahilovu petu agilnih metoda, jer je preradivanje softvera bez degradiranja njegove arhitekture teško.

PODSETNIK 2.3 SKUPOVI PROCESNIH MODELA

Videli smo u podsetniku 2.1 da proces razvoja predstavlja aktivnost rešavanja problema, ali mali broj popularnih procesnih modela obuhvata i rešavanje problema. Curtis, Krasner i Iscoe (1988) izvodili su terensko istraživanje 17 velikih projekata, da bi odredili koje faktore rešavanja problema treba uključiti u procesni model radi lakšeg razumevanja razvoja softvera. Posebno su razmatrali faktore ponašanja i organizacione faktore koji utiču na ishod projekta. Njihovi rezultati impliciraju da je model razvoja softvera višeslojni model ponašanja, koji uključuje pet bitnih aspekata: poslovno okruženje, preduzeće, projekat, tim i pojedinca. Pogled iz ugla pojedinca obezbeđuje informacije o spoznaji i motivaciji, a iz ugla projekta i tima govori o dinamici grupe. Ambijent preduzeća i poslovno okruženje obezbeđuju informacije o ponašanju u sklopu organizacije koje može da utiče kako na produktivnosti tako i na kvalitet. Ovaj model ne zamenjuje tradicionalne procesne modele. Umesto toga, on je ortogonalan i predstavlja dopunu tradicionalnih modela informacijama o tome kako ponašanje utiče na stvaralačke i proizvodne aktivnosti.

Dok projektni tim i naručilac stiču saznanja o problemu, oni udružuju svoja znanja iz date oblasti, tehnologije i poslovanja, radi izgradnje odgovarajućeg rešenja. Posmatranjem razvoja kao skupa koordiniranih procesa, možemo uočiti efekte učenja, tehničke komunikacije, interakcije sa naručiocem, kao i efekte pregovaranja o zahtevima. Aktuelni modeli koji propisuju niz razvojnih postupaka „ne obezbeđuju pomoć pri analizi one količine novih informacija koje projektanti moraju da usvoje, kako treba da se pregovara oko neusaglašenih zahteva, kako projektni tim može da razreši konflikte u arhitekturi, i kako ti i slični faktori doprinose neodređenosti i rizicima koji su svojstveni projektima" (Curtis, Krasner i Iscoe 1988). Međutim, kada uključimo kognitivne, socijalne i organizacione procese, otkrivaju se uzroci postojanja uskih grla i neefikasnosti. To je upravo ona sposobnost opažanja koja omogućava rukovodiocima da shvate proces razvoja i upravljaju njime. Agregiranjem ponašanja kroz slojeve modela, možemo uočiti kako svaki model povećava ili umanjuje uticaj činilaca iz drugog modela.

Bez obzira na to koji se procesni model koristi, mnoge aktivnosti su zajedničke za sve. Tokom izlaganja o softverskom inženjerstvu u kasnijim poglavljima, ispitaćemo svaku razvojnu aktivnost posebno da bismo videli šta ona sve obuhvata, i da bismo videli koji to alati i tehnike doprinose efikasnosti i produktivnosti.

2.3 ALATI I TEHNIKE ZA MODELOVANJE PROCESA

Nakon donošenja odluke o tome šta procesni model treba da sadrži, na raspolaganju su nam različite tehnike i alati za modelovanje. U opisima modela iz prethodnog odeljka, do sada smo videli više pristupa modelovanju. Odgovarajuća tehnika zavisi od postavljenih ciljeva i načina rada kome se daje prednost. Izbor notacije posebno zavisi od sadržaja modela. Notacija varira od tekstualne, u kojoj se procesi iskazuju kao funkcije, preko grafičkih notacija koje prikazuju procese u formi hijerarhije pravougaonika i strelica, do notacija koje kombinuju slike, tekst i grafički prikaz sa tabelama u cilju elaboriranja funk-

cija na visokom nivou apstrakcije. Mnoge notacije za modelovanje mogu da se koriste za predstavljanje zahteva i dizajna. Neke od njih ćemo analizirati u narednim poglavljima.

U ovom poglavlju, notacija je od sekundarnog značaja za vrstu modela, te našu pažnju usmeravamo na dve osnovne kategorije modela, statičke i dinamičke. **Statički model** opisuje proces, kroz prikaz transformacije ulaza u izlaze. **Dinamički model** oponaša proces na takav način da korisnik vidi kako se međuproizvodi i finalni proizvodi transformišu u vremenu.

Statičko modelovanje: Laieva notacija

Postoji mnogo načina za statičko modelovanje procesa. Početkom 1990-ih, Lai (1991) je razvio optšu notaciju namenjenu modelovanju procesa na proizvoljnom nivou detalja. Ovaj model se gradi na paradigmi da ljudi igraju uloge, a resursi izvršavaju aktivnosti, što vodi do izrade artefakata. Procesni model prikazuje odnose između uloga, aktivnosti i artefakata, dok tabele stanja prikazuju informacije o stepenu gotovosti svakog artefakta u datom trenutku.

Elementi procesa se posmatraju u svetlu sedam tipova:

1. **Aktivnost:** Nešto što će se desiti u procesu. Ovaj element može da se dovede u vezu sa zbivanjima pre ili posle, potrebnim resursima, događajem koji inicira aktivnost, pravilima koja upravljaju njome, načinom na koji se opisuju algoritmi i stečena saznanja, i odnosom između aktivnosti i projektnog tima.
2. **Sekvenca:** Redosled aktivnosti. Sekvenca može da se opiše pomoću pokretačkog događaja (okidača), programskih konstrukcija, transformacija, uređenja ili zadovoljavanja uslova.
3. **Model procesa:** Pogled na sistem. Delovi procesa mogu da se predstave kao odvojeni modeli u cilju predviđanja ponašanja ili ispitivanja pojedinih karakteristika procesa.
4. **Resursi:** To su neophodni element, alat ili osoblje. Resursi obuhvataju opremu, vreme, kancelarijski prostor, ljude, tehniku itd. Od procesnog modela zavisi koliko je pojedinih resursa potrebno za svaku aktivnost.
5. **Kontrola:** Spoljašnji uticaj na ustanovljeni proces. Kontrole mogu biti ručne ili automatske, od strane ljudi ili mehaničke.
6. **Politika:** Vodeći princip. Ograničenja uspostavljena na visokom nivou apstrakcije procesa utiču na njegove performanse. Ona obuhvata propisani proces razvoja, alat koji mora da se koristi ili obavezni stil upravljanja.
7. **Organizacija:** Hijerarhijska struktura agenasa procesa sa fizičkim grupisanjem koje odgovara logičkom grupisanju i pratećim ulogama. Preslikavanje fizičkog grupisanja u logičko treba da bude dovoljno fleksibilno i da odražava promene u fizičkom okruženju.

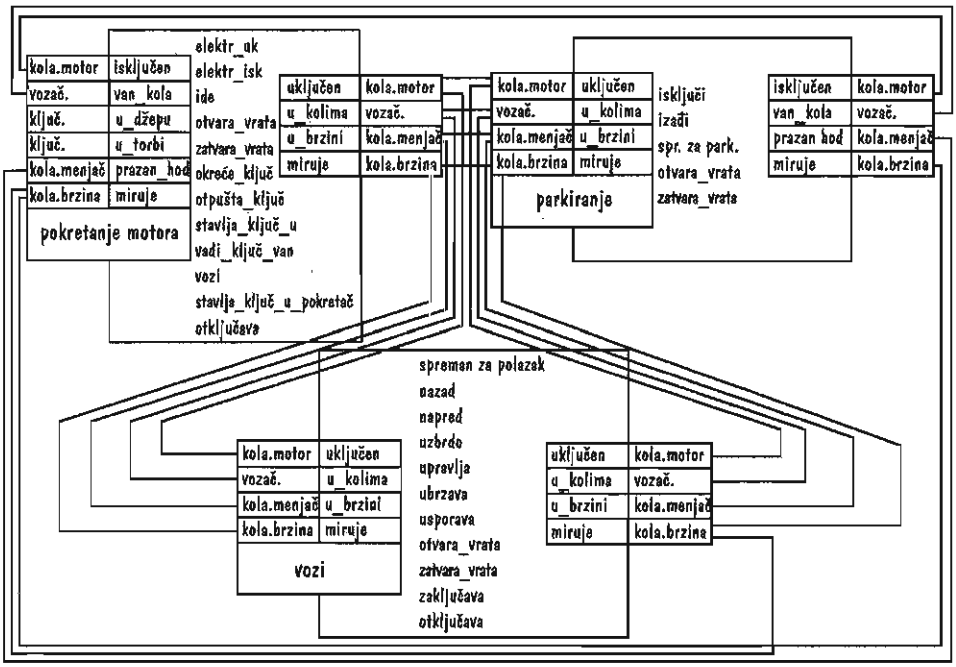
Sam opis procesa ima nekoliko nivoa apstrakcije, uključujući proces razvoja softvera koji usmerava korišćenje pojedinih resursa pri konstruisanju određenih modula, kao i generičke modele, koji podsećaju na spiralni model ili model vodopada. Notacija koju je dao Lai obuhvata više šablona, kao što je šablon za definiciju produkata (*Artifact Definition Template*) koji beleži informacije o pojedinim artefaktima.

Laiev pristup se može primeniti na modelovanje procesa razvoja softvera. Kasnije u ovom poglavlju koristićemo ga za modelovanje rizika koji prate razvoj. Međutim, da bi se pokazala njegova primena i sposobnost obuhvatanja različitih fragmenata složenih aktivnosti, primenjujemo ga na relativno jednostavan ali poznat proces upravljanja automobilom. Tabela 2.1 sadrži opis ključnog resursa u ovom procesu.

Drugi šabloni definišu relacije, stanja procesa, operacije, analizu, akcije i uloge. Grafički dijagrami prikazuju odnose između elemenata i obuhvataju glavne i sporedne odnose. Na primer, slika 2.11 ilustruje proces pokretanja automobila. Pravougaonik „pokretanje" predstavlja ulazne uslove dok pravougaonik „parkiranje" predstavlja izlazni uslov. Leva kolona unutar pravougaonika sa uslovima navodi artefakte, a desna predstavlja njihovo stanje.

Tabela 2.1 Obrazac za definisanje artefakta „KOLA"

Ime	*Kola*	
Rezime	*Ovo je artefakt koji predstavlja klasu kola.*	
Tip kompleksnosti	*Kompozitni*	
Tip podataka	*(kola c, korisnički definisan)*	
Lista stanja artefakta		
parkirana kola	*((stanje(kola.motor) = isključen) (stanje(kola.menjač) = prazan hod) (stanje(kola.brzina) = miruje))*	*Kola nisu u pokretu, motor ne radi.*
pokrenut motor	*((stanje(kola.motor) = uključen) (stanje(kola.brava) = ključ_unutra) (stanje(vozač-kola(kola.) = u_kolima) (stanje(kola.menjač) = u_položaju_vožnja) (stanje(kola.brzina) = miruje))*	*Kola nisu u pokretu, ali motor radi.*
kola u pokretu	*((stanje(kola.motor) = uključen) ((stanje(kola.brava) = ključ_unutra) ((stanje(vozač-kola(kola.) = upravlja) ((stanje(kola.menjač) = u brzini) ili (stanje(kola.menjač) = vožnja unazad)) ((stanje(kola.brzina) = stoji) ili (stanje(kola.brzina) = mala) ili (stanje(kola.brzina) = srednja) ili (stanje(kola.brzina) = velika))*	*Kola se pomeraju napred ili nazad.*
Lista podartefakta		
	vrata	*Četvoro vrata na kolima*
	motor	*Motor u kolima*
	brava	*Brava za paljenje motora*
	menjač	*Menjač u kolima*
	brzina	*Brzina kola*
Lista relacija		
kola-ključ	*Ovo je relacija između kola i ključa.*	
kola-vozač	*Ovo je relacija između kola i vozača.*	

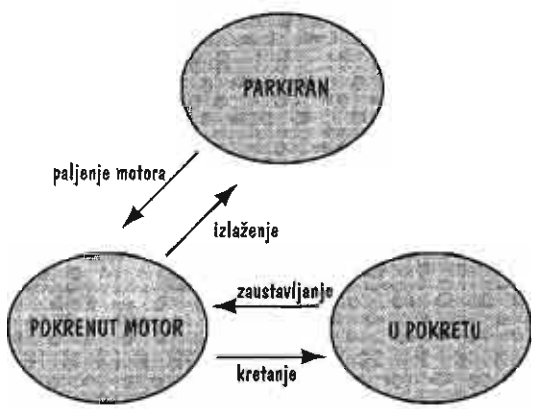


SLIKA 2.11 Proces pokretanja kola (Lai 1991)

Dijagrami prelaza služe kao dopuna procesnog modela prikazujući kako se stanja odnose jedno prema drugom. Na primer, slika 2.12 ilustruje prelaze iz stanja u stanje za kola.

Laieva notacija predstavlja dobar primer kako je moguće iskoristiti više struktura i strategija za prikupljanje većine informacija o procesu razvoja softvera. Međutim, takođe je korisna za organizovanje i opisivanje procesnih podataka vezanih za korisničke zahteve, kao što je pokazano na primeru automobila.

SLIKA 2.12 Dijagram prelaza iz stanja u stanje za automobil (Lai 1991)

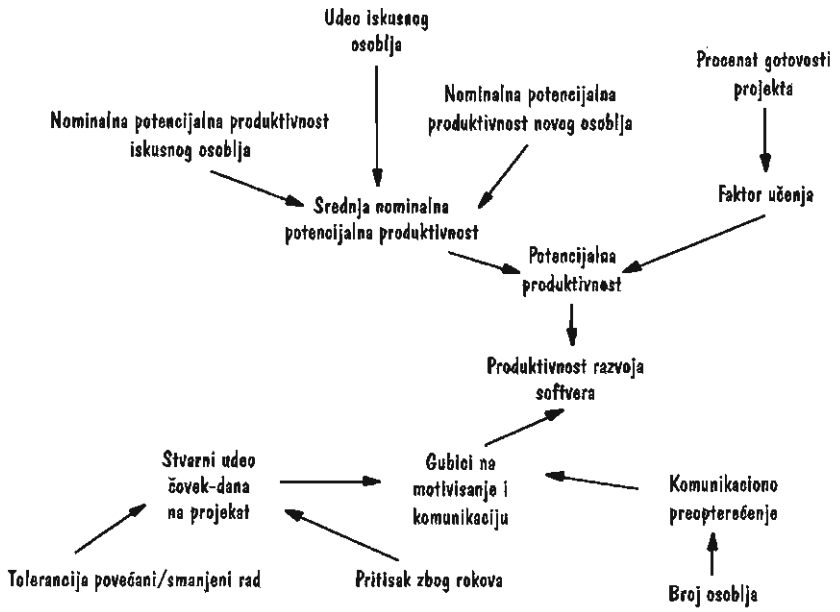


Dinamičko modelovanje: Dinamika sistema

Poželjna osobina procesnog modela je sposobnost da igra ulogu samog procesa, tako da možemo da pratimo šta se događa sa resursima i artefaktima pri obavljanju aktivnosti. Drugim rečima, želimo da opišemo procesni model, a zatim da posmatramo softver kako nam prikazuje protok resursa kroz aktivnosti, sve do izlaza. Ovaj dinamički pogled na proces omogućava da proces simuliramo i izmenimo pre nego što se resursi zaista utroše. Na primer, možemo da koristimo dinamički model kao pomoć pri donošenju odluke o broju ljudi neophodnih za proces testiranja, ili o trenutku otpočinjanja procesa testiranja da bismo ga okončali u skladu sa postavljenim rokovima. Slično tome, možemo da uključimo ili isključimo aktivnosti da bismo videli njihov uticaj na potreban rad i vreme izrade. Na primer, možemo da dodamo aktivnost pregledanja koda, da pravimo pretpostavke o broju nedostataka koje ćemo otkriti tokom pregleda i utvrdimo da li pregled značajno skraćuje vreme testiranja.

Ima nekoliko načina za izgradnju dinamičkog procesnog modela. Dinamika sistema je pristup koji je uveo Forrester 1950. godine, koji se koristi za simulaciju raznih procesa, uključujući ekološke, ekonomske i političke sisteme (Forrester 1991). Abdel-Hamid i Madnick su primenili dinamiku sistema na razvoj softvera, omogućavajući rukovodiocima projekta da „istestiraju" svoje odluke vezane za proces, pre nego što ih proslede projektnom timu (Abdel-Hamid 1989; Abdel-Hamid i Madnick 1991).

Da bismo videli kako funkcioniše dinamika sistema, pogledajmo najpre kako proces razvijanja softvera utiče na produktivnost. Možemo da izgradimo deskriptivan model različitih aktivnosti u koje se tiču vremena koje troši razvojni tim, a zatim da posmatramo kako promene u modelu uvećavaju ili umanjuju vreme potrebno za dizajn, pisanje i testiranje koda. Prvo, moramo da odredimo koji činioci utiču na ukupnu produktivnost.



SLIKA 2.13 Model činilaca koji utiču na produktivnosti (Abdel-Hamid 1996).

Slika 2.13 prikazuje Abdel-Hamidovo shvatanje tih činilaca. Strelice pokazuju kako izmene jednog činioca utiču na drugi. Na primer, ako se udeo iskusnog osoblja uveća sa jedne četvrtine na jednu polovinu ukupnog broja ljudi angažovanih na projektu, onda očekujemo da će se povećati i prosečna produktivnost. Slično tome, što je više osoblja (u svetlu upravo njihovog broja), više vremena odlazi na komunikaciju između članova projektnog tima (komunikacioni operativni trošak).

Slika pokazuje da na srednju nominalnu produktivnost utiču tri stvari: produktivnost iskusnog osoblja, udeo iskusnog osoblja i produktivnost novog osoblja. U isto vreme, novo osoblje mora da uči o projektu. Što je veći stepen gotovosti projekta, to novi članovi tima moraju više da uče pre nego što postanu produktivni.

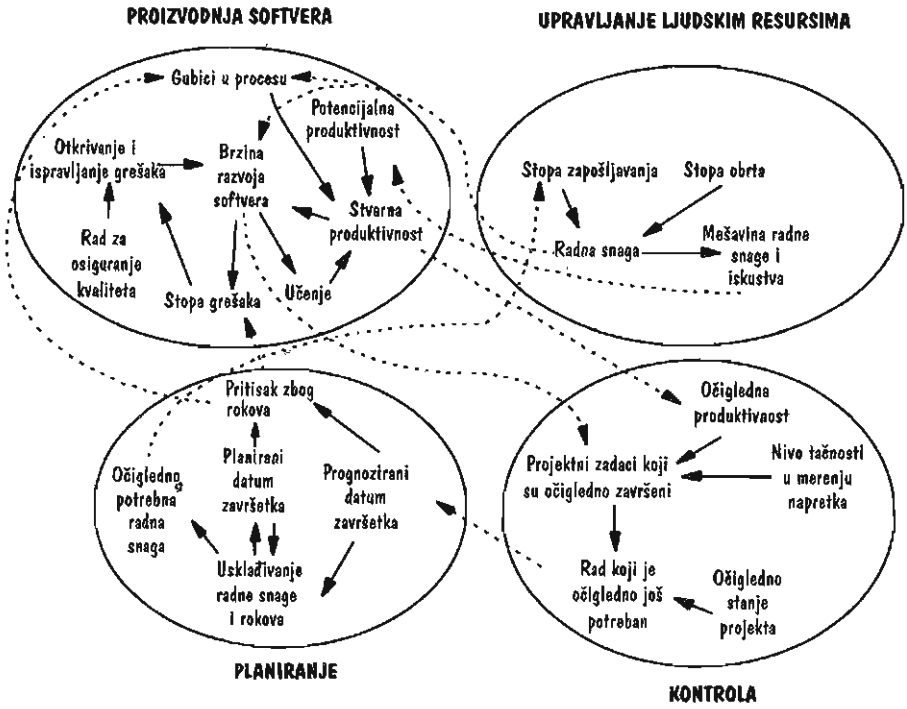
Druga pitanja utiču na opštu produktivnost procesa razvoja. Prvo, moramo da razmotrimo udeo dnevne posvećenosti radu na projektu za svakog učesnika u razvoju. Pritisci zbog rokova utiču na taj udeo, kao i tolerancije učesnika u projektu na radno opterećenje. Brojnost osoblja takođe utiče na produktivnost, ali što je više osoblja, verovatnije je da će biti potrebno dodatno vreme za komunikaciju i razmenu informacija između članova tima. Komunikacija i motivacija, kombinovani sa potencijalnom produktivnošću, predstavljenom na gornjoj polovini slike 2.13, sugerišu odnos prema opštoj produktivnosti procesa razvijanja softvera.

Stoga prvi korak u primeni dinamike sistema jeste identifikovanje navedenih odnosa kombinovanjem empirijskih dokaza, istraživačkih izveštaja i intuicije. Sledeći korak je kvantitativno određivanje odnosa. Kvantifikacija može da obuhvati direktne odnose, kao što je pomenuti odnos između brojnosti osoblja i komunikacije. Znamo da ako je n ljudi dodeljeno projektu, onda postoji $n(n - 1)/2$ potencijalnih parova ljudi koji moraju da komuniciraju i koordiniraju aktivnosti. Za neke odnose, posebno one koji uključuju resurse koji se menjaju u vremenu, moramo uključiti funkciju raspodele koja opisuje povećanje i umanjenje resursa. Na primer, retko se dešava da svi učesnici na projektu započnu rad od prvog dana. Prvo počinju analitičari sistema, koderi se pridružuju kada su dokumentovani ključni zahtevi i delovi dizajna. Pomenuta funkcija raspodele opisuje rast i pad resursa (ili njihove fluktuacije, kao što su raspoloživost pred praznike ili letnje odmore).

Dinamički model sistema može biti obiman i složen. Na primer, Abdel-Hamidov model razvoja softvera sadrži više od 100 uzročnih veza. Slika 2.4 prikazuje pregled odnosa koje je on definisao. On je definisao četiri glavne oblasti koje utiču na produktivnost: izrada softvera, upravljanje ljudskim resursima, planiranje i kontrola. Proizvodnja uključuje pitanja osiguranja kvaliteta, učenja i brzinu razvoja. Ljudski resursi se tiču zapošljavanja, promene ljudstva i iskustva. Planiranje se tiče rokova i pritisaka koje oni uzrokuju, dok se kontrola bavi merenjem napretka i rada potrebnog za završetak projekta.

Broj veza može biti prilično velik, pa su stoga dinamički modeli sistema podržani softverom koji snima veze i njihove kvantitativne opise i zatim simulira celokupni proces ili neke potprocese.

Moć dinamike sistema je impresivna, ali taj metod treba da se pažljivo koristi. Rezultati simulacije zavise od kvantifikovanih odnosa, koji su često heuristički ili magloviti, i nisu zasnovani na jasnim empirijskim istraživanjima. Međutim, kao što ćemo videti u kasnijim poglavljima, baza podataka sa istorijatom informacija dobijenih na osnovu merenja raznih aspekata razvoja, može pomoći da se stekne vera u znanje o odnosima, a samim tim i u rezultate dinamičkih modela.



SLIKA 2.14 Struktura razvoja softvera (Abdel-Hamid 1996).

PODSETNIK 2.4 PROGRAMIRANJE PROCESA

Sredinom 1980-ih, Osterweil (1987) je predložio da se proces softverskog inženjerstva opiše pomoću algoritama. To jest, ako je proces dobro shvaćen, moguće je napisati program koji opisuje proces, a zatim ga izvršiti kao zamenu za sam proces. Cilj programiranja procesa je eliminisanje neodređenosti, tako što uz dovoljno razumevanje pišemo softver koji sadrži sustinu procesa i pretvaramo ga u deterministički rešenje problema.

Kad bi programiranje procesa bilo moguće, onda bi sve aktivnosti jednog procesa bile očigledne s upravljačkog aspekta, sve aktivnosti bi se mogle automatizovati i sve aktivnosti bi se lako koordinisale i trpele izmene. Otud bi procesni programi mogli činiti osnovu okruženja za automatizovanu proizvodnju softvera.

Međutim, Curtis, Krasner, Shen i Iscoe (1987) ukazuju da Osterweilova analogija sa računarskim programiranjem ne zadovoljava varijabilnost svojstvenu razvojnim procesima. Kada se piše računarski program, programer pretpostavlja da implementaciono okruženje radi ispravno. To znači da su operativni sistem, sistem za upravljanje bazom podataka i hardver pouzdani i ispravni, tako da gotovo da nema izmena u odgovoru računara na izdatu naredbu. Međutim, kada procesni program izda naredbu članu projektnog tima, mogu se javiti značajne razlike u načinu obavljanja zadatka i samom ishodu. Kao što ćemo videti u poglavlju 3, razlike u sposobnostima, iskustvu, radnim navikama, razumevanju potreba kupaca i niz drugih činilaca mogu dramatično da uvećaju varijabilnost.

Curtis i njegove kolege sugerišu da programiranje procesa treba ograničiti samo na situacije sa minimalnom varijabilnošću. Osim toga, oni ukazuju da Osterweilovi primeri pružaju informaciju samo o redosledu zadataka. Procesni program ne pomaže da se rukovodioci upozore na probleme koji predstavljaju prepreke. „Ne izgleda da se koordinacija mreže kreativnih intelektualnih zadataka znatno poboljšava procesnim programiranjem kakvo se danas primenjuje, budući da je za koordinaciju najvažnije da se za sve agense koji su u međusobnoj interakciji obezbedi zajednička mentalna slika načina na koji sistem funkcioniše." (Curtis i dr. 1987).

2.4 PRAKTIČNO MODELOVANJE PROCESA

Modelovanje procesa bilo je dugo u središtu istraživanja softverskog inženjerstva. Ali koliko je ono praktično? Više istraživača izveštava da, ako se pravilno koristi, modelovanje procesa je od velike koristi za razumevanja procesa i otkrivanja nedoslednosti. Na primer, Barghouti, Rosenblum, Belanger i Alliegro (1995) sproveli su dve studije slučaja radi određivanja izvodljivosti, upotrebljivosti i ograničenja kod upotrebe procesnih modela u velikim organizacijama. U ovom odeljku, istražićemo šta su oni uradili i šta su pronašli.

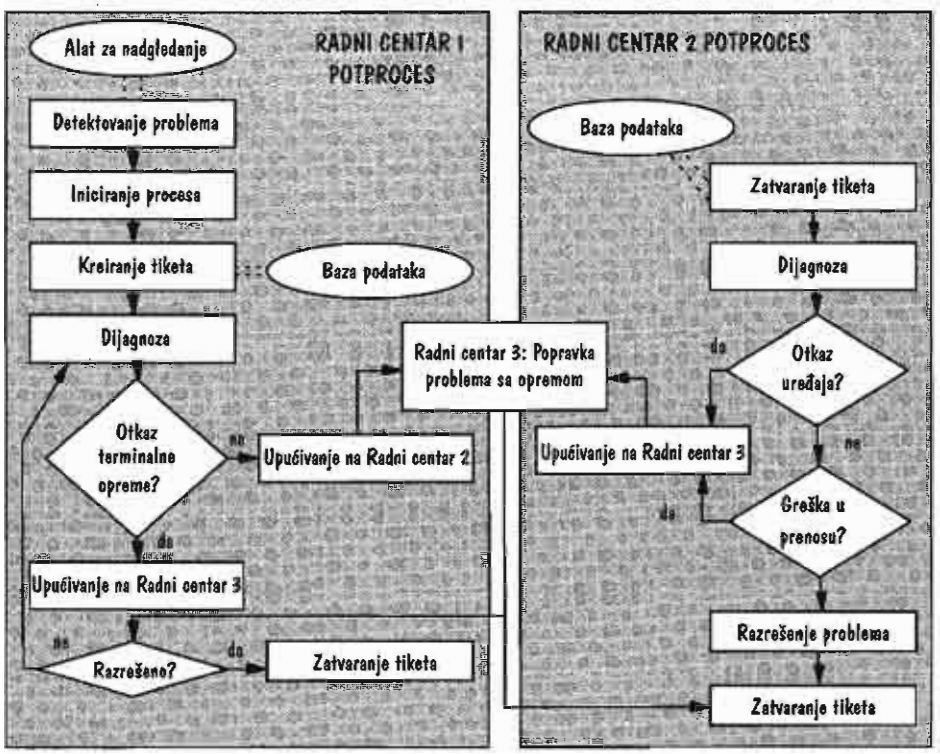
Marvelove studije slučajeva

U obe studije, istraživači su koristili MSL (*Marvel Specification Language*) za definisanje procesa, a zatim su generisali okruženje za izvođenje specifikacije na jeziku Marvel (Kaiser, Feiler i Popovich 1988; Barghouti i Kaiser 1991). MSL koristi tri glavne konstrukcije - klase, pravila i obvojnice alata - radi kreiranja trodelnog opisa procesa:

1. specifikacije ponašanja procesa zasnovane na pravilima;
2. objektno orijentisane definicije informacionih procesa u sklopu modela;
3. skupa obvojnica za sprezanje Marvela sa spoljašnjim softverskim alatima koji izvršavaju procese.

Prva studija se odnosila na mrežu za obradu poziva u kompaniji AT&T, koja se sastoji od mreže za prenos telefonskih razgovora i odvojene signalne mreže odgovorne za usmeravanje poziva i balansiranje opterećenja mreže. Za opisivanje procesa odgovornog za otkrivanje, servisiranje i razrešavanje problema sa signalnom mrežom (*Signaling Fault Resolution*) korišten je Marvel. Radni centar 1 nadgledao je mrežu, otkrivao nedostatke i prosleđivao nedostatak jednom od druga dva radna centra. Radni centar 2 razrešavao je softverske ili ljudske greške koje su zahtevale detaljnu analizu, dok se Radni centar 3 bavio hardverskim otkazima. Slika 2.15 opisuje taj proces. Dvostruke isprekidane linije pokazuju koja aktivnost koristi alat ili bazu podataka, koji su predstavljeni elipsama. Pravougaonik je zadatak ili aktivnost, a romb predstavlja odluku. Strelice pokazuju smer toka kontrole. Kao što možemo da vidimo, slika pruža opšti pregled, ali nije dovoljno detaljna da bi se mogli predstaviti suštinskielementi procesa.

Zato se svaki entitet i radni centar modeluje pomoću MSL-a. Slika 2.16 prikazuje kako je to urađeno. Gornja polovina slike definiše klasu TIKET, gde taj „tiket" predstavlja cedulju sa problemom (ili izveštaj o problemu) koja se piše svaki put kada se dogodi otkaz u sistemu. Kao što ćemo videti u poglavljima koja se bave testiranjem, cedulje sa problemom se koriste za praćenje problema od prvog pojavljivanja pa do razrešavanja. Čitava mreža predstavljena je sa 22 MSL klase. Uključene su sve informacije koje proces stvara ili zahteva.



SLIKA 2.15 Proces razrešavanja signalnih grešaka (Barghouti i dr. 1995).

SLIKA 2.16 Primer komandi Marvela (Barghouti i dr. 1995).

```
TICKET:: superclass ENTITY                                    Definicija
  status: (initial, open, referred_out, referral_done,        klase za
                  closed, fixed) = initial;                   tikete sa
  diagnostics: (terminal, non_terminal, none) = none;         problemima
  level          : integer;
  description: text;
  referred_to: link WORKCENTER;
  referrals      : set_of link TICKET;
  process        : link PROC_INST;
end

diagnose [?t: TICKET]:
  (exists PROC_INST ?p suchthat (linkto [?t.process ?p]))
  :                                                           Pravilo za
  (and (?t.status = open)(?t.diagnostics = none))             dijagnostiku
  {TICKET_UTIL diagnose ?t.Name}                              tiketa
  (and (?t.diagnostics = terminal)
          (?p.last_task = diagnose)
          (?p.next_task = refer_to_WC3));
  (and (?t.diagnostics = non_terminal)
          (?p.last_task = diagnose)
          (?p.next_task = refer_to_WC2));
```

Dalje, model se bavi ponašanjem procesa za signalizaciju grešaka (*Signaling Fault Resolution*). Donja polovina slike 2.16 je MSL pravilo koje otprilike odgovara pravougaoniku „Dijagnoza" sa slike 2.15. Prema tome, MSL opisuje pravilo za dijagnostifikovanje otvorenih problema i to pravilo se aktivira za svaki otvoreni tiket. Nakon završetka procesnog modela, bilo je neophodno 21 MSL pravilo za opis sistema.

Druga studija slučaja bavi se procesom održavanja softvera, u konkretnom slučaju komutacionog softvera 5ESS kompanije AT&T. Za razliku od prve studije, gde je cilj bio poboljšanje procesa, druga studija se bavila jedino dokumentovanjem koraka u procesu i interakcijama, koje su predstavljane pomoću MSL-a. Model je sadržavao 25 klasa i 26 pravila.

Kod svakog modela korišćeni su MSL opisi procesa u cilju generisanja „okruženja za odigravanje procesa", što je rezultiralo popunjavanjem baze podataka instancama klasa informacionog modela. Nakon toga su istraživači simulirali više scenarija za proveru da li se modeli ponašaju prema očekivanjima. Tokom simulacije, oni su sakupljali podatke o vremenu i iskorišćenosti resursa, pružajući osnove za analiziranje verovatnih performansi procesa. Scenario je izvršavan više puta sa izmenjenim pravilima pa se, na osnovu poređenja dobijenih vremena, došlo do značajnog poboljšanja procesa, uglavnom bez dodatnog ulaganja u resurse.

Vežbe modelovanja i simulacije bile su korisne za blagovremenu identifikaciju i rešavanje problema. Na primer, definicija procesa održavanja softvera otkrila je tri tipa problema kod postojećeg postupka dokumentovanja: nedostaju ulazi i izlazi zadataka, višeznačnost ulaznih i izlaznih kriterijuma, kao i neefikasno definisanje procesa. Simulacija modela za signalizaciju grešaka pokazala je neefikasnost u odvojenim opisima radnih centara.

Barghouti i njegove kolege ističu značaj deljenja problema procesnog modelovanja na dve oblasti: modelovanje informacija i modelovanje ponašanja. Odvakvo odvajanje daje sažetije i jasnije modele. Curtis i njegove kolege su takođe ukazali i na to da se intenzivne računarske aktivnosti znatno lakše modeluju nego ljudske.

Poželjne odlike alata i tehnika za modelovanje procesa

Postoje mnogi alati i tehnike za modelovanje procesa, a istraživači nastavljaju da rade na iznalaženju alata koji najbolje odgovaraju u datim situacijama. Međutim, postoje neke karakteristike koje su korisne, bez obzira na upotrebljenu tehniku. Curtis, Kellner i Over (1992) identifikovali su pet kategorija poželjnih osobina:

1. *Ljudima olakšava razumevanja i međusobnu komunikaciju.* Tehnika treba da predstavi proces u obliku koji većina naručilaca i projektanata razume, da podstiče razgovore u vezi sa procesom i da olakša usaglašavanje o njegovim oblicima i unapređenju. Tehnika treba da obuhvati dovoljno informacija da bi omogućila da jedna ili više osoba zaista i sprovede proces. Takođe, model ili alat treba da formiraju osnovu za proces obuke.
2. *Podrška poboljšanju procesa.* Tehnika treba da identifikuje suštinske komponente procesa razvoja ili održavanja. Ona treba da omogući višekratnu upotrebljivost procesa ili potprocesa u budućim projektima, poređenje alternativa i procenu uticaja izmena pre nego što se proces zaista sprovede u praksi. Slično tome, tehnika modelovanja treba da pomaže prilikom: izbora alata i drugih tehnika neophodnih

procesu; podsticanja obuke u sklopu organizacije i stvaranja podrške stalnoj evoluciji procesa.

3. *Podrška upravljanju procesom.* Tehnika treba da omogući da proces bude prilagođen projektu. Projektanti i naručioci treba da izvode zaključke o atributima izrade ili evolucije softvera. Tehnika treba takođe da podržava planiranje i prognoziranje, nadgledanje i upravljanje procesom, kao i merenje njegovih ključnih karakteristika.
4. *Obezbeđuje automatsko vođenje tokom izvršavanja procesa.* Tehnika treba da definiše celokupno ili delimično okruženje za razvoj softvera, obezbedi smernice i sugestije, i očuva višekratno upotrebljive reprezentacije procesa za kasniju upotrebu.
5. *Podrška automatskom izvršavanju procesa.* Tehnika treba da automatizuje proces u celosti ili delimično, da podržava kooperativni rad, snima relevantne rezultate merenja, kao i da nameće pravila za očuvanje integriteta procesa.

Ove karakteristike mogu da igraju ulogu korisnih smernica na osnovu kojih se biraju tehnike modelovanja procesa za određeni razvojni projekat. Tačka 4 je posebno važna ako konkretna organizacija pokušava da standardizuje svoj proces. Alati mogu da pomognu razvojnom timu oko toga šta treba sledeće da rade, a takođe i obezbeđuju pristup i kontrolne tačke radi osiguranja da proizvod zadovoljava određene standarde pre nego što se preduzmu sledeći koraci. Na primer, alat može da proveri skup komponenti koda i oceni njihovu veličinu i strukturu. Ako veličina ili struktura premašuju definisane granice, razvojni tim se obaveštava pre nego što započne proces testiranja, što može rezultirati revizijom ili preprojektovanjem nekih komponenti.

2.5 PRIMER IZ INFORMACIONOG SISTEMA

Razmotrićemo koji proces razvoja da koristimo za podršku našeg primera iz informacionog sistema, tj. za oglasni program televizije Picadilly. Podsetimo se da ima više ograničenja u vezi sa tim koja vrsta oglasnog prostora može da se proda i u koje vreme, a da propisi mogu da se promene odlukama koje donose Agencija za standardizaciju oglašavanja i druga nadzorna tela. Prema tome, mi želimo da izgradimo softverski sistem koji je lak za održavanje i unošenje izmena. Postoji čak i mogućnost da se ograničenja izmene tokom izgradnje sistema.

Možda je model vodopada suviše krut za naš sistem, jer dozvoljava vrlo malo fleksibilnosti dokle god se faza analize zahteva potpuno ne dovrši. U dizajnu korisničkog interfejska korisno je napraviti prototip, tako da želimo da uključimo neku vrstu prototipa u naš sistem. Većina neodređenosti međutim leži u propisima za oglašavanje i poslovnim ograničenjima. Želimo da se oslonimo na procesni model će biti moguće upotrebiti jednom ili više puta, kako sistem evoluira. Varijacija spiralnog modela može biti dobar kandidat za izradu sistema televizije Picadilly, jer podržava vraćanje na ranije pretpostavke, analizu rizika i izradu prototipa za različite karakteristike sistema. Preispitivanje alternativa, prikazano u gornjem levom kvadrantu spirale, pomaže da ugradimo fleksibilnost kako u zahteve tako i u dizajn.

Boehmov model spirale je na visokom nivou apstrakcije i sa nedovoljno detalja za usmeravanje akcija analitičara, projektanata, programera i onih koji rade na testiranju. Međutim, postoji više tehnika i alata za predstavljanje procesnog modela, sa mnogo finijim nivoom detalja. Izbor tehnike ili alata delom zavisi od ličnih sklonosti i iskustva, a

delom od pogodnosti za predstavljanje razmatranog procesa. Videćemo sada kako Laieva notacija može da se koristi za predstavljanje dela procesa u razvoju sistema televizije Picadilly.

Spiralni model želimo da koristimo kao pomoć prilikom upravljanja rizicima, pa zato u naš procesni model moramo da uključimo definiciju „rizika". Rizik je takođe artefakt koji moramo da opišemo na takav način da ga možemo izmeniti i pratiti u svakoj iteraciji spirale. Svaki potencijalni problem nosi svoj rizik, a o riziku razmišljamo zavisno od dva aspekta: verovatnoće javljanja i stepena ozbiljnosti. **Verovatnoća** je mera izvesnosti da će se određeni problem dogoditi, dok je **ozbiljnost** meru posledica koje problem može da ima po sistem. Na primer, pretpostavimo da razmatramo problem nedovoljne obuke u razvojnom metodu koji se koristi za gradnju sistema televizije Picadilly. Možemo da se odlučimo za objektno orijentisani pristup, ali možda članovi razvojnog tima dodeljenog projektu imaju malo ili nikakvo iskustva u objektno orijentisanom pristupu. Verovatnoća za javljanje ovog problema je možda mala, jer se svi novi radnici šalju na intenzivni, četvoronedeljni kurs objektno orijentisanog razvoja. S druge strane, ako se problem ipak javi, on će ozbiljno uticati na sposobnost razvojnog tima da završi softver u skladu sa definisanim rokovima. Prema tome, verovatnoća događaja je mala, ali je njegova ozbiljnost velika.

Takve riskantne situacije možemo da predstavimo u Laievoj tabeli artefakta 2.2. Rizik predstavlja artefakt gde su verovatnoća i ozbiljnost predstavljeni podartefaktima. Radi jednostavnosti, odabrali smo samo dva stanja svakog podartefakta: nisku i visoku verovatnoću, kao i malu i veliku ozbiljnost. U stvari, svaki od podartefakata može da ima širok opseg stanja (kao što su mala, vrlo mala, donekle mala, srednja, nešto visoka, vrlo visoka, ekstremno visoka), koja dovode do više različitih stanja samog artefakta.

Tabela 2.2 Obrazac definicije ~~artefakta~~ „Rizik"

Ime	*Rizik (Problem X)*	
Rezime procesa	*Ovo je artefakt koji predstavlja rizik da će se problem X dogoditi i prouzrokovati negativan efekat na neki aspekt procesa razvoja.*	
Tip kompleksnosti	*Kompozitan*	
Tip podataka	*(rizik_s, korisnički_definisan)*	
Lista stanja artefakta		
niska	((*stanje*(*verovatnoća.x*) = *niska*) (*stanje*(*ozbiljnost.x*) = *mala*))	Verovatnoća problema je niska, ozbiljnost problema je velika.
visoko-srednja	((*stanje*(*verovatnoća.x*) = *niska*) (*stanje*(*ozbiljnost.x*) = *velika*))	Verovatnoća problema je niska, ozbiljnost problema je mala.
nisko-srednja	((*stanje*(*verovatnoća.x*) = *visoka*) (*stanje*(*ozbiljnost.x*) = *mala*))	Verovatnoća problema je niska, ozbiljnost problema je mala.
visoka	((*stanje*(*verovatnoća.x*) = *visoka*) (*stanje*(*ozbiljnost.x*) = *velika*))	Verovatnoća problema je niska, ozbiljnost problema je mala.
Lista podartefakta		
	verovatnoća.x	*Verovatnoća da će se dogoditi problem X*
	ozbiljnost.x	*Ozbiljnost posledica ako se problem X dogodi tokom procesa.*

Na isti način možemo da definišemo druge aspekte procesa razvoja i koristimo dijagrame za ilustrovanje aktivnosti i njihovih međusobnih veza. Modelovanje procesa na ovaj način ima mnogo prednosti, što se ne svodi samo na građenje zajedničkog shvatanja o tome šta su zahtevi razvoja. Ako korisnici, naručioci i projektanti učestvuju u definisanju i objašnjavanju procesa razvoja televizije Picadilly, svaki od njih imaće očekivanja u vezi sa tim koje su aktivnosti uključene, šta one proizvode, kao i kada svaki proizvod može da se očekuje. Kombinacija spiralnog modela i tabele rizika može da se koristi za periodično ocenjivanje rizika. Sa svakim obrtom spirale, verovatnoća i ozbiljnost svakog rizika može da se ažurira u skladu sa novim stanjem. Kada su rizici neprihvatljivo visoki, model procesa može da se revidira radi uključivanja tehnika redukovanja i ublaženja rizika, kao što ćemo videti u poglavlju 3.

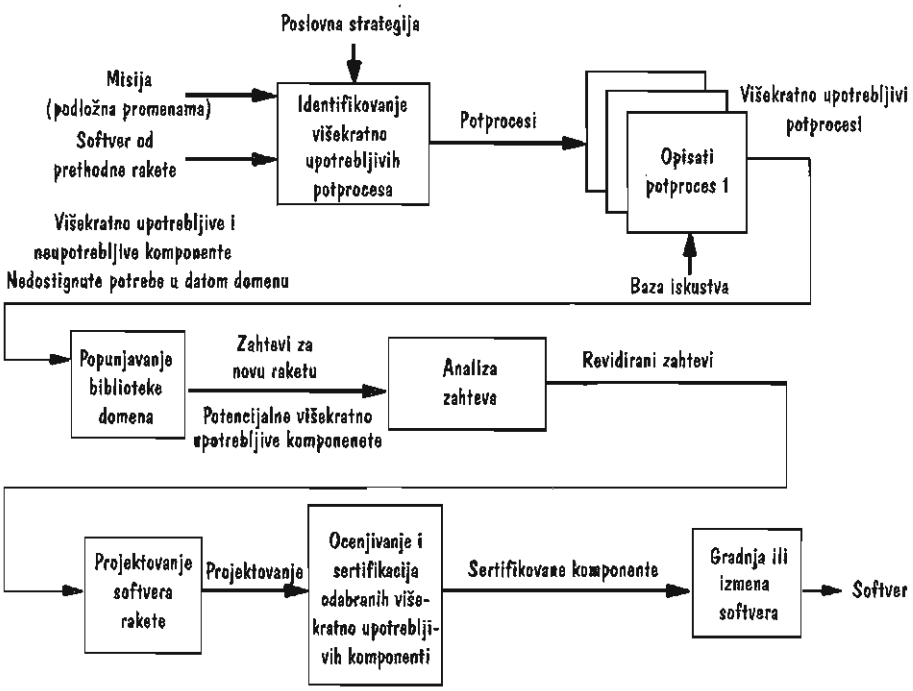
2.6 PRIMER IZ PRAKSE

Softver korišćen u projektu Ariane-5 bio je višekratno upotrebljiv. Višekratna upotrebljivost ovog softvera imala je za cilj smanjenje rizika, povećanje produktivnosti i podizanje kvaliteta. Otuda svaki procesni model za razvoj novog softvera za raketu Ariane treba da obuhvati aktivnosti vezane za višekratnu upotrebljivost. Procesni model posebno mora da obuhvati aktivnosti u vezi sa proverom kvaliteta višekratno upotrebljivih komponenti, i pri tome se s posebnom predostrožnošću treba uveriti da višekratno upotrebljivi softver radi dobro u kontekstu novodizajniranog sistema.

Takav procesni model mogao bi da izgleda slično ovom uprošćenom modelu na slici 2.17. Pravougaonici u prikazu modela predstavljaju aktivnosti. Strelice koje ulaze u pravougaonik sa leve strane su resursi, a one koje ga napuštaju sa desne strane su izlazi. Strelice koje dolaze odozgo su kontrole ili ograničenja, kao što su rokovi, budžeti ili standardi. Strelice koje dolaze odozdo su mehanizmi koji asistiraju u izvođenju aktivnosti, kao što su alati, baze podataka ili tehnike.

Proces preuzimanja softvera od Ariane-4 započinje sa misijom softvera, naime sa kontrolnim funkcijama za novu raketu, a tu su takođe i softver preuzet iz prethodnih raketa, ranije neispunjeni zahtevi, kao i softverske komponente raspoložive iz drugih izvora (kao što su kupovni softver ili višekratna softverska skladišta iz drugih projekata). Na osnovu poslovne strategije graditelja svemirskih brodova, razvojni tim može da identifikuje višekratne potprocese, da ih opiše (verovatno uz objašnjenja koja se oslanjaju na prethodna iskustva) i da ih uvrsti u biblioteku za dalje razmatranje od strane analitičara zahteva. Višekratno upotrebljivi procesi često obuhvataju višekratno upotrebljive komponente (zahteve, delove projekta ili koda, čak i testne slučajeve, opise procesa, kao i druge dokumente i artefakte).

Dalje, analitičari zahteva ispituju zahteve vezane za novo telo rakete, kao i višekratno upotrebljive komponente kojima raspolaže biblioteka. Analitičari proizvode prerađen skup zahteva, koji se sastoji od mešavine novih i preuzetih zahteva. Zatim, projektanti koriste te zahteve za projektovanje softvera. Kada je projekat gotov, projektanti ocenjuju sve komponente preuzete iz prethodnog projekta, u cilju potvrđivanja da su ispravne i konsistentne sa novim delovima projekta i opštom namenom sistema, kao što je opisano u zahtevima. Na kraju, potvrđene komponente se koriste za gradnju ili izmenu softvera i proizvodnju krajnjeg sistema. Kao što ćemo videti u kasnijim poglavljima, opisani postupak je možda mogao da spreči uništenje rakete Ariane-5.



SLIKA 2.17 Procesni model višekratne upotrebljivosti u slučaju novog softvera za raketu.

2.7 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju videli smo da proces razvoja softvera obuhvata aktivnosti, resurse i proizvode. Model procesa je koristan za upravljanje ponašanjem kada se radi u grupi. Detaljni procesni modeli pokazuju kako da koordinišemo proce i kako da sarađujemo sa kolegama tokom projektovanja i izgradnje sistema. Takođe smo videli da procesni modeli obuhvataju organizacione, funkcionalne, bihejviorističke i druge aspekte, tako da možemo da se usredsredimo na posebna gledišta razvojnog procesa, radi boljeg razumevanja radnji koje sprovodimo ili načina na koji njima upravljamo.

2.8 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Model procesa takođe ima jasne prednosti i za razvojni tim. Dobar model prikazuje svakom članu tima kada se i gde odvijaju pojedine aktivnosti i ko ih izvodi, tako da je raspodela zaduženja sasvim jasna. Pored toga, rukovodilac projekta može da koristi procesne alate za izvođenje procesa, za simulaciju aktivnosti i praćenje resursa, radi određivanja najbolje kombinacije ljudi i aktivnosti u cilju ispunjenja rokova i budžeta datog projekta. Ta simulacija se radi pre nego što su resursi zaista utvrđeni, čime se štede vreme i novac, jer nema potrebe za vraćanjem unazad i ispravljanjem grešaka. Iterativni i inkrementalni razvoj mogu da se uključe u procesni model, tako da tim može da dolazi do saznanja na osnovu prototipa ili da reaguje na promenu zahteva i pri tome očuva postavljene krajnje rokove.

2.9 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Modelovanje procesa predstavlja plodno tlo za istraživače koji se bave softverskim inženjerstvom. Mnogi učesnici u razvoju softvera osećaju da dobri procesi mogu biti garancija kvaliteta proizvoda razvoja. Prema tome, postoji više oblasti kojima se bave istraživači:

- *Notacije procesa*: Kako opisati proces na način koji je razumljiv onima koji moraju da ga izvedu.
- *Modeli procesa*: Kako opisati proces, pomoću odgovarajućeg skupa aktivnosti, resursa, proizvoda i alata.
- *Alati za podršku modelovanja procesa*: Kako modelovati ili simulirati proces tako da možemo oceniti raspoloživost resursa, njihovu upotrebu i performanse.
- *Merenje i ocenjivanje procesa*: Kako odrediti koje aktivnosti, resursi, potprocesi i tipovi modela su najbolji za izradu proizvoda visokog kvaliteta u zadatom vremenu ili okruženju.

Mnogi od ovih napora su koordinisani sa istraživanjima poboljšanja procesa, oblasti koju ćemo istražiti u poglavlju 13.

2.10 SEMINARSKI RAD

Na početku ste procesa razvoja sistema „Aranžer pozajmica" (*Loan Arranger*) za organizaciju FCO (*Financial Consolidation Organization*). Još uvek ne postoji sveobuhvatni skup sistemskih zahteva. Postoji samo pregled funkcionalnosti sistema, kao i predstava kako će sistem biti korišćen za podršku poslovanja FCO-a. Mnogi od termina datih u pregledu verovatno nam nisu bliski, tako da se od naručioca tražilo da pripremi definicije korišćene terminologije. Odgovarajuća terminologija data je u tabeli 2.3.

Pružena informacija razjašnjava neke koncepte, ali još uvek ste daleko od dobrog skupa zahteva. Ipak, možete da donesete neke preliminarne odluke o tome kako razvoj treba da se nastavi. Proučite procese predstavljene u ovom poglavlju i odredite koji je od njih najpogodniji za razvoj *Aranžera pozajmica*. Za svaki proces, napravite listu njegovih prednosti i mana, s obzirom na *Aranžera pozajmica*.

2.11 KLJUČNE REFERENCE

Kao rezultat radne grupe u okviru Petog međunarodnog radnog skupa o softverskim procesima, kojoj je predsedavao Kellner, formulisan je standardni problem prilikom ocenjivanja i poređenja nekih popularnih tehnika modelovanja procesa. Problem je dovoljno složeno formulisan da može da posluži kao test da li tehnika uključuje svaku od sledećih stavki:

- višestruke nivoe apstrakcije;
- kontrolu toka, redosled, kao i ograničenja u redosledu;
- tačke odluke;
- iteracije i povratnu spregu sa prethodnim koracima;
- kreativnost korisnika;

Tabela 2.3 Definicije termina Loan Arrangera

Zajmoprimac: Zajmoprimac je primalac novca od zajmodavca. Zajmoprimci mogu zajedno da prime pozajmicu, tj. svaka pozajmica može da ima više zajmoprimaca. Svaki zajmoprimac ima svoje ime i jedinstveni identifikacioni broj.

Rizik zajmoprimca: Faktor rizika za svakog zajmoprimca zasnovan je na njegovoj kreditnoj istoriji. Zajmoprimcu bez neizmirenih pozajmica dodeljuje se nominalni faktor rizika od 50. Faktor rizika se smanjuje kako zajmoprimac otplaćuje dug na vreme, ali se uvećava kada zajmoprimac sa zakašnjenjem otplaćuje dug ili ne izvršava novčane obaveze po osnovu date pozajmice. Rizik zajmoprimca izračunava se pomoću sledeće formule:

Rizik = 50 - [10 × (broj godina redovnog vraćanja duga po pozajmici)]
+ [20 × (broj godina zakasnelog vraćanja duga po pozajmici)]
+ [30 × (broj godina neizvršene obaveze vraćanja duga po pozajmici)]

Na primer, zajmoprimac može da ima tri pozajmice. Prva pozajmica uzeta je pre dve godine, i sve uplate bile su učinjene na vreme. Ta pozajmica je redovno otplaćivana i to je trajalo dve godine. Druga i treća pozajmica su četiri, odnosno pet godina stare, a svaka od njih je dobro otplaćivana sve doskora. Prema tome, svaka od dve, sa zakašnjenjem otplaćivane pozajmice, bila je u tom stanju u trajanju od samo jedne godine. Prema tome, rizik je:

50 - [10 × 2] + [20 × (1 + 1)] + [30 × 0] = 70

Maksimalna vrednost rizika je 100, a minimalna 1.

Paket pozajmica: Paket pozajmica je skup zajmova koji se prodaju investitoru kao jedinstvena celina. Svaki paket ima sledeća svojstva: ukupnu vrednost zajmova u paketu, zatim vremenski period aktivnosti pozajmica iz paketa (tj. koji zajmoprimci još uvek otplaćuju pozajmicu), zatim procena rizika uključenog u kupovinu paketa i na kraju profit koji se očekuje kada se svi zajmoprimci otplate svoje zajmove.

Rizik po paketu: Rizik po paketu je ponderisana srednja vrednost rizika pojedinačnih pozajmica u paketu, gde je rizik svake pozajmice (videti *rizik pozajmice* u nastavku) ponderisan prema vrednosti te pozajmice. Za izračunavanje ponderisane srednje vrednosti za n pozajmica, pretpostavimo da za svaku pozajmicu Pi preostala glavnica iznosi Gi i rizik je Ri. Ponderisana srednja vrednost tada je:

$$\frac{\sum_{i=1}^{n} G_i R_i}{\sum_{i=1}^{n} G_i}$$

Diskonto: Diskonto je cena po kojoj je FCO voljan da proda investitoru. Diskonto se računa prema sledećoj formuli:

Diskonto = (preostala glavnica) × [(kamatna stopa) × (0,2 + (0,005 × (101 - (rizik pozajmice))))]

Tip kamatne stope: Kamatna stopa za pozajmicu je ili fiksna ili promenljiva. Pozajmica sa fiksnom kamatom (FM) ima istu kamatnu stopu sve do završetka hipotekarnog kredita. Pozajmica sa promenljivom stopom (ARM) ima stopu koja se menja svake godine, prema vladinom indeksu koji obezbeđuje američko ministarstvo finansija.

Investitor: Investitor je osoba ili organizacija koja je zainteresovana za kupovinu uvezanih pozajmica od FCO-a.

Investiciona molba: Investitor podnosi investicionu molbu, u kojoj određuje maksimalni stepen rizika sa kojim će biti napravljena investicija, minimalni iznos profita koji se zahteva od paketa, kao i maksimalni vremenski period u kojem pozajmice iz paketa moraju biti otplaćene.

Zajmodavac: Zajmodavac je institucija koja daje pozajmice zajmoprimcima. Zajmodavac može da ima nula, jednu ili više pozajmica.

Informacija o zajmodavcu: Informacija o zajmodavcu je predstavljena opisnim podacima koji su uvezeni spolja. Informacija o zajmodavcu ne može da se promeni ili obriše. Informacije o svakom zajmodavcu su sledeće: ime zajmodavca (institucija), osoba ili institucija za kontakt na strani zajmodavca, telefonski broj za kontakt, kao i jedinstveni identifikacioni broj zajmodavca. Kada se dodata u sistem, element zajmodavac može da se modifikuje ali ne i da se obriše.

nastavlja se

Tabela 2.3 *(nastavak)*

Kreditna institucija: Sinonim za zajmodavca. Videti *zajmodavac*.

Pozajmica: Pozajmica je skup informacija i ona opisuje zajam za kuću i informaciju za identifikovanje zajmoprimca koja se čuva uz pozajmicu. Za svaku pozajmicu daju se sledeće informacije: iznos pozajmice, kamatna stopa, tip kamatne stope (promenljiva ili fiksna), datum obračuna (datum kada je zajmoprimac zaista pozajmio novac od zajmodavca), rok (izražen brojem godina), zajmoprimac, zajmodavac, tip pozajmice (ogromna ili obična), kao i nekretnina (adresa nekretnine). Svaka pozajmica mora da ima tačno jednog zajmodavca i tačno jednog zajmoprimca. Pored toga, svaka pozajmica se identifikuje na osnovu rizika i statusa pozajmice.

Analitičar pozajmica: Analitičar pozajmica je radnik FCO-a koji je obučen da koristi sistem *Aranžera pozajmica* za upravljanje pozajmicama i za paketiranje pozajmica. Analitičarima pozajmica je bliska terminologija pozajmica, ali oni možda nemaju pri ruci sve relevantne informacije pomoću kojih ocenjuju jednu pozajmicu ili paket pozajmica.

Rizik pozajmice: Svaka pozajmica ima svoj nivo rizika, izražen celim brojem od 1 do 100. Vrednost 1 predstavlja pozajmicu sa najnižim rizikom, tj. nije verovatno da će zajmoprimac kasniti sa uplatama, ili da neće uopšte izvršavati svoje obaveze.

Status pozajmice: Pozajmica ima jednu od tri statusne odrednice: uredna, kašnjenje i neizvršavanje. Pozajmica je u uredna ako je zajmoprimac izmirio sve dospele uplate do datog trenutka. Pozajmica je u stanju kašnjenja ako je zajmoprimčeva poslednja uplata izvršena, ali posle datuma dospeća. Pozajmica je u stanju neizvršavanja ako poslednja dospela uplata zajmoprimca nije primljena u roku od 10 dana od dana dospeća.

Tip pozajmice: Pozajmica je ili veliki hipotekarni kredit, kada vrednost nekretnine prelazi 275 000 dolara, ili je to regularni hipotekarni kredit, kada je vrednost imovine 275 000 dolara ili manje.

Portfelj: Zbirka pozajmica koje je kupio FCO i koje mogu da udu u paket pozajmica. Skladište koga održava *Aranžera pozajmica* sadrži informacije o svim pozajmicama u portfelju.

- upravljanje objektima i informacijama, kao i tok kroz proces;
- strukturu objekata, atribute i međusobne odnose;
- organizacionu odgovornost za određene zadatke;
- mehanizme fizičke komunikacije za transfer informacija;
- merenje procesa;
- vremenske aspekte (apsolutne i relativne);
- zadatke koje izvršavaju ljudi;
- profesionalno rasuđivanje ili slobodu odlučivanja;
- vezu sa narativnim objašnjenjima;
- zadatke koje pokreće ili izvršava alat;
- ograničenja i dodeljivanje resursa, određivanje vremenskog rasporeda;
- modifikaciju i poboljšanje procesa;
- višestruke nivoe agregacije i paralelizma.

Osamnaest različitih tehnika modelovanja procesa bilo je primenjeno na zajednički problem, gde je kod svake tehnike pronađen različit stepen zadovoljenja. Rezultati su dati u referenci Kellner i Rombach (1990).

Curtis, Kellner i Over (1992) predstavljaju sveobuhvatan pregled tehnika i alata za modelovanje procesa. Članak takođe sumira osnovne tipove i jezičke konstrukcije i daje primere pristupa modelovanju procesa koji koriste te jezičke tipove.

Krasner i dr. (1992) opisuju pouke koje su izvukli kada su implementirali sistem za modelovanje softverskog procesa u komercijalnom okruženju.

Više veb lokacija sadrži informacije o modelovanju procesa:

- Institut SEI (*U.S. Software Engineering Institute*) nastavlja da istražuje modelovanje procesa u cilju unapređenja procesa. Spisak njihovih tehničkih izveštaja i aktivnosti može da se pronađe na lokaciji http://www.sei.cmu.edu.
- Bilten ESPI *Exchange* fondacije ESPI (*European Software Process Improvement Foundation*), izveštava o globalnim aktivnostima vezanim za procese. Veb lokacija fondacije ESPI je http://www.espi.co.uk.
- Evropska zajednica je duže vremena sponzorisala istraživanja u oblasti modelovanja procesa i jezika za modelovanje procesa. Sažeci istraživačkih projekata koji su u toku nalaze se na lokaciji http://www.cabernet.esprit.ec.org/esp-syn/index.html.
- Tehnički savet organizacije IEEE o softverskom inženjerstvu (*IEEE Technical Council on Software Engineering*) izdaje bilten o softverskim procesima, na lokaciji http://www.se.cs.mcgill.ca.

Više informacija o notaciji koju je dao Lai i o alatima za podršku može se dobiti od International Software Process Constellation, Inc. iz Restona, Virdžinija. Upiti se šalju na adresu 74160.1442@compuserve.com.

Centar za softversko inženjerstvo Univerziteta u Južnoj Kaliforniji razvio je alat za pomoć prilikom izbora procesnog modela pogodnog za potrebe i ograničenja datog projekta. Može da se preuzme sa ftp lokacije ftp:/usc.edu-pub-soft_engineering-demos-pmsa.zip, a više informacija može da se pronađe na veb lokaciji centra http://sunset.usc.edu.

Časopisi kao što su *Software Process - Improvement and Practice* sadrže članke koji se bave ulogom modelovanja procesa u okviru razvoja i održavanja softvera. Oni takođe navode najvažnije događaje važnih konferencija, kao što su International Software Process Workshop i International Conference on Software Engineering. Izdanje časopisa IEEE Software iz jula/avgusta 2000. godine bavi se raznolikošću procesa i sadrži više članaka o uspehu pristupa razvoju softvera preko tzv. zrelosti procesa.

Na raspolaganju je mnogo resursa za učenje o agilnim metodama. Manifest agilnosti (*Agile Manifesto*) postavljen je na lokaciji http://www.agilealliance.org. Knjiga koju je napisao Kent Beck (1999) je osnovna knjiga o ekstremnom programiranju, dok Alistair Cockburn (2002) opisuje familiju metodologija pod imenom Crystal. Martin Beck (1999) objašnjava refaktorisanje, što je jedan od najtežih koraka u okviru XP-a. Dve izvrsne reference o agilnim metodima su knjiga koju je napisao Robert C. Martin (2003) o agilnom razvoju softvera i knjiga koju su napisali Daniel H. Steinberg i Daniel W. Palmer (2004) o ekstremnom softverskom inženjerstvu. Sledeće dve veb lokacije pružaju dodatnu informaciju o ekstremnom programiranju: http://www.xprogramming.com i http://www.extremeprogramming.org.

2.12 VEŽBE

1. U kakvom je odnosu opis sistema sa pojmom procesnog modela? Na primer, kako odlučujete koja je granica sistema opisanog procesnim modelom?
2. Za svaki procesni model opisan u ovom poglavlju, navesti koje su prednosti i mane upotrebe modela.

3. Za svaki procesni model opisan u ovom poglavlju, navesti kako model izlazi na kraj sa značajnim izmenama u zahtevima koje se pojave kasnije u razvoju.
4. Nacrtati dijagram koji opisuje proces kupovanja avionske karte za službeni put.
5. Nacrtati Laievu tabelu artefakta za definisanje modula. Proveriti da li su uključena stanja artefakta koja prikazuju modul kada nije testiran, kada je delimično testiran i kada je kompletno testiran.
6. Koristeći notaciju po slobodnom izboru, nacrtati dijagram toka procesa razvoja softvera koji obuhvata izradu prototipova za tri različita projektna rešenja i odabira najbolje među njima.
7. Proučite karakteristike dobrih procesnih modela opisanih u odeljku 2.4. Koje karakteristike su suštinske za procese koji treba da se koriste za projekte, a gde problem i rešenje nisu dobro razumljivi?
8. U ovom poglavlju, sugerisano je da izrada softvera nije industrijski, već stvaralački proces. Komentarisati karakteristike proizvodnog procesa koji se primenjuje na razvoj softvera i objasniti koje su karakteristike razvoja softvera slične stvaralačkim naporima.
9. Da li projektantska organizacija treba da usvoji samo jedan procesni model za sav razvoj softvera? Dati argumente za to i protiv toga.
10. Pretpostavimo da ugovor sa naručiocem propisuje određeni proces razvoja softvera. Kako rad može da se nadgleda da bi se koristio baš taj proces?
11. Proanalizirajte procese koje ste upoznali u ovom poglavlju. Koji od njih pruža najviše fleksibilnosti na izmene u odgovor na izmenjene zahteva?
12. Pretpostavimo da, tokom pregovora za ugovaranje izgradnje sistema, kompanija Amalgamated zahteva da koristite određeni procesni model. Povinujući se, gradite softver pomoću propisanih aktivnosti, resursa i ograničenja. Nakon isporuke i instalacije softvera, sistem doživljava katastrofalan otkaz. Prilikom istraživanja izvora otkaza koje sprovodi kompanija Amalgamated, vi ste optuženi da niste dovoljno puta kontrolisali kod, jer biste inače našli izvor problema pre isporuke. Vaš odgovor je da se kontrola koda nije nalazila u procesnom modelu koji vam je propisan. Koja su pravna i etička pitanja ovde pokrenuta?

3 Planiranje projekta i upravljanje njime

U ovom poglavlju obradićemo:
- praćenje napretka projekta;
- osoblje i organizaciju projekta;
- uloženi rad i rokove;
- upravljanje rizikom;
- modelovanje procesa i planiranje projekta.

Kao što smo videli u prethodnim poglavljima, ciklus razvoja softvera se sastoji iz više koraka, pri čemu se neki od njih ponavljaju sve dok se sistem ne završi, a naručilac i korisnik ne budu zadovoljni. Međutim, pre nego što se potvrde sredstva za realizaciju razvoja ili održavanja softvera, korisnik obično želi da proceni koliko će projekat da košta i traje. U ovom poglavlju ispitujemo aktivnosti koje su neophodne za planiranje razvojnog projekta softvera i upravljanje njime.

3.1 PRAĆENJE NAPRETKA

Softver je koristan samo ako realizuje željenu funkciju ili pruža traženu uslugu. Prema tome, tipičan projekat započinje kada nam se obrati naručilac radi razmatranja uočene potrebe. Na primer, velika nacionalna banka može da zatraži pomoć pri izgradnji informacionog sistema koji bi omogućavao klijentima banke da pristupaju informacijama na svojim računima bez obzira na to na kojem kraju sveta se nalaze. Ili mogu da nam se obrate okeanolozi koji bi voleli da imaju sistem povezan sa njihovom opremom za nadgledanje voda, pomoću koga bi vršili statističke analize prikupljenih podataka. Obično naručilac ima više pitanja na koja treba dati odgovor:

- Da li razumete moj problem i moje potrebe?
- Da li možete da projektujete sistem koji bi rešio moj problem ili zadovoljio moje potrebe?
- Koliko će vam vremena biti potrebno da razvijete takav sistem?
- Koliko će koštati razvoj takvog sistema?

Odgovor na dva poslednja pitanja zahteva postojanje dobro osmišljenih rokova za aktivnosti na projektu. **Projektni rokovi** opisuje razvojni ciklus nekog softvera u okviru određenog projekta, sa fazama ili etapama projekta od kojih je svaka razbijena na izolovane zadatke ili aktivnosti koje je potrebno realizovati. On takođe odslikava interakcije između tih aktivnosti i procenjuje rokove i trajanje svakog zadatka ili aktivnosti. Prema tome, rokovi predstavljaju vremensku osu koja prikazuje kada započinju aktivnosti i kada se završavaju, tj. kada će odgovarajući proizvod razvojnog procesa biti na raspolaganju.

U poglavlju 1, naučili smo da sistemski pristup obuhvata i analizu i sintezu: razbijanje problema na delove, formulisanje rešenja za svaki deo i sklapanje tih delova u koherentnu celinu. Identičan pristup možemo koristiti i za definisanje rokova za aktivnosti u okviru projektu. Posao započinjemo radom sa naručiocima i potencijalnim korisnicima da bismo shvatili njihove želje i potrebe. U isto vreme proveravamo da li su oni zadovoljni našim poznavanjem njihovih potreba. Pri tome navodimo sve **međuproizvode**, tj. stavke koje naručilac očekuje da dobije tokom razvoja projekta. U međuproizvode spadaju:

- dokumenti;
- demonstracija funkcija;
- demonstracija podsistema;
- demonstracija tačnosti;
- demonstracija pouzdanosti, bezbednosti ili performansi.

Nakon toga utvrđujemo koje aktivnosti moraju da se sprovedu u cilju realizacije definisanih elemenata isporuke. Pri tome možemo koristiti neku od tehnika za modelovanje procesa, koje smo naučili u poglavlju 2, izlažući tačno šta mora da se dogodi i koje aktivnosti zavise od drugih aktivnosti, proizvoda, odnosno resursa. Određeni događaji se deklarišu kao miljokazi i služe kao pokazatelji napredovanja projekta. Na primer, nakon dokumentovanja zahteva, provere njihove doslednosti i kompletnosti i uručenja projektantima, specifikacija zahteva se može smatrati miljokazom projekta. Slično tome, miljokazi mogu biti završetak korisničkog uputstva, izvođenje datog skupa proračuna ili demonstracija sposobnosti sistema da komunicira sa drugim sistemom.

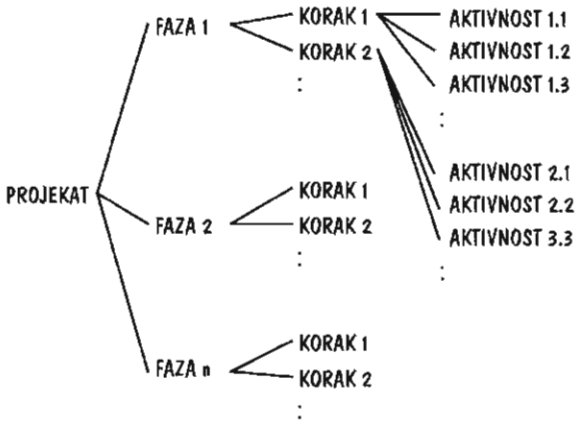
U našoj analizi projekta, moramo napraviti jasnu razliku između aktivnosti i miljokaza. **Aktivnost** je deo projekta koji se odigrava tokom nekog vremenskog perioda, dok **miljokaz** predstavlja završetak aktivnosti u određenom vremenskom trenutku. Prema tome, aktivnost ima početak i kraj, dok miljokaz predstavlja kraj neke posebno izdvojene aktivnosti. Na primer, naručilac želi da uz sistem dobije i onlajn uputstvo za rukovanje. Razvoj uputstva i pratećih programa predstavlja aktivnost koja kulminira prikazivanjem tih funkcija naručiocu, što je miljokaz.

Pažljivim ispitivanjem projekta na ovaj način, možemo da podelimo razvoj na niz faza. Svaka faza se sastoji od koraka, a svaki korak, ako je to neophodno, može biti predmet dalje podele, kao što je prikazano na slici 3.1.

Da bismo videli kako ta analiza funkcioniše, razmotrićemo faze, korake i aktivnosti date u tabeli 3.1, koje opisuju gradnju kuće. Za početak ćemo uzeti dve faze: zemljane radove i gradnju same kuće. Nakon toga ćemo svaku fazu razbiti na manje korake, kao što su: raščišćavanje i krčenje, sejanje trave i sađenje drveća i žbunova. Gde je neophodno, navedene korake možemo podeliti na aktivnosti: na primer, završetak unutrašnjosti uključuje dovršavanje unutrašnje vodovodne i električne instalacije, malterisanje, kre-

čenje, stavljanje podnih obloga, vrata i montažu uređaja. Svaka aktivnost predstavlja merljivi događaj, pri čemu imamo objektivni kriterijum na osnovu koga određujemo da li je aktivnost završena. Prema tome, kraj svake aktivnost može da bude miljokaz, pri čemu tabela 3.2 prikazuje miljokaze faze 2.

SLIKA 3.1 Faze, koraci i aktivnosti u okviru projekta



Ovo analitičko razlaganje daje naručiocu i projektantu viziju svega što je obuhvaćeno gradnjom kuće. Slično tome, analizom razvoja ili održavanja softvera, identifikovanjem faza, koraka i aktivnosti, postižemo da projektni tim i naručilac bolje shvate šta je sve obuhvaćeno gradnjom ili održavanjem sistema. U poglavlju 2 videli smo da procesni model obezbeđuje pogled sa višeg nivoa apstrakcije na faze i korake, pa prema tome predstavlja koristan način započinjanja analize projekta. U kasnijim poglavljima videćemo da glavne faze, kao što su inženjering zahteva, implementacija ili testiranje, obuhvataju više aktivnosti, pri čemu svaka od njih doprinosi kvalitetu proizvoda ili procesa.

Struktura poslova i grafovi aktivnosti

Analiza ove vrste nekada se opisuje kao proces generisanja **strukture poslova** (*work breakdown structure*, WBS) u okviru projekta, budući da opisuje projekat kao skup diskretnih poslova. Treba imati u vidu da aktivnosti i miljokazi predstavljaju elemente koje naručilac i razvojni tim mogu da koriste za praćenje razvoja ili održavanja. Naručilac će možda hteti da vidi napredak u bilo kojoj tački projekta. Kao učesnici u razvoju, možemo da ukažemo na aktivnosti, da bismo predstavili poslove koji su u toku i miljokaze, sa ciljem prezentacije stepena gotovosti posla. Međutim, struktura poslova u okviru projekta ništa ne govori o međuzavisnostima pojedinih grupa poslova ili delova projekta koji se istovremeno razvijaju.

Svaku aktivnost možemo opisati sa četiri parametra: prethodnikom, trajanjem, krajnjim rokom i krajnjom tačkom. **Prethodnik** je događaj ili skup događaja koji moraju da se dogode pre nego što posmatrana aktivnost započne. Prethodnik opisuje skup uslova koji omogućavaju započinjanje aktivnosti. **Trajanje** je vreme potrebno za kompletiranje aktivnosti. **Krajnji rok** je datum do kojeg aktivnost mora biti okončana, što je često određeno ugovorenim krajnjim rokovima. **Krajnja tačka** je obično miljokaz

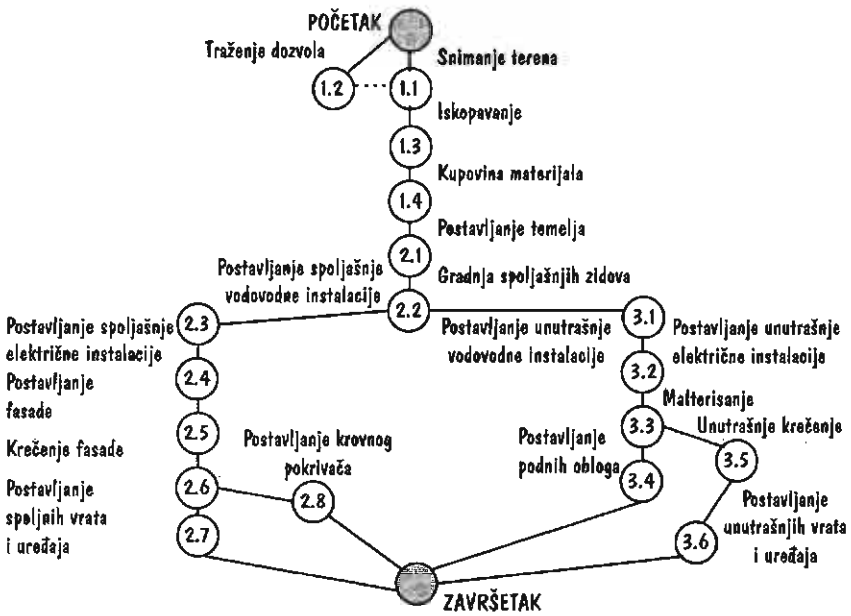
ili međuproizvod, označavajući da je aktivnost završena. Pomoću navedenih parametara možemo ilustrovati odnose između aktivnosti. Radi opisivanja zavisnosti, možemo nacrtati **graf aktivnosti.** Čvorovi grafa predstavljaju miljokaze projekta, a grane koje povezuju čvorove predstavljaju sadržane aktivnosti. Slika 3.2 prikazuje graf aktivnosti za poslove opisane u fazi 2, date u tabeli 3.1.

Tabela 3.1 Faze, koraci i aktivnosti gradnje kuće

Faza 1: Zemljani radovi	Faza 2: Gradnja kuće
Korak 1.1: raščišćavanje i krčenje	*Korak 2.1: Pripremanje gradilišta*
Aktivnost 1.1.1: Uklanjanje stabala	Aktivnost 2.1.1: Premer zemljišta
Aktivnost 1.1.2: Uklanjanje panjeva	Aktivnost 2.1.2: Traženje dozvola
Korak 2: sađenje trave	Aktivnost 2.1.3: Kopanje jame za temelj jame
Aktivnost 1.2.1: Aeracija tla	Aktivnost 2.1.4: Kupovina materijala
Aktivnost 1.2.2: Sejanje	*Korak 2.2: Gradnja spoljašnjosti*
Aktivnost 1.2.3: Zalivanje i plevljenje	Aktivnost 2.2.1: Postavljanje temelja
Korak 1.3: Planiranje žbunova i stabala	Aktivnost 2.2.2: Gradnja spoljašnjih zidova
Aktivnost 1.3.1: Dobijanje žbunova i stabala	Aktivnost 2.2.3: Postavljanje spoljašnje vodovodne instalacije
Aktivnost 1.3.2: Kopanje rupa	Aktivnost 2.2.4: Spoljašnji električni radovi
Aktivnost 1.3.3: Sadnja žbunova i stabala	Aktivnost 2.2.5: Stavljanje fasadne obloge
Aktivnost 1.3.4: Učvršćivanje stabala i zagrtanje đubrivom	Aktivnost 2.2.6: Krečenje fasade
	Aktivnost 2.2.7: Montiranje vrata i uređaja
	Aktivnost 2.2.8: Montiranje krova
	Korak 2.3: Unutrašnji završni radovi
	Aktivnost 2.3.1: Postavljanje unutrašnje vodovodne mreže
	Aktivnost 2.3.2: Postavljanje unutrašnje električne instalacije
	Aktivnost 2.3.3: Malterisanje
	Aktivnost 2.3.4: Unutrašnje krečenje
	Aktivnost 2.3.5: Postavljanje krovnog pokrivača
	Aktivnost 2.3.6: Postavljanje vrata i uređaja

Tabela 3.2 Miljokazi prilikom gradnje kuće

1.1	Gotov premer
1.2	Dobijene dozvole
1.3	Završen iskop
1.4	Spreman materijal
2.1	Postavljeni temelji
2.2	Gotovi spoljašnji zidovi
2.3	Završena spoljna vodovodna instalacija
2.4	Završeni spoljašnji električni radovi
2.5	Fasada gotova
2.6	Krečenje fasade gotovo
2.7	Postavljena vrata i uređaji
2.8	Krov završen
3.1	Završena unutrašnja vodovodna instalacija
3.2	Postavljena unutrašnja električna instalacija
3.3	Malterisanje završeno
3.4	Krečenje unutrašnjosti završeno
3.5	Postavljene podne obloge
3.6	Postavljeni vrata i uređaji



SLIKA 3.2 Graf aktivnosti gradnje kuće

Graf aktivnosti obezbeđuje vizualizaciju značajnih karakteristika projekta. Na primer, sa slike 3.2 je jasno da niti jedna od dve vodoinstalaterske aktivnosti ne može da počne pre nego što se dostigne miljokaz 2.2. Drugim rečima, 2.2 je prethodnik unutraš-

njim i spoljašnjim vodoinstalaterskim radovima. Osim toga, slika prikazuje mogućnost paralelnog izvršavanja aktivnosti. Na primer, neke od unutrašnjih i spoljašnjih aktivnosti su nezavisne (kao što je postavljanje gipsanih zidova, povezivanje spoljašnje električne i vodovodne instalacije, kao i drugo što vodi do odgovarajućih miljokaza 2.6 i 3.3). Aktivnosti na levoj putanji, što se tiče njihovog započinjanja, ne zavise od aktivnosti na desnoj putanji, tako da se na njima može istovremeno raditi. Obratimo pažnju na to da postoji isprekidana linija koja ide od traženja dozvola (čvor 1.1) do premera zemljišta (čvor 1.1). Ta linija ukazuje da te aktivnosti moraju da se dovrše pre nego što može da počne kopanje zemlje (aktivnost koja dovodi do miljokaza 1.3). Međutim, kako ne postoji stvarna aktivnost koja se izvodi nakon dostizanja miljokaza 1.2, da bi se stiglo do miljokaza1.1, isprekidana linija pokazuje odnos neaktivnosti.

Važno je da se shvati da grafovi aktivnosti zavise od shvatanja paralelne prirode zadataka. Ako rad ne može da se odvija paralelno, onda graf (koji je uglavnom pravolinijski) nije pogodan za opisivanje koordinacije zadataka. Grafovi moraju da odražavaju paralelizme koji su mogući u stvarnosti. U našem primeru građenja kuće, jasno je da će neke radove, kao što su vodoinstalaterski, realizovati drugi izvršioci a ne oni koji će obaviti električarske radove. Međutim, u razvoju softvera, gde ima veštijih i manje veštih, teoretski paralelizam možda ne odražava onaj paralelizam koji je moguć u stvarnosti. Ograničavanje broja ljudi na projektu za posledicu ima da ista osoba radi više stvari jednu za drugom, iako bi se one mogle, kod većeg razvojnog tima, paralelno raditi.

Procena gotovosti

Graf aktivnosti možemo učiniti još korisnijim dodavanjem informacija o proceni vremena koja je potrebna za dovršetak svake aktivnosti. Za posmatranu aktivnost, odgovarajuću granu grafa označavamo brojem koji predstavlja procenjeno vreme realizacije. Na primer, za aktivnosti u fazi 2 u tabeli 2.1, na graf aktivnosti sa slike 3.2 možemo da dodamo procenu broja dana potrebnih za dovršavanje svake aktivnosti. Tabela 3.3 sadrži procene za svaku aktivnost.

Kao rezultat dobijamo graf prikazan na slici 3.3. Napomenimo da su miljokazi 2.7, 2.8, 3.4 i 3.6 prethodili završetku. To jest, moraju se proći svi ti miljokazi da bi se smatralo da je projekat završen. Nule koje se nalaze uz grane od tih čvorova do završetka prikazuju da dodatno vreme nije potrebno. Postoji takođe nula koja se podrazumeva na grani od čvora 1.2 do 1.1, jer isprekidanoj vezi nije pridruženo dodatno vreme.

Grafički opis projekta govori mnogo o vremenskim rokovima projekta. Na primer, pošto smo procenili da su za završetak prve aktivnost potrebna 3 dana, ne možemo se nadati da ćemo stići do miljokaza 1.1 pre 15. dana. Iskopavanje (aktivnost 1.3) ne može da započne pre miljokaza 1.1 i 1.2, što znači da iskopavanje ne može započeti pre 16. dana.

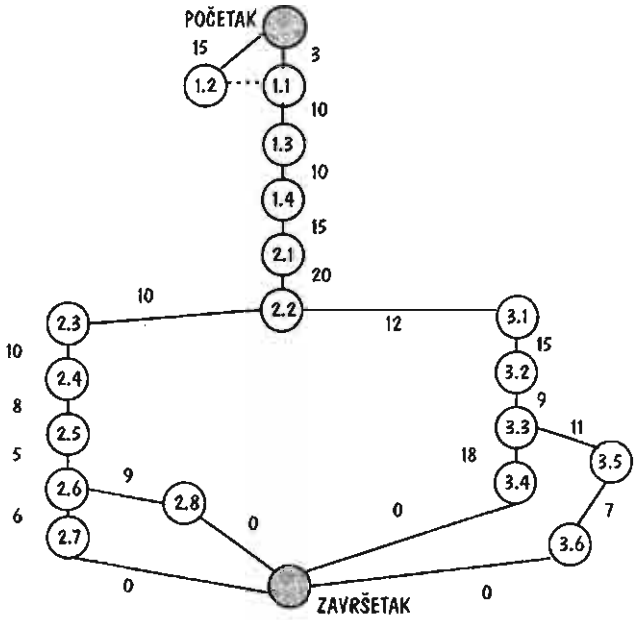
Ovakva analiza putanja između miljokaza u projektu naziva se **metod kritične putanje** - *CPM* (*Critical Path Method*). Putanje mogu da pokažu minimalno potrebno vreme za završetak projekta, dajući nam procenu trajanja svake aktivnosti. Pored toga, CPM otkriva aktivnosti koje su najkritičnije za završetak projekta na vreme.

Tabela 3.3 Aktivnosti i potrebno vreme

Aktivnost	Potrebno vreme (u danima)
Korak 1: Priprema gradilišta	
Aktivnost 1.1: Premer zemljišta	3
Aktivnost 1.2: Traženje dozvola	15
Aktivnost 1.3: Kopanje temeljne jame	10
Aktivnost 1.4: Kupovina materijala	10
Korak 2: Gradnja spoljašnjosti	
Aktivnost 2.1: Postavljanje temelja	15
Aktivnost 2.2: Gradnja spoljašnjih zidova	20
Aktivnost 2.3: Postavljanje spoljašnje vodovodne instalacije	10
Aktivnost 2.4: Spoljašnji električni radovi	10
Aktivnost 2.5: Stavljanje fasadne obloge	8
Aktivnost 2.6: Krečenje fasade	5
Aktivnost 2.7: Montiranje vrata i uređaja	6
Aktivnost 2.8: Montiranje krova	9
Korak 3: Unutrašnji završni radovi	
Aktivnost 3.1: Postavljanje unutrašnje vodovodne mreže	12
Aktivnost 3.2: Postavljanje unutrašnje električne instalacije	15
Aktivnost 3.3: Malterisanje	9
Aktivnost 3.4: Unutrašnje krečenje	18
Aktivnost 3.5: Postavljanje krovnog pokrivača	11
Aktivnost 3.6: Postavljanje vrata i uređaja	7

Da bismo videli kako CPM funkcioniše, razmotrićemo ponovo primer izgradnje kuće. Prvo, primećujemo da aktivnosti koje dovode do miljokaza 1.1 (premer) i 1.2 (traženje dozvola) mogu da se izvode istovremeno. Pošto iskopavanje (aktivnost koja kulminira na miljokazu 1.3) ne može da počne sve do 16. dana, na raspolaganju imamo 15 dana za završetak premera terena, pri čemu je realno potrebno vreme samo 3 dana. Na isti način, za svaku aktivnost, definisanu grafom, možemo izračunati dva vremena: realno potrebno vreme i raspoloživo vreme. **Stvarno potrebno vreme** ili **stvarno vreme** aktivnosti je vreme koje je potrebno za okončanje aktivnosti, dok **raspoloživo vreme** predstavlja vreme koje je, u okviru vremenskih rokova, raspoloživo za njeno okončanje. **Latentno vreme** ili **klizanje** aktivnosti predstavlja razliku između raspoloživog i stvarno potrebnog vremena za okončanje posmatrane aktivnosti:

Latentno vreme = raspoloživo vreme - stvarno potrebno vreme



SLIKA 3.3 Graf aktivnosti sa trajanjima

Drugi način posmatranja latentnog vremena je poređenje najranijeg i najkasnijeg vremena započinjanja aktivnosti, a da se pri tome ne prouzrokuje kašnjenje u projektu. Na primer, premer terena može da započne 1. dana pa je najranije vreme početka 1. Međutim, budući da traženje i dobijanje dozvole traje 15 dana, premer može da započne najkasnije 13. dana, a da ne dođe do zastoja aktivnosti na projektu. Prema tome,

Latentno vreme = vreme najkasnijeg započinjanja - vreme najranijeg započinjanja

Izračunaćemo sada latentno vreme za aktivnosti iz našeg primera, da bismo videli šta nam to govori o rokovima projekta. Računaćemo latentno vreme ispitujući sve putanje od početka do završetka. Kao što smo videli, mora da protekne 15 dana da bi se stiglo do miljokaza 1.1. i 1.2. Dodatnih 55 dana se koristi da bi se stiglo do miljokaza 1.3, 1.4, 2.1 i 2.2. U ovoj tački, postoje četiri moguće putanje:

1. Kretanje preko miljokaza 2.3 do 2.7 na grafu zahteva 39 dana.
2. Kretanje preko miljokaza 2.3 do 2.8 na grafu zahteva 42 dana.
3. Kretanje preko miljokaza 3.1 do 3.4 na grafu zahteva 54 dana.
4. Kretanje preko miljokaza 3.1 do 3.6 na grafu zahteva 54 dana.

Da bi se projekat okončao, mora se doći do miljokaza 2.7, 2.8 i 3.6, pa je zato vremenski rok ograničen najdužom putanjom. Kao što može da se vidi sa slike 3.3 i prethodnih proračuna, dve putanje na desnoj strani zahtevaju 124 dana, dok dve putanje sa leve strane zahtevaju manje dana. Za računanje latentnog vremena, možemo se vraćati putanjom i utvrditi količinu latentnog vremena za svaku aktivnost koja vodi ka čvoru. Prvo, uočavamo da je kod najduže putanje latentno vreme jednako nuli. Zatim, ispitujemo svaki od preostalih čvorova radi računanja latentnog vremena za aktivnosti koje

do njega vode. Na primer, za aktivnosti koje vode do miljokaza 2.3, 2.4, 2.5 2.6 i 2.8 imamo 54 dana, ali su za njihovo kompletiranje potrebna samo 42 dana. Prema tome, ovaj deo grafa ima 12 dana latencije. Slično tome, deo grafa koji uključuje aktivnosti od 2.3 do 2.7 zahteva samo 39 dana, tako da duž te putanje imamo 15 dana latencije. Idući grafom unazad na ovaj način duž, možemo da izračunamo najranije vreme započinjanja i latentno vreme za svaku aktivnost. Zatim, računamo najkasnije vreme započinjanja svake aktivnosti idući od završetka, preko svih čvorova, do početka. Tabela 3.4 prikazuje rezultate: latentno vreme za sve aktivnosti na slici 3.3. (Kod miljokaza 2.6, putanja se grana na 2.7 i 2.8. Najkasnija vremena započinjanja data u tabeli 3.4 računata su pomoću putanje od 2.6 do 2.8, umesto od 2.6 do 2.7.)

Najduža putanja ima nulto klizanje za svaki od svojih čvorova, jer ona definiše da li projekat jeste ili nije unutar vremenskih rokova. Zbog toga se ona naziva kritičnom putanjom. Prema tome, **kritična putanja** je ona kod koje je latentno vreme za svaki čvor jednako nuli. Kao što ćemo videti iz našeg primera, može da postoji više od jedne kritične putanje.

Tabela 3.4 Latentno vreme projektnih aktivnosti

Aktivnost	Najranije vreme započinjanja	Najkasnije vreme započinjanja	Latentno vreme
1.2	1	13	12
1.2	1	1	0
1.3	16	16	0
1.4	26	26	0
2.1	36	36	0
2.2	51	51	0
2.3	71	83	12
2.4	81	93	12
2.5	91	103	12
2.6	99	111	12
2.7	104	119	15
2.8	104	116	12
3.1	71	71	0
3.2	83	83	0
3.3	98	98	0
3.4	107	107	0
3.5	107	107	0
3.6	118	118	0
Završetak	124	124	0

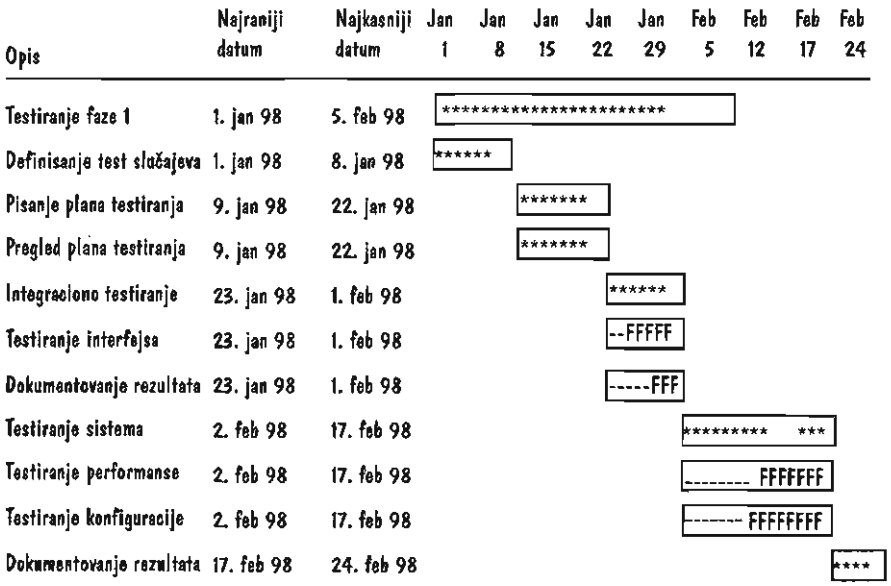
Kako kritična putanja nema klizanja, ne postoji rezerva za kompenzaciju odstupanja tokom izvođenja aktivnosti duž te putanje.

Obratimo pažnju na to šta se događa kada aktivnost na kritičnoj putanji započne kasnije (tj. kasni u odnosu na svoj najraniji početak). Zakasneli početak, u slučaju da nema

latentnog vremena, pomera unapred sve naredne aktivnosti na kritičnoj putanji čineći da i one kasne. I kod aktivnosti koje se ne nalaze na kritičnoj putanji aktivnosti koje slede mogu izgubiti pridruženo latentno vreme. Prema tome, graf aktivnosti pomaže da razumemo uticaj klizanja unutar rokova.

Razmotrimo šta se događa ako graf aktivnosti ima petlje. Petlje nastaju kada se aktivnosti ponavljaju. Na primer, u primeru izgradnje kuće, građevinski inspektor može da zahteva ponovno izvođenje vodoinstalaterskih radova. U razvoju softvera, pregled projekta može da zahteva ponovnu izradu specifikacije dizajna ili specifikacije zahteva. Pojava takvih petlji može da izmeni kritičnu putanju jer se aktivnosti unutar petlje izvršavaju više puta. U tom slučaju, uticaj na rokove nije moguće tako lako odrediti.

Slika 3.4 predstavlja prugasti dijagram koji prikazuje neke aktivnosti u projektu razvoja softvera, uključujući informacije o njihovim najranijim i najkasnijim počecima. Ovakav grafikon predstavlja tipičan proizvod automatizovanih alata za upravljanje projektom. Horizontalne pruge predstavljaju trajanje svake aktivnosti. Pruge sačinjene od zvezdica ukazuju na kritičnu putanju. Aktivnosti predstavljene crticama i slovima F ne nalaze se na kritičnoj putanji, pri čemu F označava klizanja ili latentno vreme.



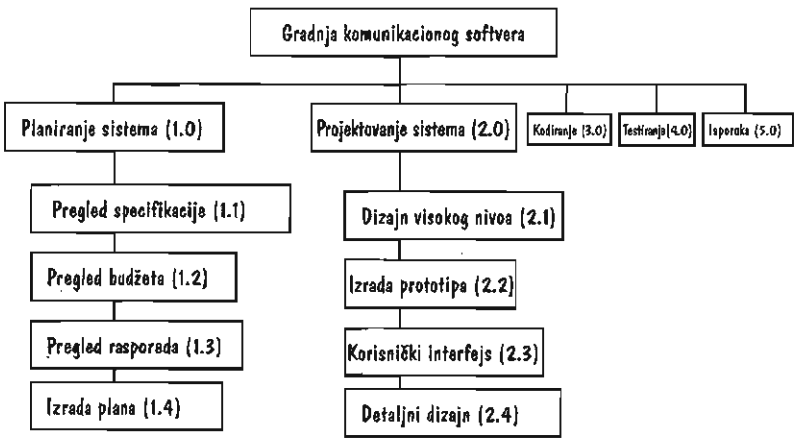
SLIKA 3.4 CPM prugasti dijagram

Analiza kritičnih putanja u rasporedu aktivnosti na projektu ukazuje na to ko mora da čeka i na šta čeka tokom realizacije projekta. Ona takođe ukazuje na aktivnosti koje se moraju okončati na vreme da bi se izbeglo kašnjenje. Ova vrsta analize može da se poboljša na više načina. U primeru izgradnje kuće pretpostavlja se da tačno znamo trajanje svake aktivnosti. To međutim često nije slučaj. Uglavnom raspolažemo procenjenim trajanjem aktivnosti koje je zasnovano na poznavanju sličnih projekata i događaja. Prema tome, svakoj aktivnosti možemo dodeliti verovatno trajanje. Pri tome se koristi neka funkcija za raspodelu verovatnoće i svaka aktivnost dobija očekivanu vrednost i varijansu. Drugim rečima, umesto poznavanja tačnog trajanja, procenjujemo intervale u koje

će stvarno vreme da upadne. Očekivana vrednost je tačka u okviru intervala, a varijansa predstavlja opseg intervala. Sigurno vam je poznata standardna raspodela verovatnoće koja se naziva normalna raspodela, čiji grafik predstavlja kriva u obliku zvona. Tehnika evaluacije i pregleda programa - PERT (*Program Evaluation and Review Technique*) je vrlo zastupljena tehnika analize kritične putanje koja pretpostavlja normalnu raspodelu. (Za više informacija o PERT-u pogledati referencu Hiller i Lieberman [1967]). PERT određuje kolika je verovatnoća da vreme najranijeg početka aktivnosti bude blizu vremenu predviđenom rokovima. Koristeći informacije kao što su raspodela verovatnoće, najkasnije i najranije vreme početka, kao i graf aktivnosti, program PERT može da izračuna kritične putanje i identifikuje aktivnosti koje predstavljaju verovatna uska grla. Mnogi rukovodioci projekata koriste CPM ili PERT metode za analizu svojih projekata. Međutim, te metode su primenljive jedino na stabilne projekte sa paralelnim aktivnostima. Ako su aktivnosti projekta uglavnom linearne (redne), onda se praktično sve aktivnosti nalaze na kritičnoj putanji i predstavljaju kandidate za uska grla. Osim toga, ako projekat zahteva preprojektovanje ili preradu, postoji verovatnoća da će doći do promene grafa aktivnosti i kritične putanje.

Alati za praćenje napretka

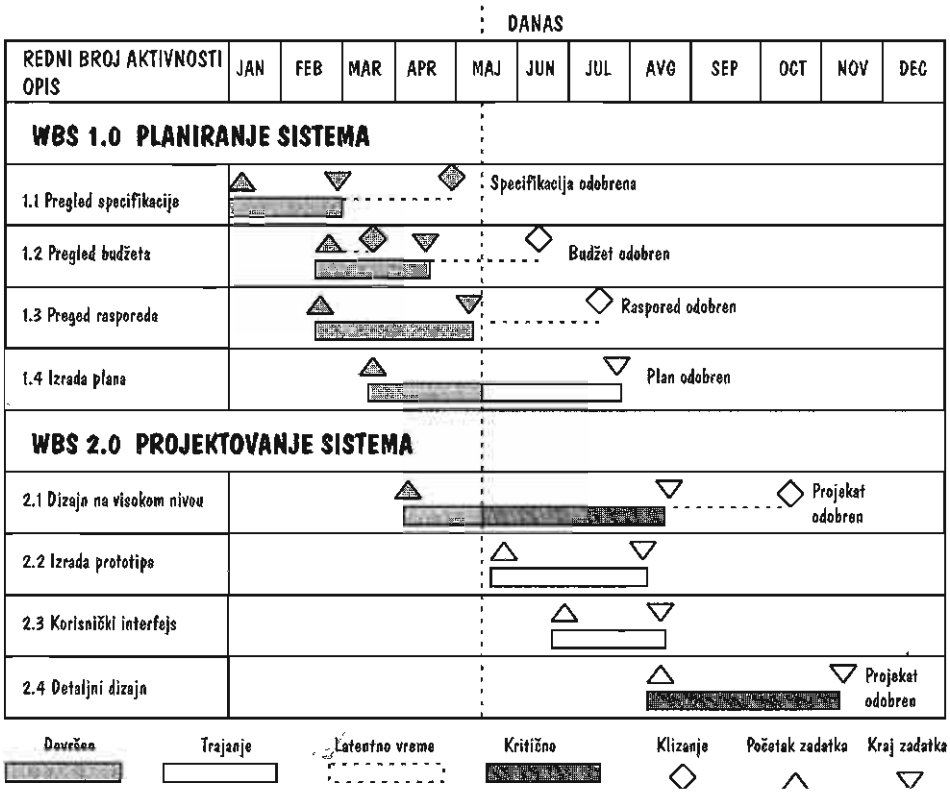
Postoje mnogi alati koji se koriste za praćenje napretka projekta. Neki alati su ručni, drugi predstavljaju jednostavne aplikacije za rad sa tabelama, a postoje i sofisticirani alati sa kompleksnim grafičkim predstavama. Da bismo videli koje vrste alata mogu da budu korisne pri projektovanju, razmotrićemo strukturu posla, prikazanu na slici 3.5. Ovde je opšti cilj izrada sistema koji uključuje komunikacioni softver, gde je rukovodilac projekta predstavio rad na projektu u pet koraka: planiranje sistema, projektovanje sistema, kodiranje, testiranje i isporuka. Radi jednostavnosti, usredsredićemo se samo na prva dva koraka. Korak 1 je podeljen na četiri aktivnosti: pregled specifikacija, pregled budžeta, pregled rokova i izrada plana projekta. Slično tome, dizajn sistema čine: dizajn visokog nivoa, izrada prototipa, projektovanje korisničkog interfejsa i detaljni dizajn.



SLIKA 3.5 Primer strukture posla

Mnogi softverski sistemi za upravljanje projektima crtaju strukturu posla i pomažu rukovodiocu projekta u praćenju napretka po svakom koraku i aktivnosti. Na primer, paket za upravljanje projektima ima mogućnost crtanja **gantograma**, tj. prikaza projekta na kom su aktivnosti prikazane paralelno, dok je stepen dovršenosti označen bojom ili sličicom. Takav dijagram pomaže rukovodiocu projekta da razume koje aktivnosti mogu da se izvode paralelno i da uoči aktivnosti koje pripadaju kritičnoj putanji.

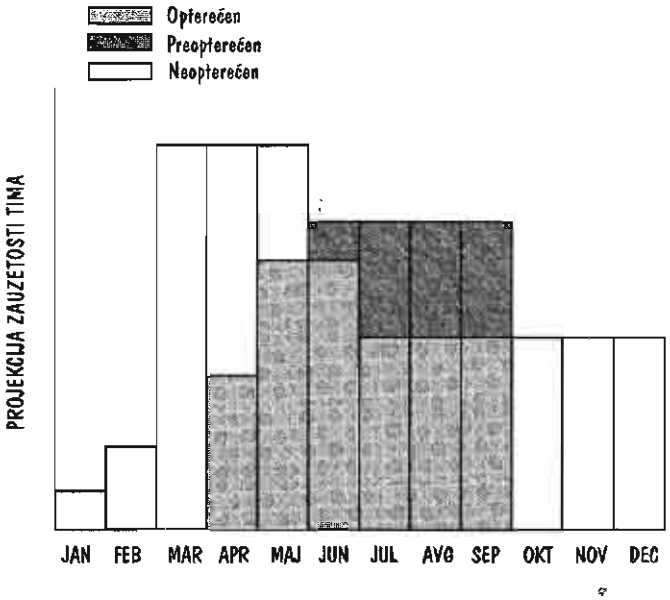
Slika 3.6 predstavlja gantogram za strukturu posla prikazanu na slici 3.5. Projekat je započet u januaru, a isprekidana vertikalna linija sa natpisom „danas", ukazuje da je projektni tim upravo na polovini meseca maja. Horizontalne pruge ilustruju napredak svake aktivnosti, a boja pruge označava stepen gotovosti, trajanje ili kritičnost. Romb pokazuje mesta klizanja, a trougao početak i završetak aktivnosti. Gantogram je sličan CPM dijagramu sa slike 3.4, ali sadrži više informacija.



SLIKA 3.6 Gantogram strukture posla

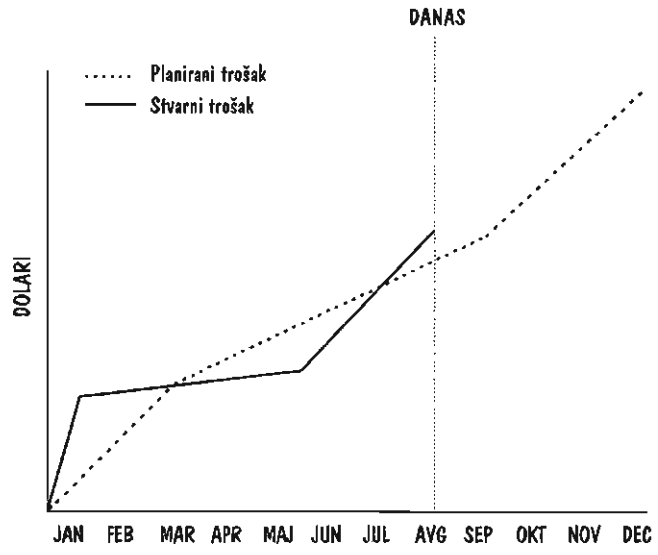
Jednostavni dijagrami i grafikoni mogu takođe da pruže i informacije o resursima. Na primer, na slici 3.7 dat je grafikon odnosa između ljudi dodeljenih projektu i ljudi potrebnih u svakoj fazi razvoja. Taj prikaz je tipičan za grafikone koji su proizvod alata za upravljanje projektima. Lako je uočiti da je tokom januara, februara i marta postojala potreba za ljudima ali da do dodele nije došlo. U aprilu i maju, neki članovi tima rade, ali nedovoljno da bi uradili zadatak koji se od njih zahteva. S druge strane, jasno je prikazan period tokom kojeg postoji višak članova tima: od početka juna do kraja septembra. Dodela resursa u

slučaju razmatranog projekta je očigledno neuravnotežena. Izmenom ulaznih podataka grafikona, moguće je promeniti dodelu resursa radi pokušaja smanjenja preopterećenja, pronalazeći najbolje opterećenje resursa za ispunjenje postavljenih rokova.



SLIKA 3.7 Histogram resursa

Videćemo kasnije u ovom poglavlju kako je moguće proceniti troškove razvoja. Alati za upravljanje projektima prate stvarne troškove i porede ih sa procenjenim vrednostima, tako da je moguće oceniti i stanje budžeta. Slika 3.8 ilustruje primer praćenja troškova.



SLIKA 3.8 Praćenje planiranih troškova prema stvarnim troškovima.

Kombinovanjem praćenja budžeta sa praćenjem osoblja možemo, pomoću alata za upravljanje projektima, odrediti najbolje resurse u skladu sa ograničenjima vezanim za budžet.

3.2 OSOBLJE NA PROJEKTU

Da bismo odredili rokove za projekat i procenili potreban rad i troškove, treba da barem otprilike znamo broj ljudi koji će raditi na projektu, na kojim poslovima, kao i koje specijalnosti i iskustvo oni moraju posedovati radi efikasnog izvršavanja dodeljenih poslova. U ovom odeljku ćemo videti kako se donose odluke ko će šta da radi i kako se tim može organizovati.

Uloge i karakteristike tima

U poglavlju 2 smo ispitali nekoliko modela procesa za izradu softvera, gde svaki od njih oslikava povezivanja različitih aktivnosti u procesu razvoja softvera. Nezavisno od modela, postoje određene aktivnosti koje su neophodne u svakom softverskom projektu. Na primer, svaki projekat zahteva ljude koji kontaktiraju sa naručiocem da bi utvrdili šta on želi i do kog roka. Neki rade na dizajniranju sistema, dok drugi pišu ili testiraju programe. U ključne aktivnosti spadaju:

1. analiza zahteva;
2. dizajn sistema;
3. dizajn programa;
4. implementacija programa;
5. testiranje;
6. obuka;
7. održavanje;
8. osiguranje kvaliteta.

Međutim, ne izvršava svaki zadatak ista osoba ili grupa ljudi. Dodeljivanje radnika zadacima zavisi od veličine projekta, iskustva i obučenosti osoblja. Postoji velika prednost dodeljivanja različitih odgovornosti različitim grupama ljudi, čime se omogućavaju „provere i izbalansiranost" koje obezbeđuju otkrivanje grešaka u ranim fazama razvoja. Na primer, pretpostavimo da testeri i projektanti i programeri nisu isti radnici. Testiranje novog ili modifikovanog softvera obuhvata testiranje sistema, gde učesnici u razvoju demonstriraju naručiocu da sistem radi onako kako je specifikovano. Testeri moraju da definišu i dokumentuju način na koji će se sprovoditi testovi, kao i kriterijume po kojima se vidi da demonstrirane funkcije i karakteristike performansi odgovaraju zahtevima koje je definisao naručilac. Tim za testiranje generiše svoj plan testiranja na osnovu dokumenata sa zahtevima, i ne mora da zna kako su sklopljeni unutrašnji delovi sistema. On se usredsređuje na funkcionalnost sistema, jer nije opterećen predubeđenjem o tome kako rade hardver i softver. Ovakav pristup olakšava testerima da otkriju greške i propuste koje su napravili projektanti ili programeri. To je jedan od razloga što „čistu sobu" koristi nezavisan tim testera, kao što ćemo videti u kasnijim poglavljima (Mills, Dyer i Linger 1987).

Iz sličnih razloga, korisno je da projektanti programa i projektanti sistema ne budu iste osobe. Projektanti programa su duboko upleteni u detalje vezane za kôd, pri čemu ponekad zanemaruju širu sliku o tome kako sistem treba da radi. Videćemo u kasnijim poglavljima da tehnike kao što su prolazak kroz projekat, inspekcije i pregledi okupljaju ove dve grupe projektanata radi sprovođenja dvostruke provere projekta pre prelaska na kodiranje i obezbeđenja kontinuiteta u procesu razvoja.

Videli smo u poglavlju 1 da postoje i mnoge druge uloge razvojnog tima ili tima za održavanje. Tokom proučavanja svakog od glavnih zadataka razvoja u sledećim poglavljima, opisaćemo i članove ovih timova koji te zadatke realizuju.

Nakon donošenja odluke o ulogama članova projektnog tima, neophodno je doneti odluku o ljudima koji su za svaku od njih potrebni. Osoblje projekta može da se razlikuje na više načina, i nije dovoljno da kažemo da projekat, na primer, zahteva jednog analitičara, dva projektanta i pet programera. Dve osobe koje rade na istom radnom mestu mogu da se razlikuju po najmanje jednom od sledećih kriterijuma:

- sposobnost izvršavanja posla;
- interesovanje za posao;
- iskustvo sa sličnim aplikacijama;
- iskustvo sa sličnim alatima ili jezicima;
- iskustvo sa sličnim tehnikama;
- iskustvo sa sličnim razvojnim okruženjem;
- obučenost;
- sposobnost komunikacije sa drugima;
- sposobnost podele odgovornosti sa drugima;
- rukovodilačke sposobnosti.

Svaka od tih karakteristika može da utiče na sposobnost pojedinca da produktivno obavlja posao. Te varijacije pomažu da objasnimo zašto jedan programer može da napiše zadati potprogram za jedan dan, dok je drugome za to potrebna cela nedelja. Te razlike mogu da postanu kritične, ne samo za projekciju rokova, već i za uspeh samog projekta.

Da bismo razumeli učinak svakog radnika, moramo poznavati njegovu sposobnost da izvrši posao koji mu je poveren. Neki radnici su dobri u sagledavanju šire problematike, ali možda ne uživaju u detaljima ako se od njih traži da rade na delu velikog projekta. Takvim pojedincima više odgovaraju zadaci u projektovanju sistema ili testiranju, nego zadaci u projektovanju programa ili kodiranju. Ponekad je sposobnost proporcionalna opuštenosti. Na časovima ili na projektima, možda ste radili sa ljudima koji se bolje snalaze programirajući na jednom jeziku nego na drugom. Ima učesnika u razvoju koji se više pouzdaju u svoje projektantske sposobnosti nego u programersku veštinu. Osećaj opuštenosti u posluje važan, jer ljudi su obično produktivniji ako imaju samopouzdanje u onome što rade.

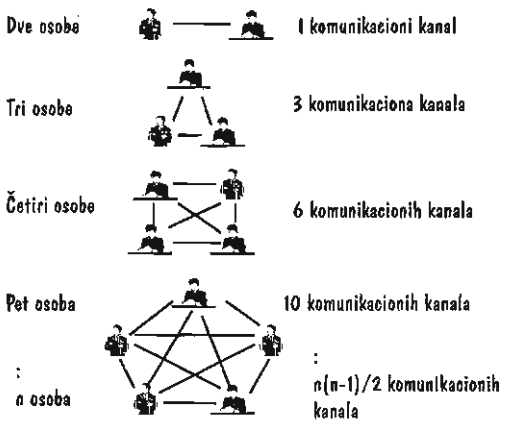
Zainteresovanost za posao može takođe da odredi nečiju uspešnost na projektu. Bez obzira što je neki radnik možda veoma dobar u nekom poslu, on je možda zainteresovaniji da proba nešto novo umesto da ponavlja nešto što je urađeno mnogo puta do tada. Prema tome, novina na poslu predstavlja ponekad faktor generisanja interesovanja za posao. S druge strane, uvek postoje ljudi koji više vole da rade samo ono što znaju i što

najbolje rade, umesto da istražuju nove teritorije. Bez obzira na to ko je izabran za određeni zadatak, važno je da ga obavlja sa entuzijazmom, bez obzira na razloge.

I dva pojedinca sa istim sposobnostima i interesovanjima uvek mogu da se razlikuju po iskustvu ili stepenu poznavanja sličnih aplikacija, alata ili tehnika. Neko ko je već uspešno koristio jezik C za pisanje komunikacionog kontrolera, verovatno će na C-u brže napisati sledeći komunikacioni kontroler, nego neko ko nema nikakvo iskustvo sa jezikom C, niti zna šta komunikacioni kontroler radi. Prema tome, izbor radnika koji će biti angažovani na projektu ne obuhvata samo individualne sposobnosti i veštine, već takođe i iskustvo i obučenost.

U svakom razvojnom projektu ili održavanju softvera, članovi razvojnog tima komuniciraju međusobno, sa korisnicima, kao i sa naručiocem. Na napredovanje projekta ne utiče samo stepen komunikacije, već i sposobnost pojedinaca da saopšte svoje ideje. Softverski otkazi mogu da budu rezultat prekida u komunikaciji i međusobnog nerazumevanja, tako da broj ljudi koji komuniciraju može da utiče na kvalitet rezultujućeg proizvoda. Slika 3.9 prikazuje kako brzo može da naraste broj komunikacionih kanala. Uvećanje radnog tima sa dve na tri osobe, utrostručuje broj mogućih komunikacionih kanala. U opštem slučaju, ako projekat ima n zaposlenih, onda postoji $n(n - 1)/2$ parova koji možda žele da komuniciraju, kao i $2n - 1$ mogućih timova koji se mogu formirati za rad na manjim delovima projekta. Prema tome, projekat koji angažuje samo 10 ljudi ima 45 komunikacionih kanala, i od njih se može organizovati 1023 tima koji se mogu obrazovati radi razvijanja podsistema!

Mnogi projekti uključuju više ljudi koji moraju da dele odgovornost za realizaciju jedne ili više aktivnosti. Oni koji rade na jednom aspektu razvoja, moraju imati poverenja da će drugi članovi tima obaviti svoj deo posla. U okviru nastave, obično imate potpunu kontrolu nad projektima koje radite. Počinjete sa zahtevima (koje obično propisuje predavač), projektujete rešenje problema, skicirate kôd, zatim pišete stvarni kôd i na kraju testirate programe. Međutim, kada radite u timu, bilo u školi bilo za račun poslodavca ili naručioca, morate biti sposobni da delite poslove. To ne samo da zahteva verbalnu razmenu ideja i rezultata, već i pisanu dokumentaciju o tome šta planirate da radite i šta ste uradili. Morate da prihvatite rezultate drugih ne ponavljajući njihov rad. Mnogi pojedinci imaju poteškoće sa ovakvom podeom odgovornosti.



SLIKA 3.9 Komunikacioni kanali u sklopu projekta

Kontrola je posebno pitanje pri upravljanju projektom. Neki ljudi su dobri u prenošenju zadataka drugima. Taj aspekt interakcije među ljudima zavisi i od zadovoljstva koje ljudi osećaju prilikom obavljanja posla. Oni kojima ne prija da prisiljavaju kolege da se pridržavaju rokova i dokumentuju svoj kôd, ili im ne prijaju sastanci sa naručiocem, nisu dobri kandidati za aktivnosti koje podrazumevaju rukovođenje drugima.

Prema tome, različiti aspekti iz radne biografije radnika mogu da utiču na kvalitet projektnog tima. Rukovodilac projekta treba da poznaje interesovanja i sposobnosti svake osobe, kada bira tim. Podsetnik 3.1 objašnjava kako sastanci i njihova organizacija mogu da poboljšaju ili ometaju napredak projekta. Kao što ćemo videti kasnije u ovom poglavlju, iskustvo radnika i sposobnost komunikacije mogu da imaju dramatične efekte na vrednost projekta i rokove realizacije.

PODSETNIK 3.1 UČINIMO DA SASTANCI DOPRINESU NAPRETKU PROJEKTA

Deo komunikacije u okviru softverskog projekta odigrava se na sastancima, bilo lično ili putem telekonferencije ili elektronske konverzacije. Međutim, sastanci mogu da oduzmu dosta vremena a da ne postignu mnogo. Dressler (1995) kaže da „održavanje loših sastanaka može dosta da košta„. sastanak osam ljudi koji zarađuju 40 000 dolara godišnje košta 320 dolara na sat, uključujući troškove plate i beneficija. To je skoro 6 dolara na minut". Uobičajene zamerke u vezi sa sastancima su:

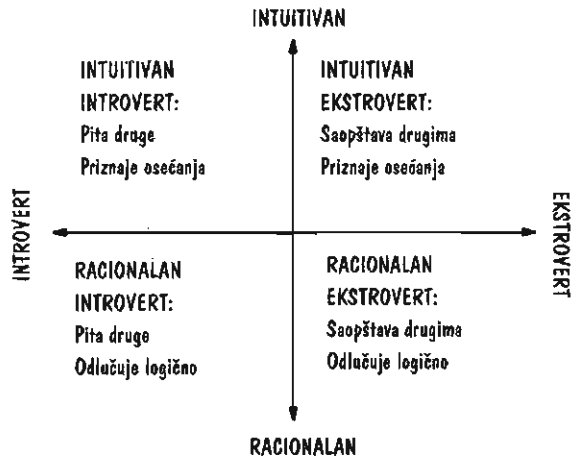
- Svrha sastanka je nejasna.
- Učesnici su nepripremljeni.
- Važni ljudi su odsutni ili kasne.
- Rasprava skreće sa teme.
- Neki učesnici sastanka ne diskutuju o hitnim pitanjima. Umesto toga, oni međusobno raspravljaju, dominiraju raspravom, ili u njoj uopšte ne učestvuju.
- Odluke donete na sastanku se nakon toga ne sprovode.

Dobro upravljanje projektom uključuje planiranje svih aktivnosti razvoja softvera, pa i sastanaka. Postoji više načina da osiguramo produktivnost sastanka. Prvo, rukovodilac mora jasno da saopšti članovima projektnog tima ko treba da bude na sastanku, kada sastanak počinje i kada se završava, kao i šta je njegova svrha. Drugo, svaki sastanak treba da ima dnevni red u pisanoj formi, koji se, ako je moguće, unapred dostavlja. Treće, neko treba da preuzme odgovornost za usmeravanje diskusije i razrešavanje sukoba. Četvrto, neophodno je nedvosmisleno zadužiti odgovorne za realizaciju donetih odluka. Najvažnije, neophodno je smanjiti broj sastanaka, kao i broj ljudi koji na njima moraju da učestvuju.

Stilovi rada

Različiti ljudi imaju različite omiljene stilove za interakciju sa saradnicima radi razumevanja problema koji u toku rada iskrsavaju. Na primer, možda vam se više sviđa da detaljno analizirate sve moguće informacije pre donošenja odluke, dok se vaše kolege, prilikom donošenja većine važnih odluka, možda oslanjaju na „dobar osećaj". Omiljeni stil rada možemo da posmatramo u funkciji dva parametra: načina na koji se razmenjuju misli i rađaju ideje, i uticaja emocija na donošenje odluka. Tokom razmene ideja, ima ljudi koji saopštavaju drugima svoja razmišljanja, dok drugi od kolega traže sugestije pre nego što formiraju vlastiti stav. Jung (1959) prve naziva **ekstrovertima**, a druge **introvertima**. Jasno je da stil komunikacije utiče na način interakcije sa drugim učesnicima u projektu. Slično tome, **intuitivne** osobe zasnivaju svoje odluke na osećaju i emotivnoj reakciji na probleme. Druge osobe su **racionalne**, i donose odluke pre svega ispitujući činjenice i pažljivo razmatrajući sve mogućnosti.

Raznolikost stilova rada možemo da posmatramo pomoću grafikona na slici 3.10, gde je stil komunikacije na horizontalnoj, a stil odlučivanja na vertikalnoj osi. Što je više neko ekstrovertan, stil rada na grafikonu pada više desno. Slično tome, što emocije imaju više udela pri odlučivanju, ide se nagore. **Racionalni ekstroverti** teže saopštavanju svojih ideja i ne dozvoljavaju da „osećaj" utiče na njihove odluke. Oni saopštavaju svojim kolegama ono što žele da oni znaju, ali retko traže dodatne informacije pre nego što to urade. Njihovo rasuđivanje se oslanja na logiku, a ne na emocije. **Racionalni introverti** takođe izbegavaju emotivne rasprave, ali su skloni da deo vremena posvete razmatranju svih mogućih pravaca delovanja. Racionalni introverti su skupljači informacija. Oni se ne osećaju ugodno pri donošenju odluke sve dok nisu uvereni da u rukama imaju sve činjenice.



SLIKA 3.10 Stilovi rada

Nasuprot njima, **intuitivni ekstroverti** većinu odluka zasnivaju na emotivnim reakcijama, težeći da ih obrazlože drugima umesto da traže argumente. Oni se u kreativnosti oslanjaju na intuiciju i pri tome često sugerišu neouobičajene prilaze rešavanju problema. Intuitivni introvert je takođe kreativan, ali kreativnost primenjuje samo nakon prikupljanja dovoljno informacija na osnovu kojih donosi odluku. Winston Churchill je bio

intuitivni introvert. Kada je želeo da sazna nešto o nekom pitanju, čitao je detaljno sav raspoloživi materijal na određenu temu. Često je donosio odluke zasnovane na osećanju stečenom takvim čitanjem (Manchester 1983).

Da bismo videli kako stilovi rada utiču na interakcije u okviru projekta, razmotrićemo više tipičnih profila radnika. Kai, koja je racionalni ekstrovert, procenjuje svoje kolege na osnovu rezultata koje oni daju. Kada donosi odluku, njen najveći prioritet je efikasnost. Prema tome, ona želi da zna samo koliki je dobitak. Ona ispituje opcije i njihove moguće efekte, ali joj nije potrebno da gleda dokumente ili sluša objašnjenja koja podržavaju pojedine opcije. Ako uviđa da gubi vreme ili je njena efikasnost na neki način ometena, ona brani svoj autoritet da bi ponovo preuzela kontrolu nad situacijom. Stoga je Kai dobra u brzom donošenju ispravnih odluka.

Marcel, koji je racionalni introvert, vrlo je različit od svoje koleginice Kai. On ocenjuje svoje kolege prema tome koliko su zauzeti, i ima malo tolerancije za one za koje smatra da se ne angažuju intenzivno sve vreme. On je dobar radnik, kome se dive zbog energije koju posvećuje poslu. Njegova reputacija dobrog radnika mu je veoma važna i ponosan je na sebe i svoju tačnost i temeljnost. On ne voli da donosi odluke bez potpune informacije. Kada se od njega traži da napravi prezentaciju, Marcel će to učiniti samo nakon prikupljanja svih relevantnih informacija o zadatoj temi.

Macel deli kancelariju sa Davidom, intuitivnim ekstrovertom. Dok Marcel ne donosi odluku bez kompletnog poznavanja situacije, David više voli da sledi svoja osećanja. Često veruje svojoj intuiciji, zasnivajući svoju odluku na profesionalnoj proceni, umesto na sporoj i pažljivoj analizi prikupljenih informacija. Kako je samopouzdan, David teži da saopšti svoje nove ideje drugima koji rade na projektu. On je kreativan i uživa kada drugi odaju priznanje njegovim idejama. David voli da radi u okruženju gde postoji visok stepen interakcije među članova tima.

Ying, koja je intuitivni introvert, je takođe srećna zbog pažnje koju joj posvećuju njene kolege. Ona je osetljiva i svesna svojih emotivnih reakcija na određene ljude ili probleme. Za nju je veoma važno da se dopadne svojim kolegama. Ying je dobar slušalac i zato je član projektnog tima kome se drugi okreću da iznesu svoja osećanja. Ying treba mnogo vremena za donošenje odluke, ne samo zato što joj je potrebna kompletna informacija, već i zbog toga što želi da donese ispravnu odluku. Osetljiva je na to šta drugi misle o njenoj sposobnosti i idejama. Analizira situaciju na sličan način kako to radi Marcel, ali sa različitim fokusom. Marcel posmatra sve činjenice i procene, dok Ying takođe ispituje odnose zavisnosti i uticaj emocija.

Jasno je da niko ne spada u potpunosti u jednu od četiri navedene kategorije. Razni ljudi imaju različite namere, ali kao osnova za opisivanje tih namera i sklonosti može da posluži šema data na slici 3.10.

Komunikacija je kritična za uspeh projekta, a stil rada definiše i stil komunikacije. Na primer, ako ste odgovorni za deo projekta koji kasni, Kai i David će vam verovatno reći kada vaš deo posla mora da bude okončan. David će ponuditi nekoliko ideja kako da posao ponovo uhvati priključak sa rokovima, dok će vam Kai dati nove rokove kojih treba da se pridržavate. Međutim, Marcel i Ying će verovatno da pitaju kada će rezultati da budu gotovi. Marcel će, tokom analiziranja vlastitih opcija, želeti da zna zašto taj deo nije završen, dok će Ying pitati da li postoji nešto što bi ona mogla da uradi i time pomogne.

Razumevanje stilova rada može da pomogne da budete fleksibilni u pristupu drugim članovima projektnog tima, kao i u pristupu korisnicima i naručiocu. Stilovi rada naročito daju informacije o prioritetima drugih ljudi. Ako su prioriteti i interesovanja kolege

različiti od vaših, informaciju možete da mu predstavite u kategorijama koje su njemu bitne. Pretpostavimo da je Claude naručilac, a da vi pripremate prezentaciju o statusu projekta. Ako je Claude introvert, tada znamo da on više voli da prikuplja, nego da daje informacije. Prema tome, možemo prezentaciju da organizujemo tako da ona većim delom govori o tome kakva je struktura projekta i kako projekat napreduje. Međutim, da je Claude ekstrovert, mogli bismo da uključimo i pitanja, i time mu pružimo priliku da nam saopšti šta želi ili šta mu je potrebno. Slično tome, da je Claude intuitivan, mogli bismo da iskoristimo njegovu kreativnost tražeći od njega nove ideje. U slučaju da je racionalan, prezentacija treba da obuhvati pre činjenice ili ilustracije, nego procene ili osećanja. Prema tome, stilovi rada imaju uticaj na interakcije između naručioca, razvojnog tima i korisnika.

Od stila rada zavisi i izbor radnika kome će se dodeliti neki zadatak. Na primer, intuitivni radnici možda više vole projektovanje i razvoj (gde se zahtevaju nove ideje) umesto održavanja programa i projekta (gde se zahteva usredsređivanje na detalje i analizu složenih rešenja).

Organizacija projekta

Projektni timovi za razvoj i održavanje softvera ne sastoje se od ljudi koji nezavisno ili nekoordinisano rade. Umesto toga, članovi tima su organizovani na način koji doprinosi brzoj izradi kvalitetnih proizvoda. Izbor odgovarajuće strukture projekta zavisi od nekoliko činilaca, a to su:

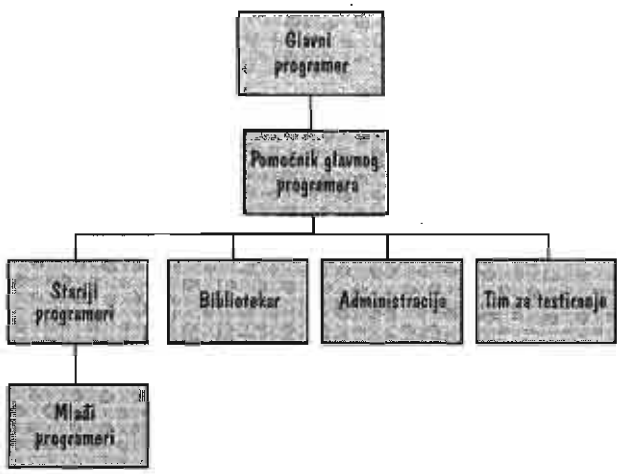
- biografije i stilovi rada članova tima;
- broj ljudi u timu;
- stilovi rukovođenja koje primenjuju naručilac i razvojni tim.

Dobri rukovodioci projekata su svesni tih pitanja i traže članove tima koji su dovoljno fleksibilni za interakciju sa svim učesnicima, bez obzira na stil rada.

Jedna popularna organizaciona struktura je tim sa glavnim programerom, koja je prvi put korišćena u IBM-u (Baker 1972). U pristupu sa **glavnim programerom**, samo jedna osoba je potpuno odgovorna za projektovanje i razvoj sistema. Svi drugi članovi tima podnose izveštaje glavnom programeru, koji ima poslednju reč prilikom donošenja svih odluka. Glavni programer nadgleda druge, dizajnira sve programe i raspoređuje posao realizacije koda drugim članovima tima. Glavnom programeru asistira njegov zamenik, čiji je glavni zadatak da služi kao zamena glavnom programeru kada je to neophodno. Timu je na usluzi i bibliotekar, koji je odgovoran za održavanje celokupne projektne dokumentacije. Bibliotekar takođe prevodi i povezuje kôd, i radi preliminarna testiranja svih modula koji su predati u biblioteku. Ova podela rada omogućava programerima da se usredsrede na ono što najbolje znaju, a to je programiranje.

Organizacija tima sa glavnim programerom prikazana je na slici 3.11. Prenošenjem odgovornosti za sve odluke na glavnog programera, ova organizacija smanjuje potrebu za komunikacijom u toku projekta. Svaki član tima mora često da komunicira sa glavnim programerom, ali ne i sa drugim članovima tima. Prema tome, ako se tim sastoji od n-1 programera plus glavni programer, tim može da uspostavi samo n-1 od mogućih $n(n - 1)/2$ komunikacionih kanala (jedan kanal za svaku interakciju člana tima sa glavnim programerom). Na primer, umesto da sami rešavaju problem, programeri se jednostavno obraćaju glavnom programeru. Slično tome, glavni programer pregleda celokupni projekat i kôd, otklanjajući potrebu za pregledom koda koju obavljaju članovi tima.

SLIKA 3.11 Organizacija tima sa glavnim programerom



Iako je tim sa glavnim programerom hijerarhijski, mogu da se formiraju i članovi tima za realizaciju nekog posebnog zadatka. Na primer, jedan član tima ili više njih mogu da formiraju administrativnu grupu radi pružanja statusnih informacija o trenutnim troškovima i rokovima realizacije razmatranog projekta.

Jasno je da glavni programer mora da bude dobar u brzom donošenju odluka, tako da je glavni programer verovatno ekstrovert. Međutim, ako većinu članova tima čine introverti, tada ova organizaciona šema nije najbolja za dati projekat. Alternativa tome je ideja programiranja **„bez egoizma"**, kao što je opisao Weinberg (1971). Umesto da odgovornost bude koncentrisana u jednoj tački, ovaj pristup posmatra svakoga učesnika kao podjednako odgovornog. Pored toga, sam proces se razlučuje od pojedinaca. Kritikuje se proizvod ili rezultat, a ne pojedinci. Struktura tima počiva na demokratskim principima, gde se svi članovi tima izjašnjavaju o odlukama, bilo da se one tiču projektnih rešenja ili tehnika za testiranje.

Naravno, postoji više drugih načina za organizovanje projekta razvoja ili održavanja, pri čemu dva prethodno opisana predstavljaju ekstremne slučajeve. Koja struktura je pogodnija? Što više ljudi učestvuje na projektu, to je veća potreba za formalnom organizacijom. Naravno, razvojni tim sa samo tri ili četiri člana ne zahteva uvek detaljno definisanu organizacionu strukturu. Međutim, tim od nekoliko desetina učesnika mora da ima dobro definisanu organizaciju. U stvari, kompanija ili naručilac mogu razvojnom timu da nametnu organizacionu strukturu, zasnovanu na ranijim uspesima, želji da se napredak prati na određeni način, ili na želji da se smanji broj kontakata. Na primer, naručilac možda insistira da tim za testiranje bude potpuno nezavisan od timova za dizajn i razvoj programa.

Istraživači nastavljaju istraživanje o tome kako struktura projektnog tima utiče na rezultujuće proizvode i na koji način, u datoj situaciji, odabrati organizaciju koja najviše odgovara. Istraživanje koje je sprovedeno od strane američke Nacionalne fondacije za nauku (1983) pokazalo je da projekti sa velikim stepenom izvesnosti, stabilnosti, uniformnosti i ponavljanja mogu da se završe još efikasnije pomoću hijerarhijske organizacione strukture, kao što je npr. tim sa glavnim programerom. Ti projekti zahtevaju malo komunikacije između članova tima, tako da dobro odgovaraju organizacijama koje stavljaju naglasak na pravila, specifikaciju, formalizme i jasno definisanu hijerarhiju.

Tabela 3.5 Poređenje organizacionih struktura

Jako strukturisane	*Slabo strukturisane*
Visoka izvesnost	Neizvesnost
Ponavljanje	Nove tehnike ili tehnologije
Veliki projekti	Mali projekti

S druge strane, ako projekat uključuje značajan stepen neizvesnosti, demokratičniji pristup bi bolje odgovarao. Na primer, ako zahtevi mogu da se menjaju tokom razvoja, projekat ima viši stepen neizvesnosti. Slično tome, ako pretpostavimo da naručilac gradi novi hardverski deo koji treba da bude interfejs ka sistemu, a tačna specifikacija hardvera nije još poznata, onda je nivo neizvesnosti visok. U ovom slučaju, učešće u donošenju odluka i ohrabrivanje otvorene komunikacije mogu da budu efikasni (slabo definisana hijerarhija).

Tabela 3.5 zbirno prikazuje karakteristike projekta sa njihovom sugerisanom organizacionom strukturom. Veliki projekat sa visokom izvesnošću i ponavljanjem verovatno zahteva veoma strukturisanu organizaciju, za razliku od malog projekta, sa novim tehnikama i visokim stepenom izvesnosti, gde je dozvoljena i manja strukturisanost. Podsetnik 3.2 opisuje potrebu uravnoteženog strukturisanja i kreativnosti.

PODSETNIK 3.2 STRUKTURISANOST U ODNOSU NA KREATIVNOST

Kunde (1997) izvestio je o rezultatima eksperimenata koje je izvela Sally Philipp, koja je radila na razvoju materijala za softversku obuku. Kada Philipp predaje na seminaru za rukovodioce, ona deli slušaoce u dve grupe. Svakoj grupi je dodeljivan isti zadatak: da sagrade maketu hotela od kartona i lepka. Neki timovi su strukturisani, gde članovi tima imaju jasno definisane odgovornosti. Drugi su prepušteni sami sebi, bez datih smernica ili strukture, osim osnovnog zadatka da izgrade hotel. Philipp tvrdi da su rezultati uvek isti. Ona kaže: „Nestrukturisani timi uvek prave neverovatno nadahnute, višespratne Tadž Mahale, i nikada ne završi na vreme. Strukturisani timovi prave motele tipa Days Inn (mali funkcionalni hotel bez dodatnih sadržaja), ali uvek završe na vreme i upravo postavljaju stolice oko bazena kada kažem da je vreme isteklo."

Jedan način kako ona strukturiše tim je podsticanje članova tima da postave krajnje rokove. Celokupni zadatak razbija se na male podzadatke, a pojedinačni članovi tima su odgovorni za procene vremena. Krajnji rokovi pomažu da se spreči stalno menjanje zahteva, tzv. „razvlačenje", tj. da se spreči ubacivanje nepotrebnih funkcionalnosti u proizvod.

Eksperti izjavljuju povodom Kundeovog članka da dobro upravljanje projektom traži da se pronađe ravnoteža između strukture i kreativnosti. Prepušteni samo sredstvima sa kojima oni barataju, projektanti softvera usmeravaju svoju pažnju jedino na funkcionalnost i kreativnost, ne obazirući se na krajnje rokove i specifikovani obim posla. Mnogi stručnjaci za upravljanje projektima iznose slične tvrdnje. Nažalost, većina tih informacija je zasnovana na anegdotama, a ne na čvrstim empirijskim istraživanjima.

РЕПУБЛИКА СРПСКА
Природно - математички
факултет Бања Лука
Ф И З И К А

Pomenuta dva tipa organizacione strukture mogu da se kombinuju tamo gde je to prikladno. Na primer, od programera može da se traži da samostalno razviju podsistem, koristeći pri tome pristup bez egoizma u okviru hijerarhijske organizacije. Ili, na primer, tim za testiranje slabo strukturisanog projekta može sebi da nametne hijerarhijsku organizaciju i odredi jednu osobu kao odgovornu za sve glavne odluke vezane za proces testiranja.

3.3 POTREBAN RAD

Jedno od prelomnih gledišta planiranja u upravljanju projektom jeste shvatanje koliko će projekat približno da košta. Prekoračenja troškova mogu dovesti do odustajanja od projekta, dok potcenjivanje troškova može da primora projektni tim da dosta vremena utroši bez finansijske nadoknade. Kao što je opisano u podsetniku 3.3, ima više uzroka netačnih procena. Dobra procena troškova u ranoj fazi životnog ciklusa projekta pomaže rukovodiocu projekta da utvrdi broj ljudi neophodnih za razvoj i dogovori da odgovarajuće osoblje bude raspoloživo upravo kada je to neophodno.

Iz budžeta projekta pokriva se više vrsta troškova: sredstva, osoblje, metodi i alati. Troškovi sredstava obuhvataju: hardver, prostor, nameštaj, telefone, modeme, grejanje i klimatizaciju, kablove, diskove, papir, olovke, aparate za fotokopiranje i sve druge stavke koje čine fizičko okruženje u kojem će raditi projektni tim. Kod nekih projekata, to okruženje može već da postoji, tako da su troškovi dobro poznati i lako se procenjuju. Međutim, kod nekih projekata, okruženje tek treba da se stvori. Na primer, novi projekat može da zahteva sef, dupli pod, kontrolu temperature ili vlažnosti ili poseban nameštaj. U tom slučaju, troškovi mogu da se procene, ali oni mogu da variraju u odnosu na početne proračune, kako se okruženje izgraduje ili menja. Na primer, instalacija kablova u zgradi može da izgleda jednostavna, sve dok građevinci ne otkriju da se radi o zdanju od posebnog istorijskog značaja, tako da kablovi moraju da se provlače oko zidova, umesto da idu kroz njih.

Nekada postoje skriveni troškovi koji nisu očigledni rukovodiocima i razvojnom timu. Na primer, neke studije ukazuju na to da je programeru potreban određeni minimum prostora i mirnog okruženja da bi mogao efikasno da radi. McCue (1978) je izvestio svoje kolege u IBM-u da minimalni standard za radni prostor programera treba da bude oko 9,3 kvadratnih metara rezervisanog prostora, sa 2,8 kvadratnih metara horizontalne radne površine. Prostor takođe zahteva pregrade od poda do plafona radi zaštite od buke. DeMarco i Lister (1987) u svom radu sugerišu da programeri koji su oslobođeni od telefonskih poziva i nezvanih gostiju, pokazuju veću efikasnost i proizvode bolje proizvode nego oni koji stalno prekidaju rad.

Drugi projektni troškovi obuhvataju kupovinu softvera i alata za podršku aktivnostima razvoja. Pored alata za projektovanje i kodiranje sistema, za projekat može da se nabavi i softver za obradu zahteva, organizaciju dokumentacije, testiranje koda i za praćenje izmena, kao i generisanje test podataka, zatim podršku sastancima grupe itd. Te alate za „računarski podržani razvoj softvera", tj. **CASE** (*Computer-Aided Software Engineering*) **alati**, ponekad uslovljava sam naručilac, ili predstavljaju sastavni deo standardizovanog procesa razvoja softvera unutar kompanije.

Kod većine projekata, najveći udeo u troškovima ima uloženi rad. Moramo da odredimo koliko dana ljudskog rada mora da se uloži za dovršenje projekta. Uloženi rad sigurno predstavlja troškovnu komponentu sa najvećim stepenom neizvesnosti. Videli smo kako na potrebno vreme za dovršenje zadatka utiču stil rada, organizacija projekta, sposobnost, in-

teresovanje, iskustvo, obuka i druge karakteristike radnika. Pored toga, kada grupa članova tima mora međusobno da komunicira i da se konsultuje, potreban uloženi rad se uvećava vremenom provedenim na sastancima ili obuci, i u dokumentovanju.

PODSETNIK 3.3 UZROCI NETAČNIH PROCENA

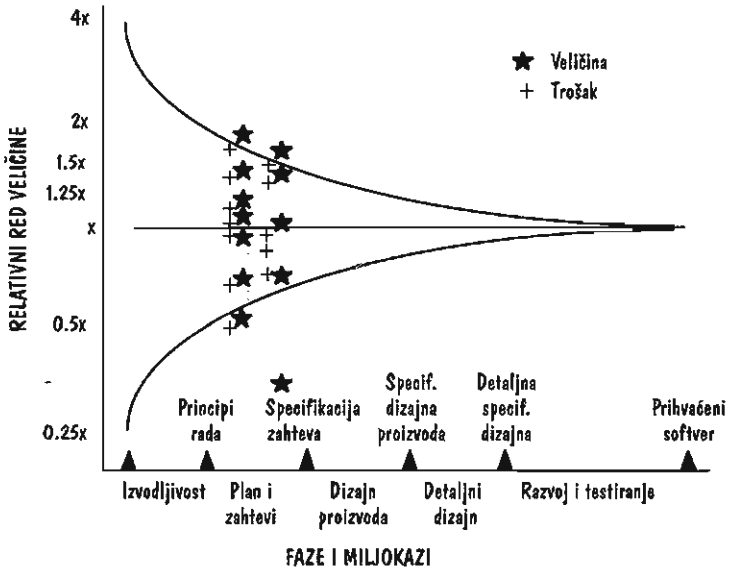
Lederer i Prasad (1992) su istraživali kako se procenjuju troškovi u 115 različitih organizacija. Trideset pet procenata ispitanih rukovodilaca na petostepenoj Likertovoj lestvici, izjavili su da su njihove dotadašnje procene bile „umereno nezadovoljavajuće" ili „vrlo nezadovoljavajuće". Ključne uzroke ovako loših procena videli su u:

- čestim izmenama od strane korisnika;
- predviđenim zadacima;
- sami korisnici ne razumeju svoje zahteve;
- nedovoljnoj analitičnosti prilikom izrade procene;
- loša koordinisanost između projektovanja, tehničkih usluga, operative, administracije podataka, i drugih funkcija tokom razvoja;
- nedostatak adekvatnih metoda ili smernica potrebnih za procenjivanje.

Aspekti projekta koji imaju ključnu ulogu za dobru procenu su:

- složenost predloženog rešenja;
- integracija sa postojećim sistemima;
- složenost pojedinačnih programa u sistemu;
- veličina sistema izražena brojem funkcija ili programa;
- sposobnosti članova projektnog tima;
- iskustvo projektnog tima sa datom primenom;
- očekivana učestanost ili obim mogućih izmena u zahtevima korisnika;
- iskustvo projektnog tima u radu sa korišćenim programskim jezikom;
- sistem za upravljanje bazom podataka;
- broj članova projektnog tima;
- obim standarda za programiranje i dokumentovanje;
- raspoloživost alata kao što su generatori aplikacija;
- iskustvo tima sa konkretnim hardverom.

Predviđanje troškova, rokova i potrebnog rada moraju da se urade u ranim fazama životnog ciklusa projekta, jer direktno utiču na dodeljivanje resursa i mogućnost realizacije projekta. (Ako su procenjeni troškovi previsoki, naručilac može da otkaže projekat.) Međutim, procene treba revidirati više puta u toku životnog ciklusa projekta. Kako se neki aspekt projekta promeni, procena se, na bazi kompletnijih informacija o karakteristikama projekta, rafinira. Slika 3.12 ilustruje kako neizvesnost na početku projekta može da ima uticaja na tačnost proračunatih troškova i veličinu projekta(Boehm i dr. 1995).



SLIKA 3.12 Promene u preciznosti procene tokom napredovanja projekta (Boehm i dr. 1995)

Zvezdica predstavlja procenjenu veličinu stvarnih projekata, dok znak plus predstavlja proračun troškova. Krive u obliku levka koje se međusobno približavaju idući na desno, odražavaju Boehmov osećaj kako procene postaju tačnije sa većim saznanjima o projektu. Imajmo u vidu da, kada detalji o projektu još uvek nisu poznati, procena može da se razlikuje od eventualnog stvarnog troška i do 4 puta. Tokom donošenja odluka o proizvodu i procesu, taj faktor se smanjuje. Mnogi stručnjaci tvrde da procene variraju samo 10 procenata u odnosu na stvarne vrednosti, ali Boehmovi podaci ukazuju na to da se takve procene obično događaju samo kada je projekat pri kraju, a tada je suviše kasno da bi bile od koristi za upravljanje projektom.

Da bismo se pozabavili potrebom izrade tačnih procena, softverski inženjeri su razvili tehnike za snimanje odnosa između potrebnog rada i karakteristika radnika, projektnih zahteva, kao i drugih činilaca koji imaju uticaja na vreme, potrebni rad i troškove razvoja softverskog sistema. U ostatku ovog poglavlja, usredsredićemo se na tehnike za procenu potrebnog rada.

Stručni sud

Mnoge metode za procenu potrebnog rada počivaju na stručnoj proceni. Neke od metoda su neformalne, zasnovane na iskustvu rukovodioca sa sličnim projektima. Prema tome, tačnost predviđanja zasnovana je na kompetenciji, iskustvu, objektivnosti i opažanju onoga ko vrši procenu. U svom najjednostavnijem obliku, to je približna ocena rada potrebnog za izgradnju celokupnog sistema ili odgovarajućih podsistema. Kompletna procena toga šta je sve potrebno, može da se odredi pomoću analize „odozdo nagore" ili „odozgo nadole".

U mnogo slučajeva za procenu potrebnog rada se koriste analogije. Ako smo već gradili sistem koji je vrlo sličan predloženom, onda možemo da koristimo sličnost kao

osnovu za procenjivanje. Na primer, ako je sistem A sličan sistemu B, onda cena izrade sistema A treba da bude približna ceni izrade sistema B. Tu analogiju možemo da proširimo ako kažemo da ako je sistem A otprilike upola složen kao sistem B, tada sistem A treba da bude upola jeftiniji od sistema B.

Proces pravljenja analogije može da se formalizuje ako od nekoliko stručnjaka tražimo da naprave tri pretpostavke: pesimističku (x), optimističku (y) i najverovatniju (z). Onda je naša procena srednja vrednost beta raspodele verovatnoće, na osnovu sledeće formule: $(x + 4y + z)/6$. Koristeći tu tehniku, pravimo procenu koja predstavlja „normalizaciju" pojedinačnih procena.

Metod Delfi koristi stručnu procenu na drugačiji način. Od stručnjaka se traži da u tajnosti naprave pojedinačna predviđanja, na osnovu svoje stručnosti i koristeći postupak po vlastitom izboru. Zatim se računa srednja procena i ona se predstavlja grupi. Svaki stručnjak ima mogućnost da, ako to želi, revidira svoju procenu. Proces se ponavlja sve dok nijedan stručnjak ne želi da izvrši reviziju. Neki korisnici metoda Delfi raspravljaju o srednjoj proceni, pre nego što se napravi nova procena, dok u ostalim trenucima ne dozvoljavaju raspravu. U još jednoj varijaciji, anonimna obrazloženja procene svih eksperata cirkulišu između njih.

Wolverton (1974) je formulisao jedan od prvih modela napora pri razvoju softvera. Njegova matrica softverskih troškova obuhvata njegovo iskustvo sa troškovima projekata u okviru TRW-a, američke kompanije za razvoj softvera. Kao što je prikazano u tabeli 3.6, naziv reda predstavlja tip softvera, a kolona označava teškoće, koje zavise od dva faktora: da li je problem star (S) ili je nov (N), kao i da li je lak (L), umeren (U) ili težak (T). Element matrice je trošak po redu koda, kao što je standardizovano prema istorijskim podacima iz TRW-a. Da bismo koristili matricu, potrebno je da podelimo predloženi softverski sistem na module. Zatim, procenjujemo veličinu svakog modula zavisno od broja redova koda. Koristeći matricu, računamo trošak po modulu, a zatim sabiramo po svim modulima. Na primer, pretpostavimo da imamo sistem sa tri modula: jednim ulazno/izlaznim modulom koji je star i lak, jednim algoritamskim modulom koji je nov i težak, i jednim modulom za upravljanje podacima koji je stari i umeren. Ako pretpostavimo da će moduli imati 100, 200 i 100 redova koda, prema redosledu u kojem smo ih naveli, na osnovu Wolvertonovog modela procenjeni trošak iznosi $(100 \times 17) + (200 \times 35) + (100 \times 31) = 11\,800$ dolara.

Tabela 3.6 Matrica troškova prema Wolvertonovom modelu

	Težina					
Tip softvera	SL	SU	ST	NL	NU	NT
Kontrola	21	27	30	33	40	49
Ulaz/izlaz	17	24	27	28	35	43
Pre/postprocesor	16	23	26	28	34	42
Algoritam	15	20	22	25	30	35
Upravljanje podacima	24	31	35	37	46	57
Vremenski-faktor	75	75	75	75	75	75

Kako je model zasnovan na podacima TRW-a i kao osnov koristi dolar iz 1974. godine, nije primenljiv na današnji razvoj softvera. Međutim, tehnika je korisna i može lako da se prenese na neko drugo razvojno okruženje, ili na održavanje softvera.

U opštem slučaju, pošto su eksperimentalni modeli većinom zasnovani na stručnoj proceni, ona ujedno predstavlja izvor svih njihovih nepreciznosti. Modeli se zasnivaju na sposobnosti stručnjaka da odredi koji su projekti slični i na koji način. Međutim, projekti koji izgledaju vrlo slično, mogu u stvari da budu prilično različiti. Na primer, danas trkač može da pretrči jedan kilometar za 2,5 minuta. Maratonska trka zahteva od trkača da pretrči 42 kilometra i 195 metara. Ako ekstrapoliramo vreme od 2,5 minuta, možemo da očekujemo da trkač pretrči maraton za 1 sat i 45 minuta. Međutim, nijedan maraton nije nikada pretrčan za manje od 2 sata. To znači da postoje karakteristike trčanja maratona koje se veoma razlikuju od trčanja na jedan kilometar. Slično tome, često neki projekat ima karakteristike koje ga čine vrlo različitim od drugog projekta, ali te karakteristike nisu očigledne.

Čak i kada znamo da se jedan projekat razlikuje od drugog, ne znamo uvek kako te razlike utiču na troškove. Strategija proporcionalnosti je nepouzdana, jer troškovi projekta nisu uvek linearni. Dve osobe ne mogu da proizvedu kôd dvostruko brže nego jedna osoba. Dodatno vreme je verovatno potrebno za komunikaciju i koordinaciju ili za prilagođavanje razlika u interesovanju, sposobnosti i iskustvu. Sackman, Erikson i Grant (1968) nalaze da je odnos produktivnosti između najboljih i najgorih programera u proseku 10 prema 1, bez mogućnosti da se lako odredi veza između iskustva i performanse. Slično tome, skorašnja Hudgesova (1996) studija utvrđuje veliku raznolikost u načinima projektovanja i razvoju softvera, tako da model koji funkcioniše u jednoj organizaciji možda nije primenljiv na drugu. Hudges takođe napominje da prethodno iskustvo i poznavanje raspoloživih resursa predstavljaju glavne faktore za utvrđivanje troškova.

Stručni sud pati ne samo od promenljivosti i subjektivnosti, već i od zavisnosti od aktuelnih podataka. Ti podaci na kojima je zasnovan model stručnog suda moraju da odražavaju aktuelnu praksu, pa se prema tome moraju često ažurirati. Pored toga, mnoge tehnike stručnog prosuđivanja su jednodimenzionalne i zanemaruju veliki broj činilaca koji mogu da utiču na potreban rad za realizaciju razmatranog projekta.

Iz tog razloga, praktičari i istraživači su se okrenuli algoritamskim metodama za procenu potrebnog rada.

Algoritamske metode

Istraživači su napravili modele koji izražavaju odnos između potrebnog rada i faktora koji na njega utiču. Ti modeli se obično opisuju pomoću jednačina, u kojima je napor funkcija, a više činilaca (kao što su iskustvo, veličina i tip aplikacije) nezavisne promenljive. Većina tih modela potvrđuje da je veličina projekta najuticajniji faktor u jednačinama gde se rad izražava na sledeći način:

$$E = (a + bS^c)m(X)$$

gde je S procenjena veličina sistema, a a, b i c su konstante. X je vektor faktora troškova, xq do xn, a m je korekcioni faktor koji od njih zavisi. Drugim rečima, rad se meri uglavnom na osnovu veličine predloženog sistema, uz korektivni uticaj više drugih karakteristika projekata, procesa, proizvoda ili korišćenih resursa.

Walston i Felix (1977) su razvili jedan od prvih modela ovog tipa, nalazeći da IBM-ovi podaci iz 60 projekata dovode do jednačine oblika:

$$E = 5.25S^{0.91}$$

Projekti, iz kojih su uzimani podaci, služili su za gradnju sistema čija je veličina bila u opsegu od 4 000 do 467 000 redova koda, napisanih na 28 različitih programskih jezika visokog nivoa na 66 računara, i koji predstavljaju uloženi rad od 12 do 11 758 meseci ljudskog rada. Veličina je iskazana u redovima koda, uključujući komentare koji ne premašuju 50 procenata ukupnog broja redova u programu.

Osnovna jednačina dopunjena je indeksom produktivnosti koji odražava 29 faktora koji mogu da utiču na produktivnost, prikazanih u tabeli 3.7. Imajmo u vidu da su ti faktori usko vezani za vrlo specifične tipove projekata, uključujući dve platforme: operativni računar i razvojni računar. Taj model odražava poseban razvojni stil organizacije IBM Federal Systems koja je obezbedila podatke.

Tabela 3.7 Faktori produktivnosti po modelu Walstona i Felixa

1.	Složenost kupčevog interfejsa	16.	Primene inspekcija projekta i koda
2.	Učešće korisnika u definisanju zahteva	17.	Primene razvoja „odozgo nadole"
3.	Izmene u dizajnu programa inicirane od strane korisnika	18.	Primene tima sa glavnim programerom
4.	Iskustvo naručioca u oblasti primene	19.	Ukupna složenost koda
5.	Opšte iskustvo osoblja	20.	Složenost obrade
6.	Procenat programera koji su učestvovali u izradi specifikacije funkcija	21.	Složenost toka programa
7.	Prethodno iskustvo sa računarom na kome će se softver koristiti	22.	Ukupna ograničenja vezana za dizajn programa
8.	Prethodno iskustvo sa programskim jezikom	23.	Ograničenja programa vezana za operativnu memoriju
9.	Prethodno iskustvo sa aplikacijama slične veličine i složenosti	24.	Ograničenja u vezi sa vremenom izvršenja programa
10.	Odnos srednjeg broja osoblja prema trajanju projekta	25.	Kôd za interaktivni rad ili rad u realnom vremenu ili za izvršavanje pod ekstremnim vremenskim ograničenjima
11.	Hardver se razvija kada i softver	26.	Procenat koda spremnog za isporuku
12.	Pristup razvojnom računaru moguć pod posebnim uslovima	27.	Kôd ne spada u matematičke aplikacije ili programe za formatiranje ulaza/izlaza
13.	Zabranjen pristup razvojnom računaru	28.	Broj klasa u bazi podataka na 1000 redova koda
14.	Potrebno poverljivo bezbednosno okruženje računara i najmanje 25% programa i podataka	29.	Broj stranica isporučene dokumentacije na 1000 redova koda
15.	Primena strukturnog programiranja		

Svakom od 29 faktora se dodeljuje težina 1 ako on uvećava produktivnost, 0 ako nema uticaja na produktivnost a -1 ako umanjuje produktivnost. Ponderisani zbir 29 faktora se zatim koristi za generisanje procene potrebnog napora pomoću osnovne jednačine.

Bailey i Basili (1981) su sugerisali tehniku modelovanja, nazvanu metamodel, za generisanje formule za procenu koja odražava osobine organizacije koja vrši razvoj. Oni su demonstrirali svoju tehniku pomoću baze podataka za 18 naučnih projekata pisanih na programskom jeziku Fortranu u agenciji NASA u centru Goddard Space Flight. Kao prvo, minimizovali su standardnu grešku procene i formulisali vrlo preciznu jednačinu:

$$E = 5.5 + 0.73S^{1.16}$$

Zatim su tu početnu procenu korigovali na osnovu odnosa grešaka. Ako je R odnos između stvarnog rada E i predviđenog rada E', onda se korekcija rada definiše kao

$$ER_{pod} = \begin{cases} (R-1), & \text{ako je } R \geq 1 \\ 1 - 1/R & \text{ako je } R < 1 \end{cases}$$

Zatim su početnu procenu rada *Epod* korigovali ovako:

$$E_{pod} = \begin{cases} (1+ER_{pod})E, & \text{ako je } R \geq 1 \\ E/(1+ER_{pod}) & \text{ako je } R < 1 \end{cases}$$

Na kraju, Bailey i Basili (1981) su uključili i druge faktore koji utiču na potrebni rad, i koji su prikazani u tabeli 3.8. Za svaku stavku u tabeli, projekat se boduje od 0 (nije prisutan) do 5 (vrlo bitan), zavisno od procene rukovodioca projekta. Prema tome, ukupan broj poena za METH može biti najviše 45, za CPLX najviše 35, a za EXP najviše 25. Njihov model opisuje proceduru, zasnovanu na višelinijskoj regresiji sa minimalnim srednjekvadratnim odstupanjem, za primenu dobijenog broja poena za dodatnu korekciju potrebnog rada.

Tabela 3.8 Bailey-Basilijevi modifikatori potrebnog rada

Totalna metodologija (METH)	*Kumulativna složenost* (CPLX)	*Kumulativno iskustvo* (EXP)
Dijagrami stabla	Složenost interfejsa naručioca	Kvalifikacije programera
Projektovanje „odozgo nadole"	Složenost primene	Iskustvo programera sa mašinom
Formalna dokumentacija	Složenost toka programa	Iskustvo programera sa jezikom
Timovi sa glavnim programerom	Unutrašnja složenost komunikacije	Iskustvo programera sa primenom
Formalna obuka	Složenost baze podataka	Iskustvo tima
Formalni plan testiranja	Složenost spoljašnje komunikacije	
Formalizam projektovanja	Izmene u dizajnu programa koje je inicirao naručilac	
Čitanje koda		
Katalog jedinica razvoja		

Jasno je da je jedan od problema sa modelima ovog tipa njihova zavisnost od veličine projekta kao ključne promenljive. Procene se obično zahtevaju u ranoj fazi, mnogo pre nego što je raspoloživa tačna informacija o veličini, a sigurno pre nego što je sistem preveden u redove koda. Stoga ti modeli jednostavno transliraju problem potrebnog u problem potrebne veličine. Boehmov konstrukcioni model troškova - COCOMO (*Constructive Cost Model*) registruje taj problem pa u poslednjoj verziji, COCOMO II, uključuje tri tehnike za procenu veličine.

Boehm (1981) je razvio originalni COCOMO model u 1970-im, koristeći opsežnu bazu podataka sa informacijama o projektima realizovanim u američkoj kompaniji koja se bavi izradom softvera za različite klijente, TRW. Posmatrajući razvoj softvera sa inženjerske i ekonomske tačke gledišta, Boehm koristi veličinu kao primarnu odrednicu troška, a zatim prilagođava inicijalnu procenu uz oslonac na više od deset „nosilaca troškova", uključujući osobine osoblja, projekta, proizvoda i razvojnog okruženja. Tokom 1990-ih, Boehm je ažurirao originalni COCOMO model i formulisao COCOMO II koji odražava mehanizme sazrevanja procesa razvoja softvera.

Procena zasnovana na COCOMO II odražava tri glavna stadijuma svakog razvojnog projekta. Dok je originalni COCOMO model koristio već isporučene redove izvornog koda kao ključni ulazni podatak, novi model priznaje da je broj redova koda nemoguće znati na početku razvojnog ciklusa. U stadijumu 1, u projektima se obično grade prototipovi u cilju razrešavanja visoko rizičnih pitanja koja obuhvataju korisnički interfejs, interakciju softvera i sistema, performanse ili tehničku zrelost. U ovom trenutku se malo zna o verovatnoj veličini finalnog proizvoda koji razmatramo. Stoga COCOMO II izražava veličinu u jedinicama koje autori nazivaju aplikativne tačke. Kao što ćemo da vidimo, ova tehnika iskazuje veličinu u formi generatora rada na visokom nivou apstrakcije, kao što su: broj ekrana, broj izveštaja i broj komponenti na jezicima treće generacije.

U stadijumu 2, prvi dizajn, odluka da se krene sa dizajnom je već doneta, ali projektanti moraju da istraže alternativne arhitekture i koncepte funkcionisanja. Ponovo nema dovoljno informacija neophodnih za fino granulirane procene rada i trajanja, ali je daleko više toga poznato nego u fazi 1. U fazi 2, COCOMO II koristi funkcionalne tačke kao meru veličine. Tehnika funkcionalnih tačaka, istražena detaljno u referenci IFPUG (1994a i b), za procenu koristi funkcionalnost koja je opisana u zahtevima, jer oni sadrže bogatije opise sistema nego aplikativne tačke.

Do prelaska u stadijum 3, postarhitektonski, razvoj je započeo, i poznato je mnogo više informacija. U ovom stadijumu određivanje veličine može da se izvede u zavisnosti od funkcionalnih tačaka ili redova koda, a mnogi faktori troškova mogu da se procene sa određenim stepenom lagodnosti.

COCOMO II takođe obuhvata modele višekratne upotrebljivosti, uzima u obzir održavanje i prekide (tj. izmene u zahtevima tokom vremena), i još ponešto. Kao i u slučaju originalnog COCOMO-a, model uključuje faktor troškova za korekciju inicijalnog predviđanja potrebnog rada. Ne postoje objavljeni podaci o preciznosti metodologije COCOMO II jer je ona još relativno nova.

Pogledaćemo COCOMO II malo detaljnije. Osnovni model je oblika

$$E = bS^c m(\boldsymbol{X})$$

gde se inicijalna procena bSc koja je zasnovana na veličini koriguje vektorom informacije o nosiocima troškova $m(\boldsymbol{X})$. Tabela 3.9 opisuje nosioce troškova u svakoj fazi, kao i upotrebu drugih modela za modifikovanje procene.

Tabela 3.9 Tri faze modela COCOMO II

Gledište modela	Stadijum 1: Kompozicija primene	Stadijum 2: Rani dizajn	Stadijum 3: Postarhitektonski
Veličina	Aplikativne tačke	Funkconalne tačke (FP) i jezik	FP i jezik ili redovi izvornog koda (SLOC)
Višekratna upotrebljivost	Podrazumeva se u modelu	Ekvivalent SLOC-a kao funkcija drugih promenljivih	Ekvivalent SLOC-a kao funkcija drugih promenljivih
Izmena zahteva	Podrazumeva se u modelu	% izmene izražen kao faktor troška	% izmene izražen kao faktor troška
Održavanje	Aplikativne tačke Promet izmena na nivou godine	Funkcija od ACT, razumevanje softvera, nepoznavanje	Funkcija od ACT, razumevanje softvera, nepoznavanje
Skala (c) u jednačini nominalnog rada	1.0	0,91 do 1,23 zavisno od predistorije, konformizma, rane arhitekture, rešavanje rizika, kohezije tima i SEI stepena zrelosti procesa	0,91 do 1,23 zavisno od predistorije, konformizma, rane arhitekture, rešavanje rizika, kohezije tima i SEI stepena zrelosti procesa
Proizvod kao nosilac troška	Nema	Složenost, zahtevana višekratna upotrebljivost	Pouzdanost, veličina baze podataka, potrebe za dokumentacijom, višekratna upotrebljivost i složenost proizvoda
Platforma kao nosilac troška	Nema	Poteškoće u vezi sa platformom	Ograničenja vremena izvršavanja, ograničenja operativne memorije, nestalnost virtualne mašine
Radna snaga kao nosilac troška	Nema	Sposobnost i iskustvo osoblja	Sposobnost analitičara, iskustvo sa primenama, sposobnost programera, iskustvo sa jezicima i alatima, kontinuitet osoblja
Projekat kao nosilac troška	Nema	Zahtevani razvojni raspored, razvojno okruženje	Upotreba softverskih alata, zahtevani raspored razvoja, razvoj na više lokacija

U stadijumu 1, aplikativne tačke služe za merenje veličine projekta. Ta mera veličine je proširenje pristupa objektnih tačaka i nju su sugerisali Kaufman i Kumar (1993), kao i podataka produktivnosti koje su objavili Banker, Kauffman i Kumar (1994). Za izračunavanje aplikativnih tačaka, prvo treba da prebrojimo ekrane, izveštaje i komponente realizovane na jeziku treće generacije, koji će biti uključeni u aplikaciju. Pretpostavlja se da su ti elementi definisani na standardni način kao deo integrisanog računarski podržanog okruženja za razvoj softvera.

Dalje, klasifikujemo svaki element aplikacije kao jednostavan, srednji ili težak. Tabela 3.10 sadrži smernice za tu klasifikaciju.

Tabela 3.10 Nivoi složenosti aplikativnih tačaka

Za ekrane				Za izveštaje			
	Broj i izvor tabela podataka				Broj i izvor tabela podataka		
Broj sadržanih pogleda	Ukupno <4 (<2 servera, <3 klijenta)	Ukupno <8 (2-3 servera, 3-5 klijenata)	Ukupno 8+ (>3 servera, >3 klijenata)	*Broj sadržanih sekcija*	Ukupno <4 (<2 servera, <3 klijenta)	Ukupno <8 (2-3 servera, 3-5 klijenata)	Ukupno 8+ (>3 servera, >3 klijenata)
< 3	Jednostavan	Jednostavan	Srednji	0 ili 1	Jednostavan	Jednostavan	Srednji
3 - 7	Jednostavan	Srednji	Težak	2 ili 3	Jednostavan	Srednji	Težak
8+	Srednji	Težak	Težak	4+	Srednji	Težak	Težak

Brojevi koji se koriste za jednostavne, srednje ili teške tačke aplikacije odražavaju težinu složenosti datu u tabeli 3.11. Težine odražavaju relativni napor zahtevan za implementaciju izveštaja ili ekrana na datom nivou složenosti.

Tabela 3.11 Težinski faktori složenosti aplikativnih poena

Tip elementa	Jednostavan	Srednji	Težak
Prikaz na ekranu	1	2	3
Izveštaj	2	5	8
3GL komponenta	-	-	10

Zatim se sabiraju ponderisani izveštaji i prikazi na ekranu, radi dobijanja jedne vrednosti aplikativne tačke. Ako će r procenata objekata biti iskorišćeno iz prethodnog projekta, broj poena nove aplikacije računa se kao

$$\text{poeni nove aplikacije} = (\text{aplikativni poeni}) \times (100 - r)/100$$

Da bi se ova brojna vrednost koristila za procenu rada, koristimo stopu produktivnosti kao korektivni faktor, dobijenu na osnovu iskustva i sposobnosti učesnika u razvoju, uz zrelost i sposobnost po metodu CASE. Na primer, ako su iskustvo i sposobnost učesnika u razvoju nisko ocenjeni, a CASE zrelost i sposobnost su takođe dobili nisku ocenu, onda tabela 3.12 pokazuje da je faktor produktivnosti 7, tako da je neophodan broj meseci ljudskog rada jednak novim aplikativnim poenima podeljenim sa 7. Kada je iskustvo učesnika u razvoju nisko, a CASE zrelost visoka, procena produktivnosti je srednja vrednost: 16. Slično tome, kada razvojni tim ima nivoe iskustva koji variraju, produktivnost se izračunava ponderisanjem srednjom vrednošću iskustva i sposobnosti.

U stadijumu 1, nosioci troškova se ne uzimaju u obzir. Međutim, u stadijumu 2, predviđeni rad, koji se izračunava na osnovu funkcijskih tačaka, koriguje se stepenom višekratne upotrebljivosti, izmenama zahteva i održavanjem. Skala (tj. vrednost c u jednačini

za rad) se u stadijumu 1 postavlja na 1,0. U stadijumu 2, skala je u opsegu od 0,91 do 1,23, zavisno od stepena novina u sistemu, konformnosti, rane arhitekture, rešavanja rizika, kohezije tima i zrelosti procesa.

Tabela 3.12 Izračunavanje produktivnosti

Iskustvo i sposobnost učesnika u razvoju	Vrlo nisko	Nisko	Nominalno	Visoko	Vrlo visoko
CASE zrelost i sposobnost	Vrlo nisko	Nisko	Nominalno	Visoko	Vrlo visoko
Faktor produktivnosti	4	7	13	25	50

Nosioci troškova u stadijumima 2 i 3 predstavljaju korektivne faktore koji se množe sa vrednošću rada, na osnovu vrednovanja projekta od „ekstra nisko" do „ekstra visoko", zavisno od karakteristika projekta. Na primer, za iskustvo razvojnog tima sa nekim tipom aplikacije smatra se da je:

- *ekstra nisko* ako je manje od 3 meseca;
- *vrlo nisko* ako je jednako ili veće od 3 ali manje od 5 meseci;
- *nisko* ako je jednako ili veće od 5 ali manje od 9 meseci;
- *nominalno* ako je jednako ili veće od 9 meseci, a manje od jedne godine;
- *visoko* ako je jednako ili veće od 1 godinu ali manje od 2 godine;
- *vrlo visoko* ako je jednako ili veće od 2 godine ali manje od 4 godine ;
- *ekstra visoko* ako je jednako ili veće od 4 godine.

Slično tome, sposobnost analitičara se meri na ordinalnoj procentualnoj skali. Na primer, ocena je „vrlo visoko", ako analitičar spada u devedeset procenata a „nominalno" u pedeset pet procenata. Shodno tome, COCOMO II mu dodeljuje faktor rada od 1,42 što predstavlja vrlo nizak, do 0,71 što je vrlo visok. Ti množioci odražavaju mesto gde analitičar sa vrlo niskom sposobnošću uvećava potrebni rad 1,42 puta u odnosu na nominalno potreban, tj. u poređenju sa prosečnim analitičarem, dok analitičaru sa visokom sposobnošću treba oko tri četvrtine rada prosečnog analitičara. Slično tome, tabela 3.13 navodi kategorije nosioca troška u slučaju upotrebe alata, a faktori se kreću u opsegu od 1,17 za nisku sposobnost do 0.78 za visoku sposobnost.

Tabela 3.13 Kategorije upotrebe alata

Kategorija	*Značenje*
Vrlo niska	Funkcije editora, unošenja koda, otklanjanje grešaka
Niska	Jednostavan direktni i pozadinski CASE, nizak stepen integracije
Nominalna	Osnovni alati vezani za životni ciklus, umeren stepen integracije
Visoka	Jaki, zreli alati vezani za životni ciklus, umeren stepen integracije
Vrlo visoka	Jaki, zreli, praktični alati vezani za životni ciklus, dobro integrisani sa procesima, metodima i ponovnim korišćenjem

Imajmo u vidu da je stadijum 2 u sklopu COCOMO II namenjen prvim fazama projektovanja. Skup nosilaca troškova u ovom stadijumu je uži nego skup koji se koristi u stadijumu 3, što je posledica manjeg stepen razumevanja parametara projekta u stadijumu 2.

U modelu COCOMO je moguće prilagoditi razne njegove komponente tako da odgovaraju karakteristikama određene organizacije. U modelu COCOMO II na raspolaganju stoje i alati pomoću kojih se proračuni prave na osnovu karakteristika projekta koje se zadaju kao ulazni podaci. Kasnije u ovom poglavlju, primenićemo COCOMO na primer iz informacionog sistema.

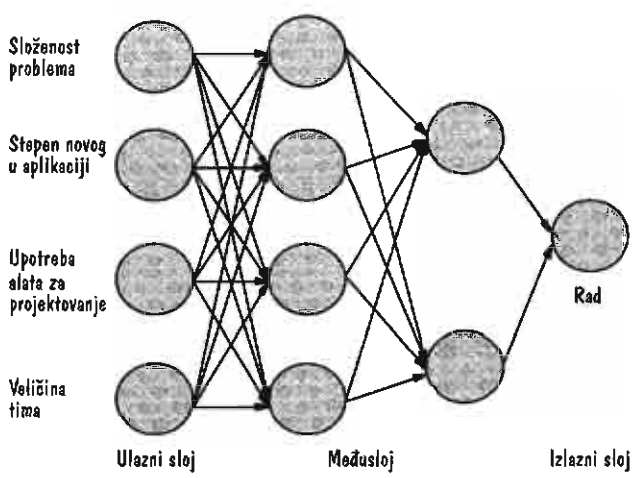
Metode mašinskog učenja

U prošlosti je većina tehnika modelovanja troškova i rada počivala na algoritamskim metodima. To jest, istraživači su ispitivali podatke iz ranijih projekata i na osnovu njih generisali jednačine, koje su dalje korišćene za predviđanje potrebnog rada i troškova za buduće projekte. Međutim, ima istraživača koji se bave mašinskim učenjem kao pomoćnim sredstvom za generisanje dobrih prognoza. Na primer, oslanjajući se na neuronske mreže moguće je predstaviti veći broj povezanih i zavisnih jedinica, tako da one predstavljaju sredstvo koje obećava kada se razmatra modelovanje različitih aktivnosti u procesu projektovanja softvera. U neuronskim mrežama, svaka jedinica (tj. neuron, predstavljen mrežnim čvorom) predstavlja jednu aktivnost. Svaka aktivnost ima svoje ulaze i izlaze. Svakoj jedinici mreže pridružuje se softver koji proračunava svoj ulaz, izračunavajući ponderisanu sumu. Ako suma premašuje vrednost zadatog praga, jedinica proizvodi izlaz. Taj izlaz dalje postaje ulaz za druge povezane jedinice u mreži, sve dok mreža ne proizvede konačnu vrednost. Neuronska mreža je, u svom značenju, proširenje grafa aktivnosti, koji smo ispitali ranije u ovom poglavlju.

Postoje mnogi načini kako neuronska mreža može da proizvede izlazne informacije. Neke tehnike posmatraju unazad ono što se dogodilo u drugim čvorovima. One se nazivaju tehnike *prostiranja unazad*. Slične su metodama koje koristimo kod analize grafova aktivnosti u procesu određivanja latentnog vremena putanje. Druge tehnike posmatraju unapred, u cilju predviđanja šta će se upravo dogoditi.

Neuronske mreže se razvijaju njihovim „obučavanjem" na osnovu podataka iz ranijih projekata. Relevantni podaci se ubacuju u mrežu, a sama mreža za „učenje" koristi algoritme koji idu unapred ili unazad, identifikujući šablone u podacima. Na primer, istorijski podaci o ranijim projektima mogu da sadrže informaciju o iskustvu članova razvojnog tima. Mreža može da identifikuje odnose između nivoa iskustva i količine rada neophodnog za dovršavanje projekta.

Slika 3.13 ilustruje kako Shepperd (1997) koristi neuronsku mrežu za izračunavanje potrebnog rada. Postoje tri sloja u mreži, pri čemu mreža nema cikluse. Četiri ulaza su faktori koji utiču na rad u okviru projekta. Mreža ih koristi za proizvođenje jednog izlaza koji predstavlja rad. Na početku, mreža se inicijalizuje slučajnim vrednostima težinskih faktora. Zatim se nove težine prosleđuju mreži. Te nove težine su izračunate pomoću „skupa za obučavanje", tj. skupa ulaznih i izlaznih vrednosti zasnovanih na istorijatu. Korisnik modela određuje algoritam obučavanja koji objašnjava kako se koriste podaci za obučavanje. Taj algoritam se takođe zasniva na istorijatu, i obično uključuje prostiranje unazad. Jednom kada je mreža prošla obuku (tj. kada su vrednosti mreže tako podešene da odražavaju prethodno iskustvo), ona može da se koristi za procenu potrebnog napora za nove projekte.



SLIKA 3.13 Shepperdova neuronska mreža sa prostiranjem unapred

Više istraživača je koristilo, na sličnim neuronskim mrežama, algoritme sa prostiranjem unazad za predviđanje rada, uključujući projekte koji koriste jezike četvrte generacije (Wittig i Finnie 1994; Srinivasan i Fisher 1995; Samson, Ellison i Dugard 1997). Shepperd (1997) smatra da je izgleda tačnost tog tipa modela suviše osetljiva na odluke o topologiji neuronske mreže, broj faza obučavanja, kao i inicijalne slučajne težine neurona u mreži. Izgleda da mreža, da bi dala dobru prognozu, mora da se obuči na velikom skupu. Drugim rečima, mreže moraju biti zasnovane na velikom iskustvu, umesto na nekoliko reprezentativnih projekata. Ponekad je teško da se dobiju podaci tog tipa, a posebno je teško neprestano ih prikupljati u velikim količinama, tako da odsustvo podataka ograničava upotrebljivost ove tehnike. Pored toga, korisnici često imaju problem da razumeju neuronske mreže. Međutim, ako ova tehnika proizvodi tačnije procene, organizacije će možda radije da sakupljaju podatke za njene potrebe.

U principu, ovaj pristup „učenja" isprobali su na različite načine i drugi istraživači. Srinivasan i Fisher (1995) koriste Kemererove podatke (Kemerer 1989) uz statističku tehniku koja se zove regresiono stablo. Oni su napravili tačnija predviđanja nego ona iz originalnog modela COCOMO i SLIM-a, modela koji predstavlja privatno tehnološko i komercijalno rešenje. Međutim, njihovi rezultati nisu bili tako dobri kao oni dobijeni pomoću neuronske mreže ili modela zasnovanog na funkcionalnim tačkama. Briand, Basili i Thomas (1992) su dobili bolje rezultate koristeći tehniku indukcionog stabla i Kermererove i COCOMO-ve skupove podataka. Porter i Selby (1990) su takođe koristili pristup zasnovan na stablu. Oni su konstruisali stablo odlučivanja koje identifikuje karakteristike projekta, procesa i proizvoda koje mogu biti korisne za predviđanje verovatnog napora. Oni su takođe koristili tu tehniku da bi predvideli koji moduli su verovatno bez nedostataka.

Tehnika mašinskog učenja, „rasuđivanje na osnovu slučaja", tj. CBR (*Case-Based Reasoning*), može da se primeni na predviđanja po analogiji. Ova tehnika se koristi u krugovima koji se bave veštačkom inteligencijom. CBR gradi algoritam odlučivanja zasnovan na više kombinacija ulaza koji mogu da se evidentiraju u projektu. Poput drugih ovde opisanih tehnika, CBR zahteva informacije o ranijim projektima. Shepperd (1997)

ukazuje da CBR nudi dve jasne prednosti u poređenju sa mnogim drugim tehnikama. Prvo, CBR se bavi samo događajima koji se stvarno dešavaju, umesto mnogo obimnijim skupom svih mogućih pojava. Ova odlika omogućava CBR-u da se bavi nedovoljno shvaćenim domenima. Drugo, korisnicima je lakše da razumeju pojedinačne slučajeve nego da opisuju događaje u vidu lanca pravila ili neuronskih mreža.

Procenjivanje pomoću tehnike CBR obuhvata četiri koraka:

1. Korisnik identifikuje novi problem kao slučaj.
2. Sistem pronalazi slične slučajeve u skladištu istorijskih informacija.
3. Sistem ponovo koristi znanje sadržano u ranijim slučajevima.
4. Sistem predlaže rešenje za novi slučaj.

To rešenje se može revidirati zavisno od stvarnih događaja, a rezultat se smešta u skladište, upotpunjujući kolekciju dovršenih slučajeva. Međutim, postoje dve velike prepreke prilikom izgradnje uspešnog CBR sistema: karakterizacija slučajeva i određivanje sličnosti.

Slučajevi se karakterišu na osnovu informacija koje su trenutno raspoložive. Obično se od stručnjaka traži da obezbede spisak odlika koje su značajne za opisivanje slučajeva, a posebno da bi se odredilo da li su dva slučaja slična. U praksi se sličnost meri pomoću n-dimenzionog vektora koji sadrži n odlika. Shepperd, Schofield i Kitchenham (1996) su utvrdili da je CBR pristup precizniji od tradicionalnih algoritamskih metoda zasnovanih na regresionoj analizi.

Pronalaženje modela za vašu situaciju

U današnje vreme u upotrebi su mnogi modeli za procenu rada i troškova. To su komercijalni alati zasnovani na prethodnom iskustvu ili komplikovanim modelima razvoja, kao i alati razvijeni u vlastitom aranžmanu koji pristupaju bazama istorijskih podataka o ranijim projektima. Potvrđivanje tih modela u praksi je teško, jer je za to neophodna velika količina podataka. Osim toga, ako model treba da se primeni na velik i raznolik skup situacija, baza podataka za podršku mora da uključuje merenja koja potiču iz mnoštva različitih razvojnih okruženja.

Čak i kada pronađete modele koji su projektovani za vaše razvojno okruženje, morate da procenite koji od modela daje najtačnije rezultate za konkretan projekat. Postoje dve statističke metode koje se često koriste za ocenu tačnosti, a to su PRED i MMRE. **PRED(x/100)** je procenat projekata kod kojih procena ulazi u x% stvarne vrednosti. Kod mnogih modela procenu rada, troškova i rokova, rukovodioci procenjuju PRED(0,25), tj. one modele koji ulaze u 25% stvarne vrednosti, a smatra se da model funkcioniše dobro ako je PRED(0,25) veći od 75%. Srednja veličina relativne greške MMRE (*Mean Magnitude Relative Error*) trebalo bi da za određeni model bude veoma mala. Neki istraživači smatraju da je MMRE od 0,25 prilično dobra vrednost, a Boehm (1981) sugeriše da MMRE treba da bude 0,10 ili manji. Tabela 3.14 navodi najbolje vrednosti za PRED i MMRE za razne modele objavljene u literaturi. Kao što možemo da vidimo, statistički podaci za većinu modela su razočaravajući, jer ukazuju da nijedan model nije obuhvatio suštinske karakteristike i njihove odnose za sve tipove razvoja. Međutim, odnosi između faktora troškova nisu jednostavni, a modeli moraju biti dovoljno fleksibilni da bi prihvatali stalne promene upotrebljenih alata i metoda.

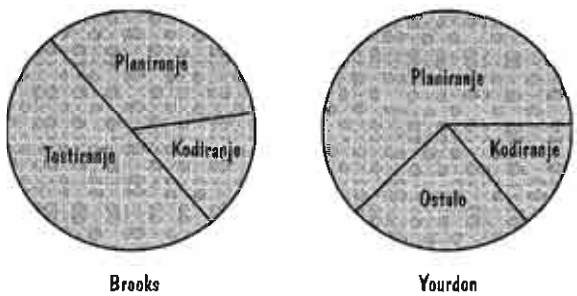
Tabela 3.14 Rezime performansi modela

Model	*PRED(0,25)*	*MMRE*
Walston-Felix	0,30	0,48
Osnovni COCOMO	0,27	0,60
Srednji COCOMO	0,63	0,22
Srednji COCOMO (varijacija)	0,76	0,19
Bailey-Basili	0,78	0,18
Pfleeger	0,50	0,29
SLIM	0,06 - 0,24	0,78 - 1,04
Jensen	0,06 - 0,33	0,70 - 1,01
COPMO	0,38 - 0,63	0,23 - 5,7
Opšti COPMO	0,78	0,25

Pored toga, Kitchenham, McDonnell, Pickard i Shepperd (2000) ukazuju da statističke metode MMRE i PRED nisu direktna mera tačnosti predviđanja. Oni predlažu da se jednostavne prave proporcije procenjenih i stvarnih vrednosti, tj. procenjeno/stvarno. Ta mera ima raspodelu koja direktno odražava tačnost procene. Nasuprot tome, MMRE i PRED predstavljaju meru raspodele (standardna devijacija), kao i koeficijent izduženosti (kurtozisa) te proporcije, tako da nam u stvari govore jedino o prirodi same raspodele.

Čak i kada modeli procenjivanja proizvode umereno tačne procene, moramo da razumemo koji tipovi rada su potrebni tokom razvoja. Na primer, dizajneri možda nisu potrebni sve dok analitičari zahteva ne završe rad na izradi specifikacije. U nekim modelima rada i troškova koriste se formule zasnovane na proteklom iskustvu radi raspodele rada u svim fazama životnog ciklusa softvera u modelu. Na primer, originalni COCOMO model predlaže rad neophodan za razvojne aktivnosti, na osnovu procenta dodeljenog ključnim aktivnostima u procesu. Međutim, kao što je prikazano na slici 3.14, izveštaji istraživača govore o konfliktu između vrednosti takvih predviđanja (Brooks 1975; Yourdon 1982). Prema tome, kada gradite sopstvenu bazu podataka za podršku procenjivanju u okviru vaše organizacije, važno je da upišete ne samo koliko je rada potrošeno na projekat, već i ko je to obavljao i u sklopu koje aktivnosti.

SLIKA 3.14 Različiti izveštaji o raspodeli rada



3.4 UPRAVLJANJE RIZICIMA

Kao što smo videli, mnogi rukovodioci softverskih projekata preduzimaju korake da bi osigurali da se njihovi projekti završe na vreme i u okviru ograničenja u pogledu rada i troškova. Međutim, upravljanje projektom podrazumeva daleko više praćenja rada i vremena. Rukovodioci moraju da odrede da li neki neželjeni događaj može da se desi tokom razvoja ili održavanja, pa stoga prave planove za izbegavanje tih događaja, ili, ako su oni neizbežni, minimizovanje negativnih posledica. Rukovodioci projekata moraju da učestvuju u **upravljanju rizicima** da bi razumeli rizike i njima upravljali u svojim projektima.

Šta je to rizik?

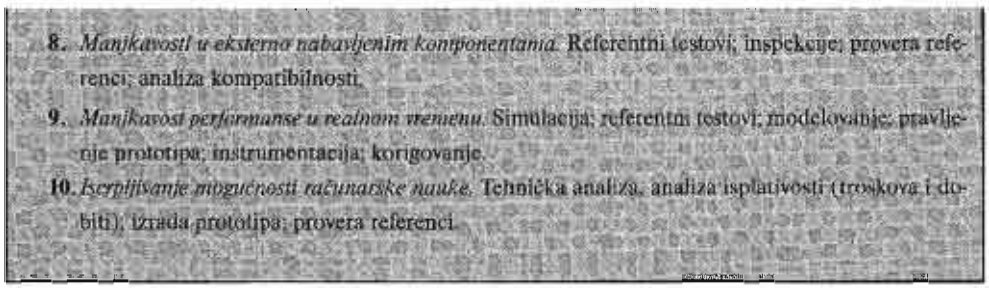
Mnogi događaji se dešavaju tokom razvoja softvera. Podsetnik 3.4 navodi Boehmov pogled na neke od najrizičnijih. Rizik razlikujemo od drugih događaja u projektu na osnovu tri pokazatelja (Rook 1993):

1. *Gubitak po događaju.* Određeni događaj mora da stvori situaciju u kojoj se projektu dešava nešto negativno: gubitak vremena, kvaliteta, novca, kontrole, razumevanja itd. Na primer, ako se zahtevi dramatično izmene nakon što je urađen projekat, onda sam projekat može da pati od gubitka kontrole i razumevanja, ako su novi zahtevi vezani za funkcije ili osobine koje nisu bliske projektnom timu. Takođe, radikalna promena u zahtevima verovatno dovodi do gubitka vremena i novca ako projektna rešenja nisu dovoljno fleksibilna da mogu brzo i lako da se izmene. Gubitak koji rizik nosi naziva se **uticaj rizika**.

PODSETNIK 3.4 BOEHMOVIH DESET NAJRIZIČNIJIH ČINILACA

Boehm (1991) je identifikovao 10 rizičnih činilaca, i predložio za njih odgovarajuće tehnike upravljanja rizicima.

1. *Manjak osoblja.* Kadrovske dopune najdarovitijim ljudima; davanje odgovarajućih poslova; izgrađivanje tima; izgrađivanje morala; unakrsna obuka; preraspoređivanje ključnih ljudi.
2. *Nerealni rokovi i budžet.* Detaljni trošak iz više izvora i predviđanje rokova; projektovanje prema trošku; inkrementalni razvoj; višekratna upotrebljivost softvera; pročišćavanje programskih zahteva.
3. *Razvoj pogrešnih softverskih funkcija.* Analiza organizacije; analiza misije; formulacija koncepta rada; ispitivanje korisnika; izrada prototipova; rana korisnička uputstva.
4. *Razvoj pogrešnog korisničkog interfejsa.* Izrada prototipa; izrada scenarija; analiza zadataka.
5. *Poslačivanje.* (dodavanje više u sistem nego što je neophodno). Pročišćavanje zahteva; izrada prototipa; analiza isplativosti (troškova i dobiti); projektovanje prema trošku.
6. *Neprekidni niz izmena u zahtevima.* Visok prag izmena; skrivanje informacija; inkrementalni razvoj (odlaganje izmena za kasnije inkremente).
7. *Manjkavost eksterno realizovanih poslova.* Provera referenci; provere pre dodeljivanja; ugovori bez naknade; nadmetanje u projektovanju ili izradi prototipa; građenje tima.



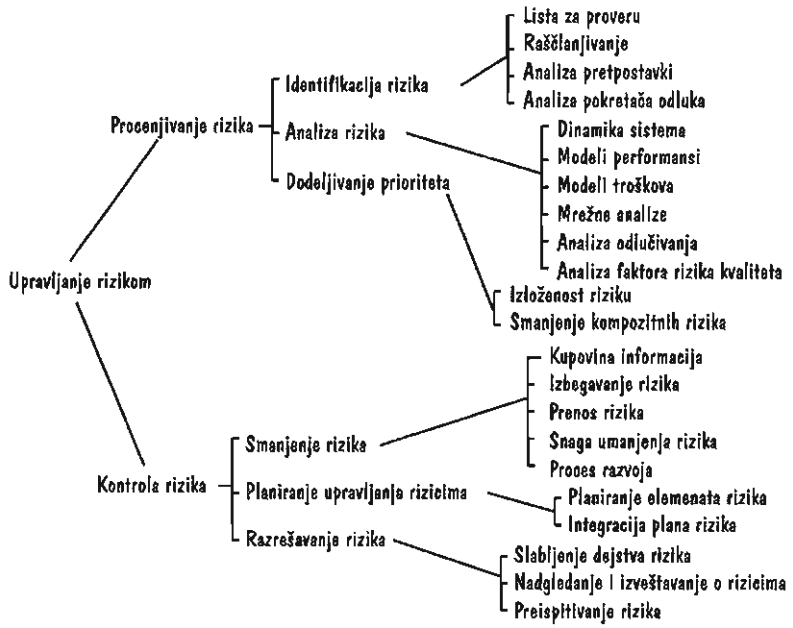
2. *Izvesnost pojave događaja.* Moramo imati bar neku predstavu o verovatnoći pojave razmatranog događaja. Na primer, pretpostavimo da se projekat razvija na jednoj mašini i da treba, nakon što se sistem potpuno istestira, da bude prenet na drugu. Ako je druga mašina potpuno novi model koji dobavljač tek treba da isporuči, nužno je proceniti verovatnoću da ona ne bude isporučena na vreme. Izvesnost pojave rizika, iskazana vrednošću od 0 (nemoguć događaj) do 1 (izvestan događaj), naziva se **verovatnoća rizika**. Kada je verovatnoća rizika jednaka 1, tada rizik predstavlja **problem**, jer će se sigurno dogoditi.
3. *Stepen do kog je moguće promeniti ishod.* Za svaki rizik, moramo odrediti šta je moguće učiniti na smanjenju ili izbegavanju njegovog uticaja. **Kontrola rizika** obuhvata skup radnji koje se preduzimaju u cilju redukovanja ili eliminisanja rizika. Na primer, ako zahtevi mogu da se izmene nakon dizajna, uticaj izmene je moguće smanjiti fleksibilnijim projektnim rešenjima. Ako druga mašina još nije spremna nakon testiranja softvera, možemo da pronađemo drugi model koji ima istu funkcionalnost i performanse i da na njemu izvršavamo softver do isporuke novog modela.

Efekte identifikovanih rizika možemo kvantifikovati množenjem uticaja rizika sa njegovom verovatnoćom, čime dobijamo tzv. **izloženost riziku**. Na primer, ako je verovatnoća da će doći do promene zahteva nakon dizajna jednaka 0,3, a trošak preprojektovanja po novim zahtevima iznosi 50 000 dolara, onda je izloženost riziku jednaka 15 000 dolara. Jasno je da da se verovatnoća rizika i njegov uticaj mogu promeniti tokom vremena tako da je jedan od zadataka rukovodioca projekta da te promene prati i u skladu sa tim usaglašava planove.

Postoje dva glavna izvora rizika: opšti rizici i rizici koji su specifični za razmatrani projekat. **Opšti rizici** predstavljaju rizike koji su zajednički za sve softverske projekte, kao što su recimo: pogrešno shvatanje zahteva, odlazak ključnih izvršilaca, ili nedovoljno vreme za testiranje. **Rizici specifični za projekat** predstavljaju pretnje koje su posledica slabih tačaka konkretnog projekta. Na primer, dobavljač je obećao da će isporučiti mrežni softver do određenog datuma, ali postoji rizik da do isporuke neće doći na vreme.

Aktivnosti upravljanja rizicima

Upravljanje rizicima obuhvata više važnih koraka, ilustrovanih na slici 3.15. Prvo procenjujemo rizik datog projekta, da bismo razumeli šta se sve može desiti tokom razvoja ili održavanja. Procenjivanje se sastoji od tri aktivnosti: identifikovanja rizika, analize rizika i dodeljivanje prioriteta svakom od njih. Za identifikaciju rizika možemo da koristimo više različitih tehnika.



SLIKA 3.15 Koraci u upravljanju rizicima (Rook 1993)

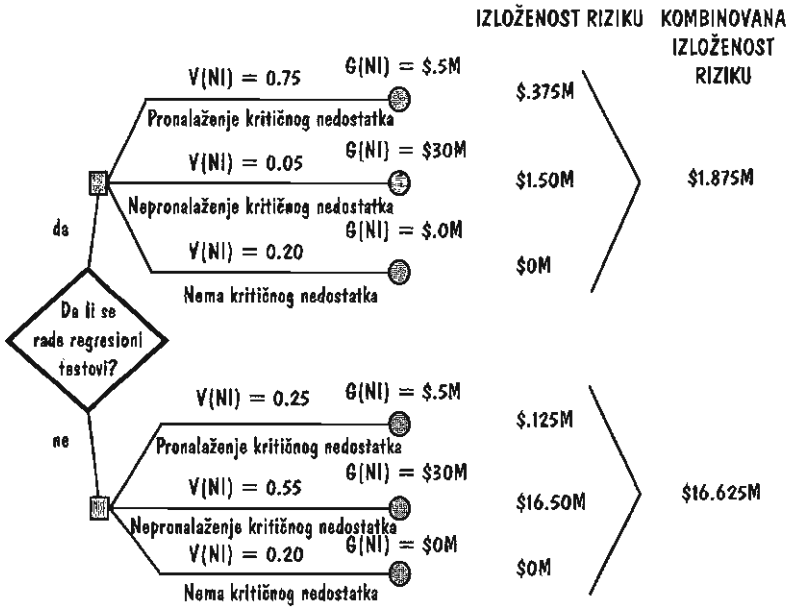
Ako je sistem koji gradimo na neki način sličan nekom sistemu koji smo pre toga gradili, možda već imamo spisak mogućih problema. Analizom tog spiska možemo utvrditi ima li izgleda da novi projekat bude predmet navedenih rizika. Ako se radi o sistemima koji, na neki način, predstavljaju nešto novo, spisak možemo proširiti analizom svake aktivnosti u ciklusu razvoja. Raščlanjivanje procesa na manje delove omogućiće predviđanje problema koji mogu da iskrsnu. Na primer, možemo da utvrdimo postojanje rizika da glavni projektant podnese otkaz u fazi dizajna. Slično tome, možemo da analiziramo pretpostavke ili odluke vezane za: način realizacije projekta, ko će raditi i sa kojim resursima. Zatim se svaka pretpostavka procenjuje da bi se odredili rizici koje sadrži.

Na kraju treba analizirati identifikovane rizike, sa ciljem da se shvati, način, uzrok i potencijalno mesto pojave rizičnih događaja, u najvećoj mogućoj meri. Postoji više tehnika koje se mogu koristiti za podizanje stepena spoznaje, uključujući modele dinamike sistema, modele troškova, modele performansi, mrežne analize itd.

Po identifikovanju svih rizika, dodeljujemo im prioritete na osnovu vlastitog shvatanja njihove prirode. Šema prioriteta omogućava nam da, inače ograničene resurse, usmerimo samo na rizike od kojih postoji najveća pretnja. Obično se dodela prioriteta zasniva na stepenu izloženosti riziku, koji ne uzima u obzir samo mogući uticaj, već takođe i verovatnoću da se rizični događaj desi.

Izloženost riziku se računa na osnovu verovatnoće pojave događaja koji nosi određeni rizik i posledica tog rizika, što znači da se oba ova aspekta rizika moraju proceniti. Da bismo videli kako se vrši kvantifikacija, razmotrimo analizu predstavljenu na slici 3.16. Pretpostavimo da smo analizirali proces razvoja sistema, i da su rokovi isporuke kratki. Sistem se realizuje kroz niz izdanja, gde svako sledeće sadrži više funkcija nego prethodno. Kako je sistem projektovan tako da su funkcije relativno nezavisne, razmatra se samo testiranje dodatnih funkcija u novom izdanju, uz pretpostavku da postojeće

funkcije još uvek rade kao i pre. Prema tome, možemo, na primer, zaključiti da postoji rizik vezan za odsustvo **regresionih testova**, koji obezbeđuju potvrdu očuvanja prethodne funkcionalnosti.



SLIKA 3.16 Primer izračunavanja izloženosti riziku

Za svaki mogući ishod procenjuju se dve veličine: verovatnoća neželjenog ishoda V(NI) i gubitak koji on prouzrokuje, G(NI). Na primer, postoje tri moguće posledice izvođenja regresionih testova: pronalaženje kritičnog nedostatka, ako takav postoji, nepronalaženje kritičnog nedostatka (čak iako takav postoji), ili utvrđivanje (ispravno) da kritičan nedostatak ne postoji. Kao što je ilustrovano na slici, procenili smo da je verovatnoća prvog slučaja jednaka 0.75, verovatnoća drugog 0,05, a trećeg 0,20. Gubitak koji neželjeni ishod donosi se procenjuje na 500 000 dolara, ako je kritični nedostatak pronađen pa je izloženost riziku jednaka 375 000 dolara. Slično tome, računamo izloženost riziku za druge grane stabla odlučivanja, pri čemu pronalazimo da je izloženost riziku kod sprovođenja regresionih testova gotovo 2 miliona dolara. Međutim, identične analize pokazuju da je izloženost riziku skoro 17 miliona dolara ako se regresiono testiranje ne sprovede. Prema tome, možemo slobodno konstatovati da je rizik veći ako ne sprovodimo regresione testove.

Izloženost riziku nam pomaže da navedemo rizike prema redosledu prioriteta, gde je riziku koji nas najviše zabrinjava dodeljen najviši prioritet. Dalje, moramo da preduzmemo korake za kontrolu rizika. Korišćenjem termina kontrola priznajemo da možda nećemo moći eliminisati sve rizike. Ali zato rizik možemo svesti na najmanju meru, ili ga ublažiti preduzimajući blagovremeno mere za upravljanje događajima sa neželjenim ishodom na odgovarajući način. Prema tome, kontrola rizika obuhvata umanjenje rizika, planiranje rizika i razrešavanje rizika.

Postoje tri strategije za umanjenje rizika:

- izbegavanje rizika, izmenom zahteva u vezi sa performansom ili funkcionalnošću;

- prenošenje rizika, dodeljivanjem rizika drugim sistemima ili ugovaranjem osiguranja radi pokrivanja finansijskog gubitka u slučaju pojave rizika;
- podrazumevanje rizika, njegovim prihvatanjem i kontrolom u okviru resursa projekta.

Da bismo doprineli donošenju odluka koje utiču na smanjenja rizika, moramo da uzmemo u obzir trošak koji te odluke proizvode. Pod pojmom **sprega rizika** podrazumevamo količnik razlike izloženosti riziku pre i posle umanjenja rizika i troška koje to umanjenje nosi. Drugim rečima, sprega umanjenja rizika je:

(Izloženost riziku pre umanjenja - izloženost riziku posle umanjenja)
/(trošak umanjenja rizika)

Ako vrednost sprege nije dovoljna da opravda akciju, tada možemo potražiti neke druge, manje skupe ili efikasnije tehnike, za umanjenje rizika.

U nekim slučajevima, možemo da odaberemo proces razvoja koji doprinosi smanjenju rizika. Na primer, videli smo u poglavlju 2 da izrada prototipa može da poboljša razumevanje zahteva i projektnih rešenja, tako da odabiranje procesa izrade prototipa može da umanji mnoge rizike na projektu.

Korisno je evidentirati donete odluke u sklopu **plana za upravljanje rizicima**, tako da i naručilac i razvojni tim mogu imati uvid u načine za izbegavanje problema, kao i postupke u slučaju njihove pojave. Pored toga, neophodno je nadgledati projekat, uz periodičnu ponovnu procenu rizika, verovatnoće pojavljivanja i mogućeg uticaja.

3.5 PLAN PROJEKTA

Da bismo naručiocima preneli rezultate analize rizika i predložili upravljanje njima, predviđene troškove u vezi s projektom, rokove i organizaciju, obično praivmo **plan projekta**. Plan u pisanom obliku sadrži korisnikove potrebe/zahteve i naše namere vezane za zadovoljenje tih potreba/zahteva. Naručilac analizira plan da bi dobio informacije o aktivnostima tokom razvojnog procesa, što mu omogućuje da lakše prati napredak projekta. Plan takođe može biti osnov za potvrdu učinjenih pretpostavki, posebno onih vezanih za troškove i rokove.

Dobar plan projekta mora da ima sledeće elemente:

1. obim projekta;
2. rokove projekta;
3. organizaciju projektnog tima;
4. tehnički opis predloženog sistema;
5. standarde koji se u projektu koriste, postupke, kao i predložene tehnike i alate;
6. plan osiguranja kvaliteta;
7. plan upravljanja konfiguracijom;
8. plan dokumentovanja;
9. plan upravljanja podacima;
10. plan upravljanja resursima;
11. plan testiranja;
12. plan obuke;

13. plan bezbednosti;
14. plan upravljanja rizicima;
15. plan održavanja.

Obim definiše granice sistema, odnosno definiše šta će biti obuhvaćeno sistemom, a šta neće. On garantuje naručiocu da smo razumeli šta se od projekta zahteva. Rokovi se mogu izraziti preko strukture posla, miljokaza i vremenskih rokova, da bi se prikazalo šta će se događati u svakoj tački tokom životnog ciklusa projekta. Gantogram je koristan za ilustrovanje paralelne prirode nekih zadataka u procesu razvoja.

Plan projekta takođe navodi članove razvojnog tima, kako su oni organizovani i čime će se baviti. Kao što smo ranije videli, nisu svi izvršioci potrebni sve vreme trajanja projekta, tako da se u planu projekta obično vidi kako su resursi raspoređeni, odakle se vidi i angažovanje izvršilaca u različitim vremenskim intervalima.

Izrada tehničkih opisa primorava nas da formulišemo odgovore i razmotrimo probleme dok predviđamo dalji kurs projekta. Tehnički opis sadrži hardver i softver, uključujući prevodioce, interfejse, kao i opremu ili softver posebne namene. U planu projekta dokumentuju se sva posebna ograničenja vezana za kabliranje, vremena izvršavanja, vremena odziva, bezbednost ili druge aspekte funkcionalnosti ili performansi. Plan takođe navodi sve standarde i metode koji se moraju koristiti, kao npr.:

- algoritmi;
- alati;
- tehnike pregleda ili inspekcije;
- jezici za dizajn ili predstavljanje;
- jezici za kodiranje;
- tehnike testiranja.

Za velike projekte je možda pogodnije postojanje posebnog plana za obezbeđenje kvaliteta, u kome se opisuje kako pregledi, inspekcije, testiranje i druge tehnike pomažu pri proceni kvaliteta i obezbeđenju da on odgovara potrebama naručioca. Slično tome, kod velikih projekata je potreban plan upravljanja konfiguracijom, posebno ako će postojati više verzija i izdanja datog sistema. Kao što ćemo videti u poglavlju 10, upravljanje konfiguracijom pomaže uspostavljanju bolje kontrole nad više primeraka softvera. Plan upravljanja konfiguracijom ukazuje naručiocu na način praćenja izmena u zahtevima, dizajnu, kodu, planovima testiranja i dokumentaciji.

Tokom razvoja nastaje više dokumenata, posebno u slučaju velikih projekata kod kojih informacija o projektnim rešenjima mora da bude dostupna svim članovima projektnog tima. Plan projekta navodi dokumente koji treba da se izrade, objašnjava ko će ih i kada izraditi, a takođe, u saglasnosti sa planom upravljanja konfiguracijom, opisuje način izmene dokumenata.

Svaki softverski sistem ima ulazne podatke, proračune i izlazne podatke, pa stoga plan projekta mora da objasni način prikupljanja, skladištenja, manipulacije i arhiviranja tih podataka. Plan takođe treba da predstavi način korišćenja resursa. Na primer, ako konfiguracija hardvera obuhvata i izmenjive diskove, tada deo plana projekta, koji se odnosi na upravljanje resursima, treba da sadrži objašnjenja o tome koji se podaci nalaze na svakom disku, kako se vrši dodeljivanje grupa diskova ili disketa i kako se prave rezervne kopije sadržaja tih medijuma.

Da bi testiranje bilo efikasno, neophodno je da bude dobro isplanirano. Stoga plan projekta teba da sadrži sveobuhvatan pristup testiranju u sklopu projekta. Plan posebno treba da sadrži način kako će se generisati podaci za test, testiranje pojedinačnih programskih modula (na primer, testiranje svih putanja ili svih iskaza), integrisanje pojedinačnih modula i testiranje celina, testiranje sistema i poimence navedene izvršioce zadužene za sprovođenje određenih vrsta testova. Ponekad se sistemi realizuju stadijumski ili po fazama tako da je u planu testiranja neophodno specifikovati način testiranja svakog stadijuma. Kada se nova funkcionalnost fazno uvodi u sistem, kao što smo videli u poglavlju 2, tada plan testiranja mora da uključi i regresiono testiranje, koje obezbeđuje potvrdu da su očuvane postojeće funkcionalnosti.

Da bi proces obuke počeo neposredno po isporuci sistema, a nekada i pre, obično se kursevi za obuku i prateća dokumentacija pripremaju u fazi razvoja, umesto nakon isporuke sistema. Plan projekta definiše način sprovođenja obuke, sadržaj kurseva, pomoćni softver i dokumentaciju, kao i potrebna predznanja svakog polaznika.

U slučaju da sistem postavlja zahteve vezane za bezbednost, ponekad je neophodan i poseban bezbednosni plan. Njime se specifikuje način zaštite podataka, korisnika i hardvera. Bezbednost obuhvata poverljivost, raspoloživost i integritet, te stoga plan mora da objasni kako će svaki od ovih segmenata uticati na razvoj sistema. Na primer, ako će pristup sistemu biti ograničen pomoću lozinki, onda plan mora da opiše ko izdaje i održava lozinke a ko razvija softver za rukovanje lozinkama i koji način za šifriranje lozinki će se koristiti.

Konačno, ako će projektni tim da održava sistem i posle isporuke, plan projekta treba da razmotri i odgovornosti za izmene koda, popravke hardvera i ažuriranje prateće dokumentacije i materijala za obuku.

3.6 MODELI PROCESA I UPRAVLJANJE PROJEKTOM

Do sada smo videli kako različiti aspekti projekta mogu da utiču na potreban rad, troškove i vremenske rokove, kao i na sadržane rizike. Najuspešniji rukovodioci projektima, koji realizuju kvalitetne proizvode na vreme i u okviru zadatog budžeta, jesu oni koji tehnike rukovođenja projektom prilagođavaju posebnim karakteristikama neophodnih resursa, odabranom procesu i dodeljenim ljudima.

Da bismo razumeli šta da radimo u okviru sledećeg projekta, korisno je da ispitamo tehnike rukovođenja projektom korišćene u uspešnim projektima iz bliske prošlosti. U ovom odeljku razmotrićemo dva projekta: Digitalov Alpha AXP program i softver aviona F-16. Takođe ćemo da istražimo objedinjavanje upravljanja procesima i projektom.

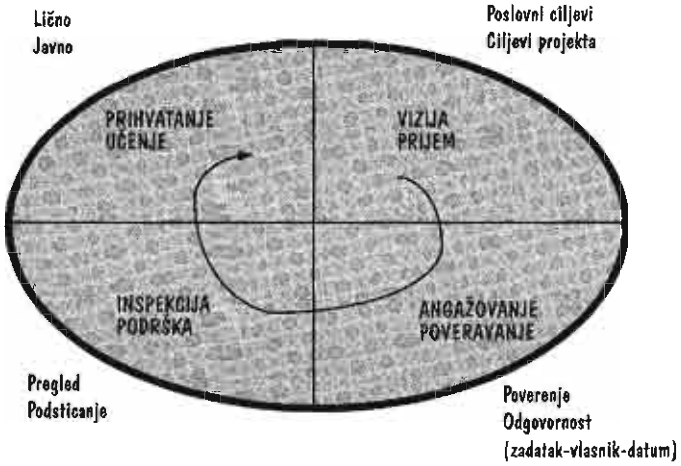
Upravljanje u paketima

Kompanija Digital Equipment Corporation utrošila je više godina na razvoj sistema Alpha AXP, a nova arhitektura sistema i prateći proizvodi učinili su od njega najveći projekat u Digitalovoj istoriji. Softverski deo je obuhvatao izradu četiri operativna sistema i 22 grupe za inženjere softvera, koji su radili na projektovanju alata za migraciju, mrežnih sistema, prevodioca, baze podataka, okruženja za integraciju, kao i same aplikacije. Za razliku od drugih razvojnih projekata, glavni problemi sa projektom Alpha bio je prerano dostizanje miljokaza! Prema tome, poučno je da vidimo kako je projektom upravljano i kakav je bio uticaj procesa upravljanja na finalni proizvod.

Tokom razvoja, rukovodioci projekata razvili su model, koji je inkorporisao četiri principa, i koji je nazvan Enrollment Management („prijemno upravljanje"):

1. uspostavljanje dovoljno široke zajedničke vizije;
2. potpuno dodeljivanje zadataka i dobijanje posebnih saglasnosti učesnika;
3. energično vršenje inspekcija i obezbeđivanje povratnih informacija;
4. potvrdu svakog unapređenja i učenje tokom napretka programa (Conklin 1996).

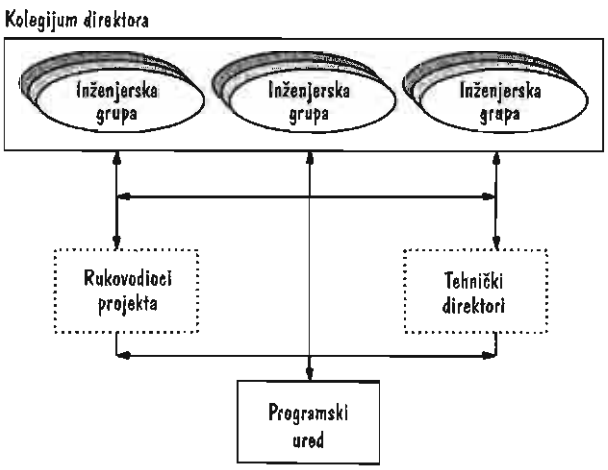
Slika 3.17 ilustruje dati model. Vizija je korišćena za „prijem" odgovarajućih programa tako da svi dele zajedničke ciljeve. Svaka grupa ili podgrupa projekta definisala je sopstvene ciljeve zavisno od globalnih ciljeva navedenih za projekat, uključujući poslovne ciljeve kompanije. Dalje, kako su rukovodioci razvijali planove, oni su poveravali zadatke grupama, tražeći od njih da se angažuju prilikom formulisanja zadataka i da komentarišu i zadatke i namentuta ograničenja u vezi sa rokovima. Svaki zahtevani rezultat bio je merljiv i identifikovan sa određenim vlasnikom, osobom koja je smatrana odgovornom za njegovo sprovođenje. Vlasnik nije nužno bio osoba koja je obavljala posao, već je predstavljao osobu odgovornu da se posao uradi.



SLIKA 3.17 Model prijemnog upravljanja (Conklin 1996)

Rukovodioci su neprestano vršili inspekciju projekta da bi proverili da li će isporuka biti na vreme. Od članova projektnog tima je traženo da identifikuju rizike, a u slučaju da je postojala opasnost da rizik spreči da tim postigne svoje ciljeve, rukovodilac projekta bi proglašavao da je projekat došao na „prekretnicu", tj. da je nastupilo ono što se smatra kritičnim događajem. Takva objava je značila da članovi tima prihvataju da naprave suštinske promene u cilju obezbeđenja nastavka projekta. Za svaki stadijum u okviru projekta, rukovodioci su lično i javno odavali priznanja vezana za napredak. Evidentirali su naučene lekcije i od članova tima tražili sugestije o tome kako stvari mogu da se unaprede u budućnosti.

Koordinisanje svih hardverskih i softverskih grupa bilo je teško, a rukovodioci su shvatili da treba da nadgledaju kako tehničke događaje, tako i događaje u projektu. To jest, tehničko gledište je obuhvatalo tehnička projektna rešenja i strategiju, dok je projektno gledište uključivalo angažovanje i predmete isporuke. Slika 3.18 ilustruje organizaciju koja je omogućila da oba pogleda daju doprinos ukupnom programu.



SLIKA 3.18 Organizacija projekta Alpha (Conklin 1996)

Jednostavnost modela i organizacije ne znači da je upravljanje programom Alpha bilo jednostavno. Projekat se nekoliko puta nalazio na prekretnicama koje su razrešene na različite načine. Na primer, rukovodstvo nije uspelo da izradi globalni plan tako da su rukovodioci pojedinačnih projekata imali poteškoća sa usaglašavanjem. U isto vreme, tehničko rukovodstvo je generisalo neprihvatljivo obimnu i nerazumljivu projektnu dokumentaciju. Da bi imali kontrolu nad situacijom, rukovodiocima programa Alpha bio je neophodan plan rada na nivou celokupnog programa koji bi ilustrovao redosled kojim svaki sadržani zadatak treba da se uradi, kao i način njegove koordinacije sa drugim zadacima. Oni su međutim izradili glavni plan koji je jedino razmatrao kritične komponente programa, tj. one koje su bile kritične sa aspekta poslovnog uspeha. Plan je bio ograničen na samo jednu stranicu teksta, tako da su svi učesnici mogli da vide globalnu sliku, bez složenosti ili detalja. Slično tome, projektna rešenja, rokovi i drugi ključni elementi opisani sa po jednom stranicom teksta, omogućili su učesnicima u projektu da steknu globalnu sliku o tome šta da rade, kada i kako to da urade.

Sledeća prekretnica se dogodila kada je objavljeno da realizacija kritičnog zadatka kasni nekoliko meseci. Rukovodstvo je rešilo taj problem uvođenjem redovnih operacionih inspekcija napretka, sa ciljem da se preduprede dalja iznenađenja. Inspekcija je podrazumevala podnošenje izveštaja čiji obim takođe nije prelazio jednu stranicu teksta, a koji je sadržao obaveštenje o sledećim tačkama od ključnog značaja za projekat:

- vremenske rokove;
- miljokaze;
- događaje na kritičnoj putanji u proteklom mesecu;
- aktivnosti duž kritične putanje u sledećem mesecu;
- razrešena otvorena pitanja i posledice;
- nerazrešena otvorena pitanja i posledice (sa naznakom vlasnika i rokova).

Za uspeh programa Alpha bilo je važno to što su rukovodioci shvatili da su inženjeri obično motivisaniji priznanjima nego finansijskom dobiti. Umesto da učesnike nagrađuju novcem, oni su se usredsredili na objavljivanje napretka i na obezbeđenje da javnost sazna koliko rukovodioci cene inženjerski rad.

Rezultat fleksibilnog i fokusiranog upravljanja projektom Alpha bio je program koji je zadovoljio rokove „u mesec", uprkos usputnim odstupanjima. Prijemno upravljanje omogućilo je malim grupama da već na početku prepoznaju probleme koje bi mogli imati i da preduzmu korake za njihovo razrešenje dok su još mali i lokalizovani. Stalnost namera kombinovana sa neprestanim učenjem dovela je do izuzetnog proizvoda. Program Alpha je dostigao ciljeve u pogledu performansi, a njegov kvalitet je vrlo visoko ocenjen.

Model nosilaca odgvornosti

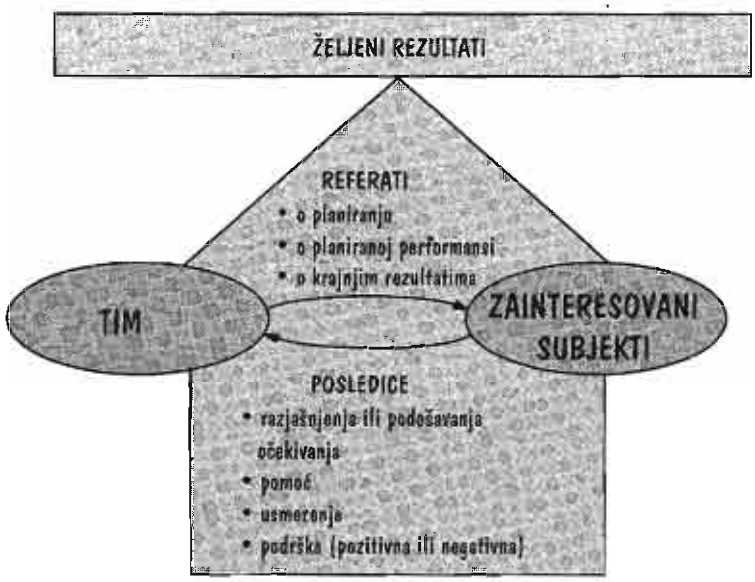
Vazduhoplovne snage SAD i Lockheed Martin su formirali razvojni tim (*Integrated Product Development Team*) radi izrade modularnog softverskog sistema za borbeni avion F-16, projektovan tako da bude kapacitetniji, obezbedi potrebne funkcije i smanji troškove i vreme naknadnih izmena. Rezultujući softver iznosio je više od četiri miliona redova koda, od čega je četvrtina implementirala ograničenja u toku leta. Razvoj softvera za F-16 takođe je uključivao i: izradu rukovaoca uređajima; proširenja izvršnog sistema programskog jezika Ada elementima rada u realnom vremenu; mrežu radnih stanica za softversko inženjerstvo, Adainog prevodioca za modularni namenski računar, alate za izgradnju softvera i rukovanje konfiguracijom, softver za simulaciju i testiranje i interfejse za učitavanje softvera u avion (Parris 1996).

Zahtevi u vezi sa osobinama softvera za podršku letu bili su vrlo dobro shvaćeni i stabilni, bez obzira što je 250 članova razvojnog tima, organizovanih u osam radnih timova sa glavnim inženjerom, rukovodiocem programa i pomoćnim osobljem, čekalo da napišu milione redova koda. Međutim, uobičajene stvari je trebalo primeniti na neuobičajen način: izraditi modularni softver oslanjajući se na programski jezik Ada, uraditi objektno orijentisanu analizu i dizajn, i preći sa velikog računara na radne stanice. Ograničenja u vezi sa upravljanjem projektom obuhvatala su striktno određene rokove i razvoj tri, takođe obimne, verzije koje su nazvali „trake". Pristup je bio visoko rizičan, jer je prva traka ostavljala malo vremena potrebnog za učenje novih metoda i alata koji su se ticali konkurentnog razvoja (Parris 1996).

Pritisak na projekat se uvećao kresanjem budžeta dok su krajnji rokovi smatrani krajnje nerealnim. Pored toga, projekat je bio organizovan na način koji nije bio blizak većini inženjera. Učesnici su bili naviknuti na rad u **matričnoj organizaciji**, gde je svaki inženjer pripadao funkcionalnoj jedinici na osnovu sposobnosti (kao što je projektantska grupa ili grupa za testiranje) i dodeljivan jednom projektu ili većem broju projekata, kako se javljala potreba za određenim stručnim profilom. Drugim rečima, radnici su se poistovetili sa svojim mestom u matrici, gde su funkcionalne veštine bile jedna, a nazivi projekata druga dimenzija. U toj tradicionalnoj organizaciji, odluke su se donosile u sklopu hijerarhije funkcionalnih jedinica. Međutim, ugovor za F-16 je zahtevao da projekat bude organizovan u formi **integrisanog razvojnog tima**. To je postignuto kombinovanjem pojedinaca iz različitih funkcionalnih grupa u jednu interdisciplinarnu radnu jedinicu potpomognutu odvojenim kanalima odgovornosti.

Da bi omogućili članovima projekta da prihvate kulturne promene koje je donosila nova organizacija, za projekat F-16 korišćen je model nosilaca odgovornosti prikazan na slici 3.19. U tom modelu, tim je grupa ljudi odgovornih za postizanje datog rezultata. Zainteresovani subjekt je svako na koga taj rezultat, ili način na koji je on ostvaren, utiče. Taj proces uključuje neprestano izveštavanje iz domena svoje odgovornosti (izveštaja o tome šta ste uradili, šta radite i šta planirate da radite) i saopštavanje posledica, sa ciljem da se radi samo ono što i tim i pojedinac smatraju razumnim. Model je bio primenjen na izgradnju upravljačkog

mehanizama i operativnih procedura u sklopu tima, zamenjujući nezavisna ponašanja zavisnim, naglašavajući da treba „biti dobar a ne izgledati dobro" (Parris 1996).

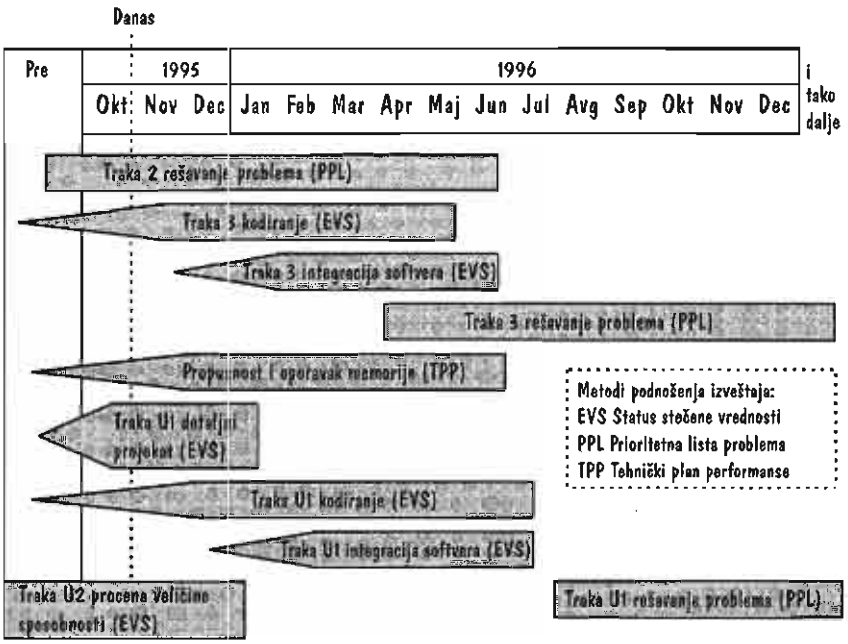


SLIKA 3.19 Model nosilaca odgovornosti (Parris 1996)

Posledica je bila u tome da se javila potreba za uvođenjem novih poslovnih običaja, na primer nedeljnog jednosatnog pregleda statusa tima. Da bi se naglasio značaj ponete odgovornosti i referisanja, svaka pojedinačna aktivnost je imala izričit kriterijum okončanja i bila predmet praćenja do završetka. Aktivnost se mogla dodeliti članu tima, odnosno, zainteresovanom subjektu, a često je uključivala razjašnjenje otvorenih pitanja ili zahteva, obezbeđujući informacije koje su nedostajale ili razrešenje sukoba.

Kako je tim realizovao više konkurentnih aktivnosti, korišćena je mapa aktivnosti za ilustrovanje napretka svake od njih u sveopštem kontekstu projekta. Slika 3.20 prikazuje deo mape aktivnosti. Svaka pruga predstavlja aktivnost i svakoj aktivnosti je pripisan metod izveštavanja o njenom napretku. Tačka na prugi znači da je u tom trenutku završen detaljan plan aktivnosti. Linija „danas" prikazuje trenutni status, a mapa aktivnosti se koristila i tokom nedeljnih sastanaka da bi pružila uvid u napredak koji je predmet razmatranja.

Za svaku aktivnost, napredak je praćen pomoću odgovarajućeg metoda procenjivanja ili performanse. Nekada je metod obuhvatao procenu troškova, analizu kritične putanje, ili praćenje rokova. *Stečena vrednost* je korišćena kao zajednička mera za poređenje napretka različitih aktivnosti, tj. obrazac za poređenje aktivnosti utvrđivao je koliki je deo projekta kompletiran okončanjem određene aktivnosti. Računanje stečene vrednosti uključivalo je pondere koji su predstavljali procenat ukupnog procesa koji nosi svaki korak, prema ukupnom naporu. Slično tome, svakoj komponenti bila je dodeljena vrednost veličine, koja je predstavljala njenu srazmeru u odnosu na celokupni projekat, tako da je mogao da se prati i relativni napredak u odnosu na krajnju veličinu. Zatim je na svakom kontrolnom sastanku bio predstavljan grafikon kumulativne stečene vrednosti, slično kao na slici 3.21.

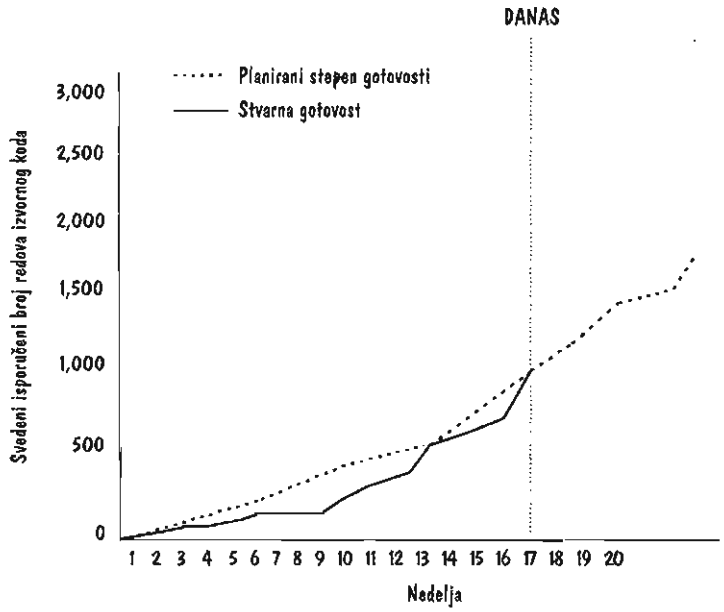


SLIKA 3.20 Jednostavan plan aktivnosti (preuzeto iz Parris 1996)

Za svaki završeni deo proizvoda napredak se više nije pratio. Tada su se pratile samo njegove performanse i evidentirani su problemi. Zainteresovani subjekti su svakom problemu dodeljivali prioritet, a na nedeljnim kontrolnim sastancima predstavljan je pregled od po pet najznačajnijih problema od svakog proizvodnog tima i o tim problemima se raspravljalo. Pokušavali su da se sagledaju razlozi nastanka problema, moguće akcije za njihovo prevazilaženje i načini prevencije sličnih problema u budućnosti.

Rukovodioci projekata su konstatovali da je najznačajniji problem kod modela nosilaca odgovornosti činjenica da on nije ništa govorio o koordinaciji između različitih timova. Zato su izradili softver za evidentiranje i praćenje elemenata primopredaje između timova, tako da je svaki tim mogao da zna koji timovi čekaju na njih da završe neki posao ili neki proizvod. Jedan model primopredaje korišćen je i u planiranju, čime su eliminisani neželjeni obrasci ili scenariji. Analiza modela primopredaje postala je sastavni deo nedeljnih sastanaka.

Lako je uočiti kako se model nosilaca odgovornosti u sprezi sa modelom primopredaje, bavi različitim aspektima upravljanja projektom. Prvo, obezbeđen je mehanizam za komunikaciju i koordinaciju. Drugo, podstaknuto je upravljanje rizicima, posebno primoravanjem članova tima da ispituju probleme na kontrolnim sastancima. I treće, došlo je do integrisanja izveštavanja o napretku i procesa rešavanja problema. Prema tome, taj model u suštini opisuje proces upravljanja sproveden u okviru projekta F-16.



SLIKA 3.21 Primer grafikona kumulativne stečene vrednosti (Parris 1996)

Čvrsto vezivanje za sidrišne miljokaze

U poglavlju 2 smo predstavili više procesnih modela, koji opisuju kako su napredovale tehničke aktivnosti u procesu razvoja softvera. Zatim smo u ovom poglavlju prikazali više metoda organizovanja projekata u cilju sprovođenja tih aktivnosti. Primeri projekata Alpha AXP i F-16 pokazali su nam da upravljanje projektom mora da bude čvrsto integrisano sa procesom razvoja, ne samo u cilju njegovog praćenja, već da bi se sprečilo da veći problemi izbace projekat iz koloseka, što je mnogo važnije s aspekta efikasnijeg planiranja i donošenja odluka. Boehm (1996) je identifikovao tri miljokaza, zajednička za sve procesne modele softverskog razvoja, koji mogu da posluže kao osnova kako za tehnički proces tako i za upravljanje projektom:

- ciljevi životnog ciklusa;
- arhitektura životnog ciklusa;
- inicijalna operativnost sistema.

Detaljnije ćemo ispitati svaki od ovih miljokaza.

Svrha miljokaza *ciljevi životnog ciklusa* je da potvrdi da su se svi zainteresovani subjekti saglasili sa ciljevima posmatranog sistema. Ključni zainteresovani subjekti deluju kao tim za određivanje granica sistema, okruženja u kome on treba da radi, kao i spoljašnjih sistema sa kojima stupa u interakciju. Nakon toga oni pretresaju različite scenarije korišćenja sistema. Ti scenariji mogu biti iskazani prototipom, ekranskim prikazima, tokovima podataka, ili na neki drugi način, od kojih ćemo neke proučavati u narednim poglavljima. Ako je sistem kritičan sa aspekta poslovanja ili bezbednosti, ti scenariji treba takođe da uključe situacije u kojima može doći do otkaza sistema, tako da projektanti mogu odrediti način reakcije sistema na kritični otkaz, čak i način na koji

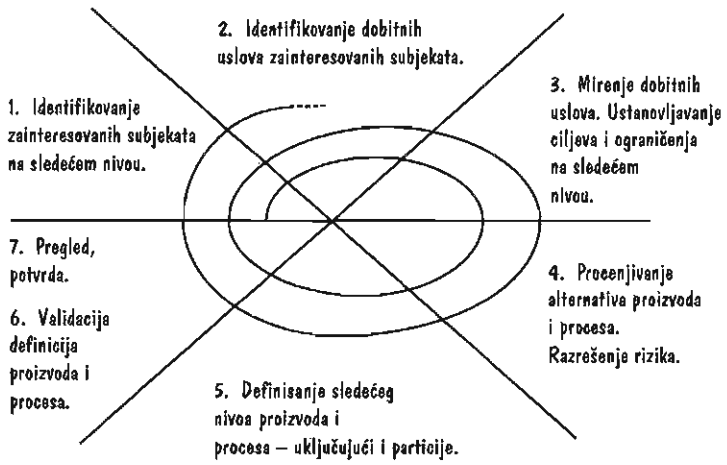
može da se izbegne kritični otkaz. Slično tome izvode se i dogovaraju i ostale suštinske odlike. Rezultat je inicijalni plan životnog ciklusa koji trasira sledeće (Boehm 1996):

- *Ciljevi*: Zašto se posmatrani sistem razvija?
- *Miljokazi i rokovi*: Šta će i do kada biti urađeno?
- *Odgovornosti*: Ko je odgovoran za funkciju?
- *Pristup*: Kako će posao biti urađen, sa tehničke strane i strane rukovođenja?
- *Resursi*: Koliko je potrebno pojedinačnih resursa?
- *Izvodljivost*: Da li se to može uraditi i da li postoji opravdan poslovni razlog da se to uradi?

Arhitektura životnog ciklusa je usklađena sa ciljevima životnog ciklusa. Namena miljokaza u sklopu arhitekture životnog ciklusa jeste da se definišu arhitekture sistema i softvera, čije ćemo delove izračunavati u poglavljima 5, 6 i 7. Izbor arhitekture mora da odražava rizike definisane u sklopu plana upravljanja rizicima. Pri tome, arhitektura mora da se jednako usredsredi na dugoročni razvoj sistema i kratkoročne sistemske zahteve.

Ključni elementi koji određuju *inicijalnu operativnost sistema* su spremnost samog softvera, spremnost lokacije na kojoj će se sistem koristiti, kao i izbor i obuka tima koji će da koristi sistem. Boehm napominje da se za njegovu implementaciju mogu koristiti različiti postupci, uz primenu različitih tehnika procenjivanja u različitim stadijumima.

Boehm predlaže upotrebu spiralnog modela Win-Win, prikazanog na slici 3.22, kao dopunu miljokazima.Ovaj model predstavlja prošireni bazni spiralni model koji smo analizirali u poglavlju 2. On podstiče učesnike da u sledećem stadijumu sustema konvergiraju ka zajedničkom razumevanju ciljeva, alternativa i ograničenja.



SLIKA 3.22 Spiralni model Win-Win (Boehm 1996)

Boehm je primenio tzv. pristup Win-Win, koji se još naziva i teorija W, na program STARS Ministarstva odbrane SAD, čiji cilj je bio razvoj skupa prototipskih okruženja za softversko inženjerstvo. Projekat je bio dobar kandidat za primenu teorije W, jer je postojalo veliko neslaganje između onoga šta je vlada planirala da sagradi i onoga što su poten-

cijalni korisnici hteli i šta im je bilo potrebno. Win-Win model je doveo do niza ključnih ustupaka, uključujući dogovor o skupu zajedničkih otvorenih specifikacija interfejsa, što je omogućilo isporučiocima alata veći udeo na tržištu uz smanjenje troškova, kao i uključivanje tri demonstraciona projekta za smanjenje rizika. Boehm izveštava da su troškovi vazduhoplovnih snaga na datom projektu bili smanjeni sa 140 na 57 dolara po redu koda, a da je kvalitet poboljšan sa 3 na 0,035 nedostataka na hiljadu isporučenih redova koda. Izveštaji o sličnom uspehu postoje za više različitih projekata. Koristeći miljokaze koje je definisao Boehm, TRW je, u pet inkremenata, razvila preko pola miliona redova koda složenog distribuiranog softvera u roku i u okviru budžeta. Prvi inkrement je, kao miljokaz u sklopu arhitekture životnog ciklusa, uključivao softver za distribuirano jezgro. Od projekta se tražilo da pokaže da će biti u stanju da zadovolji sve veći broj zahteva koji se pred njega budu postavljali tokom vremena (Royce 1990).

3.7 PRIMER IZ INFORMACIONOG SISTEMA

Ponovo se vraćamo na sistem za prodaju oglasnog vremena televizije Picadilly, da bismo videli kako možemo da procenimo količinu potrebnog rada za izradu softvera. Kako se nalazimo u inicijalnoj fazi u kojoj jedino treba da shvatimo šta softver treba da radi, možemo se osloniti na model COCOMO II za uloženi rad, i na osnovu njega prognozirati potreban broj radnih meseci **Radni mesec** predstavlja radno angažovanje jednog čoveka u toku jednog meseca rada na projektu softverskog razvoja. Model COCOMO podrazumeva da broj ranih meseci ne uključuje praznike i godišnji odmor, kao ni vikend. Broj radnih meseci nije jednak vremenu potrebnom za dovršavanje izgradnje sistema. Na primer, sistem može da zahteva 10 radnih meseci, ali može biti završen za jedan mesec tako što će deset ljudi raditi paralelno mesec dana, ili za dva meseca ako to čini pet ljudi (uz pretpostavku da zadaci mogu da se izvršavaju na taj način).

Prvi COCOMO II model, kompozicija aplikacije, osmišljen je da se koristi u najranijim fazama razvoja. Tu izračunavamo aplikativne poene koji nam pomažu da odredimo verovatnu veličinu projekta. Broj aplikativnih poena određuje se na osnovu tri proračuna. Prvi proračun nam daje broj tabela sa podacima o serverima koje se prikazuju na ekranu ili koriste u izveštajima. Drugi proračun je broj tabela sa podacima o klijentima koje se prikazuju na ekranu ili koriste u izveštajima. Treći proračun je procenat prikaza, izveštaja i modula preuzetih iz ranijih aplikacija. Pretpostavimo da ne koristimo nikakav preuzeti kôd prilikom gradnje sistema za televiziju Picadilly. Zatim moramo da započnemo predviđanje koliko ćemo prikaza i izveštaja da koristimo u ovoj aplikaciji. Neka je inicijalna pretpostavka da su nam potrebna tri ekranska prikaza i jedan izveštaj:

- ekran za rezervacije, za evidentiranje novih rezervacija za prodaju oglasnog prostora;
- ekran indeksa gledanosti, gde se prikazuju indeksi gledanosti reklama za svaki dan i sat;
- ekran raspoloživosti koji navodi koji vremenski intervali su raspoloživi;
- izveštaj o prodaji gde se prikazuje ukupna prodaja za mesec i godinu, uz poređenje sa prethodnim mesecima i godinama.

Za svaki ekran i izveštaj, koristimo uputstva iz tabele 3.10 i izračunavamo koliki broj tabela sa podacima će biti potrebno za pravljenje opisa ekrana ili izveštaja. Na primer,

ekran za rezervaciju zahteva upotrebu tri tabele; tabelu sa raspoloživim vremenskim intervalima, tabelu sa vremenima koje je ranije koristio isti naručilac, i tabelu sa podacima neophodnim za kontaktiranje sa datim naručiocem (kao što su ime, adresa, poreski broj, radnik na prodaji). Prema tome, potreban broj tabela je manji od četiri, tako da moramo da se odlučimo da li nam treba više od osam pogleda. Pošto će nam verovatno trebati manje od osam pogleda, procenjujemo prikaz za rezervacije kao „jednostavan", saglasno tabeli aplikativnih poena. Slično tome, procenjujemo prikaz stope gledanosti kao „jednostavan", prikaz raspoloživosti kao „srednji" i izveštaj o prodaji kao „srednji". Dalje, koristimo tabelu 3.11 za dodeljivanje indeksa složenosti 1 za jednostavne ekrane, 2 za srednje ekrane i 5 za srednje izveštaje. Rezime ovih procena prikazan je u tabeli 3.15.

Tabela 3.15 Procena potrebnh ekrana i izveštaja za sistem Picadilly

Naziv	Ekran ili izveštaj	Složenost	Vrednosti
Rezervacije	Ekran	jednostavan	1
Stopa gledanosti	Ekran	jednostavan	1
Raspoloživost	Ekran	srednji	2
Prodaja	Izveštaj	srednji	5

Sabiranjem svih vrednosti u krajnjoj desnoj koloni generišemo zbir poena za novu aplikacije, tj. NOP poene za novi objekat, koji ovde iznose 9. Pretpostavimo da razvojni tim ima malo iskustvo i nizak stepen CASE zrelost. Tabela 3.12 govori nam da je stopa produktivnosti u ovom slučaju jednaka 7. Zatim nam model COCOMO govori da procenjeni napor za gradnju sistema Picadilly dobijamo kada vrednost NOP podelimo sa stopom produktivnosti, što daje 1,29 radnih meseci.

Kada budemo više razumeli potrebe sistema televizije Picadilly, moći ćemo da koristimo i druge delove modela COCOMO, kao što su model ranog projektovanja i postarhitektonski model, zasnovan na nominalnim procenama napora izvedenim iz broja redova koda ili funkcionalnih tačaka. Ti modeli koriste eksponent razmere koji se izračunava na osnovu faktora razmere projekta, navedenih u tabeli 3.16.

Tabela 3.16 Faktori razmere za COCOMO II model ranog projektovanja i postarhitektonski model

Faktori razmere	Vrlo nizak	Nizak	Nominalan	Visok	Vrlo visok	Ekstra visok
Prethodno iskustvo	Bez prethodnog iskustva	U velikoj meri novo	Delimično novo	Generalno blisko	U velikoj meri blisko	Potpuno blisko
Fleksibilnost	Rigorozna	Ponekad relaksirana	Nešto relaksirana	Uopšteno ugodna	Donekle ugodna	Opšti ciljevi
Eliminisani značajni rizici	Malo (20%)	Nešto (40%)	Često (60%)	Uopšteno (75%)	Uglavnom (90%)	Potpuno (100%)
Interakcije u timu	Vrlo teške interakcije	Nešto teže interakcije	U osnovi kooperativne interakcije	Vrlo kooperativne	Visoko kooperativne	Bespredmetne interakcije
Zrelost procesa	Određena upitnikom	Određena upitnikom	Određena upitnikom	Određena upitnikom	Određena upitnikom	Određena upitnikom

Oznaka „ekstra visok" odgovara oceni nula, „vrlo visok" predstavlja ocenu 1, „visok" predstavlja ocenu 2, „nominalan" predstavlja ocenu 3, „nizak" predstavlja ocenu 4 i „vrlo

nizak" predstavlja ocenu 5. Svaki od faktora razmere se rangira, a zbir svih rangova se koristi kao vrednost za izračunavanje inicijalnog rada. Na primer, pretpostavimo da nam je poznato da je tip aplikacije, koju gradimo za Picadilly, generalno poznat razvojnom timu. Zbog toga možemo prvi faktor razmere da ocenimo kao „visok". Slično tome, fleksibilnost može da se oceni kao „vrlo visoka", dok je rešavanje rizika „nominalano". Interakcija tima je „visoka", a rang zrelosti može da se ispostavi da je „niska". Kada saberemo sve vrednosti (2 + 1 +3 + 2+ 4) dobijamo faktor razmere 12. Zatim računamo eksponent razmere kao

$$1.01 + 0.01(12)$$

što daje rezultat 1.13. Eksponent razmere nam govori da, ako je naša procena inicijalnog rada iznosila 100 radnih meseci, tada naša nova procena, u odnosu na karakteristike iskazane u tabeli 3.16, ima vrednost $100^{1.13}$, ili 182 radna meseca. Na sličan način, nosioci troškova koriguju tu procenu na osnovu karakteristika kao što su upotreba alata, stručnost analitičara i zahtevi vezani za pouzdanost. Kada odredimo faktor korekcije, njime množimo našu procenu od 182 radna meseca, čime dobijamo korigovanu procenu potrebnog rada.

3.8 PRIMER IZ PRAKSE

Komisija koja je istraživala otkaz rakete Ariana-5, ispitivala je softver, dokumentaciju, kao i podatke snimljene pre i tokom leta, da bi utvrdila šta je dovelo do otkaza (Lions i dr. 1996). U njihovom izveštaju napominje se da je lansirna raketa počela da se dezintegriše 39 sekundi nakon poletanja, jer je napadni ugao premašivao 20 stepeni, prouzrokujući da se potisni motori odvoje od glavnog tela rakete. Ovo odvajanje aktiviralo je samouništenje lansirane rakete. Napadni ugao određivan je pomoću softvera na računaru u samoj raketi, na osnovu podataka koje emituje aktivni inercijalni referentni sistem SRI2. Kao što je napomenuto u izveštaju, SRI2 je trebalo da sadrži validne podatke o letu, ali umesto toga, sadržao je dijagnostički uzorak, koji je bio pogrešno protumačen kao podatak o letu. Pogrešni podaci su klasifikovani kao otkaz uređaja, pa je izvršeno blokiranje SRI2 sistema. Uobičajeno je bilo da se računar letelice prebaci na drugi inercijalni referentni sistem, SRI1, ali je taj, iz istog razloga, bio isključen.

Greška se dogodila u softverskom modulu koji je davao smislene rezultate samo pre poletanja. Čim je došlo do uzletanja lansirne rakete, funkcije koje je taj modul izvodio nisu više bile potrebne ostatku sistema. Međutim, modul je nastavio svoje rad otprilike u sledećih 40 sekundi leta, u skladu sa zahtevima definisanim kod rakete Ariana-4, koji nisu bili potrebni u slučaju rakete Ariana-5.

Interni događaji koji su doveli do otkaza, reprodukovani su simulacijom proračuna uz pomoć iščitane memorije i ispitivanjem samog softvera. Prema tome, uništenje Ariane-5 moglo je biti sprečeno da su rukovodioci projekta razvili plan upravljanja rizicima, izvršili njegov ponovni pregled i razvili plan izbegavanja ili ublažavanja svakog identifikovanog rizika. Da bismo videli kako, razmotrimo ponovo korake sa slike 3.15. Prva faza ocenjivanja rizika je njegova identifikacija. Mogući problem sa preuzimanjem delova postojećeg softvera za Arianu-4 mogao je biti identifikovan raščlanjivanjem funkcija. Neko je mogao u ranoj fazi da prepozna da se zahtevi za Arianu-4 razlikuju od zahteva Ariane-5. Ili je tokom analiza pretpostavki moglo da se otkrije da su pretpostavke u vezi sa SRI uređajima u raketi Ariana-4 bile različite od pretpostavki u Ariani-5.

Nakon identifikovanja rizika, faza analize mogla je da obuhvati i simulacije, koje bi verovatno istakle problem koji je na kraju doveo do uništenja rakete. Takođe je određivanje prioriteta moglo da identifikuje izloženost riziku ako SRI ne radi kao što je planirano. Visoka izloženost je mogla da podstakne projektni tim da pre same implementacije pažljivije ispita SRI i njegov rad.

Kontrola rizika obuhvata umanjenje rizika, plansko upravljanje rizicima i razrešavanje rizika. Čak i da je tokom aktivnosti ocenjivanja rizika propušteno uočavanje problema koji su bili neizbežni zbog SRI uređaja preuzetog iz Ariane-4, pomoću tehnika smanjenja rizika, uključujući analizu izbegavanja rizika, bilo je moguće uočiti da isti razloga može dovesti do istovremene blokade oba SRI uređaja. Izbegavanje rizika je moglo da se odradi korišćenjem i jednog i drugog SRI uređaja u dva različita projektna rešenja, i tada bi greška u projektu dovela do blokade jednog, ali ne i drugog uređaja. Činjenica da SRI proračuni nisu potrebni nakon uzletanja, mogla je da podstakne projektante ili programere da ugrade ranije isključenje SRI-ja, pre nego što on ošteti podatke za proračunavanje ugla. Slično tome, razrešavanje rizika uključuje planove za ublažavanje i kontinuirano ponovno ocenjivanje rizika. Čak i da rizik od otkaza SRI-ja nije bio uočen ranije, ponovno ocenjivanje rizika tokom projektovanja ili čak tokom testiranja celina moglo je da otkrije problem u toku samog razvoja. Prepravke projektnih rešenja ili dodatni razvoj u toj fazi bili bi skupi, ali ne toliko skupi kao kompletan gubitak Ariane-5 na njenom prvom letu.

3.9 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju upoznali ste se sa nekim od ključnih koncepata u upravljanju projektima, uključujući planiranje projekta, procenjivanje troškova i rokova, upravljanje rizicima, kao i organizaciju tima. Ove informacije možete da iskoristite na više načina, čak i ako niste rukovodilac. Planiranje projekta uključuje ulazne informacije od svih članova tima, uključujući i vas, a razumevanje procesa planiranja i tehnika procenjivanja daje vam dobru sliku o tome kako će vaši ulazni podaci biti upotrebljeni za donošenje odluka na nivou celog tima. Takođe, videli smo kako sa uvećanjem tima raste broj mogućih putanja komunikacije. Komunikaciju možemo da uzmemo u obzir kada planiramo rad i procenjujemo potrebno vreme za kompletiranje sledećeg zadatka.

Videli smo takođe kako se razlikuju stilovi komunikacije i kako oni utiču na način međusobne interakcije prilikom obavljanja posla. Razumevanjem stilova vaših kolega iz tima, možete za njih da napravite izveštaje i prezentacije koji odgovaraju njihovim očekivanjima i potrebama. Možete da pripremite sumarne informaciju za osobe koje ne vole previše informacija i da ponudite kompletnu analitičku informaciju onima koji imaju racionalan pristup.

3.10 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

U isto vreme, naučili ste kako da organizujete razvojni tim, tako da interakcija u okviru tima pomogne postizanju boljih rezultata. Na raspolaganju vam je nekoliko mogućih struktura tima, od hijerarhijskog tima sa glavnim programerom, do slabo povezanog pristupa bez egoizma. Svaka od struktura ima svoje prednosti, i svaka do određenog stepena zavisi od neizvesnosti projekta i njegove veličine.

Takođe smo videli kako tim može da radi od samog početka projekta na predviđanju i redukovanju rizika. Redundantna funkcionalnost, kontrolni sastanci tima, kao i druge tehnike mogu da nam pomognu kod ranog otkrivanja grešaka, pre nego što one budu ugrađene u kôd i postanu nedostataci koji mogu da prouzrokuju otkaze.

Slično tome, procenjivanje troškova treba da se radi od samog početka projekta i zatim često, uključujući ulazne podatke dobijene od članova tima o napretku u izradi specifikacija, projektovanju, kodiranju i testiranju sistema. Procena troškova i upravljanje rizicima mogu da idu zajedno. Kako procene troškova uvećavaju sumnje vezane za ispunjenje rokova i budžet, tehnike upravljanja rizicima mogu da se koriste za ublažavanje ili eliminisanje rizika.

3.11 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

U ovom poglavlju opisano je više tehnika koje još uvek zahtevaju značajna istraživanja. Nema dovoljno znanja o tome koja organizacija tima je najbolja za određene situacije. Pored toga, modeli procene troškova i rokova nisu precizni onoliko koliko bismo želeli, tako da su poboljšanja moguća sa sve potpunijim shvatanjem načina na koje projekat, proces, proizvod i karakteristike resursa utiču na efikasnost i produktivnost projektanata. Neki metodi dosta obećavaju, npr. mašinsko učenje, ali oni, da bi bili precizniji, zahtevaju obimne istorijske podatke. Istraživači mogu da nam pomognu da razumemo kako da uravnotežimo praktičnost primene i tačnost upotrebljene tehnike za procenu.

Slično tome, potrebna su obimna istraživanja da bi tehnike upravljanja rizicima postale praktične. Proračun izloženosti riziku je trenutno više umetnost nego nauka, a nama su potrebni metodi koji doprinose relevantnosti proračuna rizika i efikasnosti tehnika ublažavanja njihovih uticaja.

3.12 SEMINARSKI RAD

Često se dešava da kompanija ili organizacija moraju da procene napor i vreme potrebno za završetak projekta, čak i pre nego što su formulisani detaljni zahtevi. Pomoću pristupa opisanih u ovom poglavlju, ili alata po vašem izboru iz drugih izvora, procenite potreban napor za gradnju sistema *Aranžer pozajmica*. Koliko će ljudi biti potrebno? Koja znanja oni treba da poseduju? Koliko iskustva? Koje vrste alata ili tehnika možete da koristite za skraćivanje vremena potrebnog za realizaciju projekta?

Možda ćete za generisanje procena hteti da koristite više različitih pristupa. Ako je tako, uporedite rezultate. Ispitajte svaki pristup (njegove modele i pretpostavke) da biste utvrdili čime možete objasniti suštinske razlike između procena.

Nakon napravljenih procena, potrebno je izvršiti njihovo ocenjivanje, kao i ocenjivanje osnovanih pretpostavki, da biste videli koliki stepen neizvesnosti one sadrže. Zatim treba izvesti analizu rizika. Sačuvajte rezultate, jer možete da ih ispitate na kraju projekta da biste videli kako se rizici pretvaraju u realne probleme, kao i sagledali one rizike koji su ublaženi odabranim strategijama upravljanja rizicima.

3.13 KLJUČNE REFERENCE

Veliki deo informacija o modelu COCOMO pruža Centar za softversko inženjerstvo pri Univerzitetu Južne Kalifornije (Center for Software Engineering, University of Sou-

thern California). Veb lokacija http://sunset.usc.edu/COCOMOII/Cocomo.html sadrži aktuelna istraživanja čiji je predmet model COCOMO, uključujući implementaciju modela COCOMO II na programskom jeziku Java. Na toj lokaciji možete da pronađete informacije o sastancima COCOMO korisničkih grupa i dobijete primerak COCOMO II korisničkog uputstva. Informacije u vezi sa funkcionalnim tačkama pruža IFPUG (*International Function Point User Group*) iz Westervilla, Ohio.

Centar za softversko inženjerstvo takođe sprovodi istraživanja u vezi sa upravljanjem rizicima. Tehnički savetnik za softverske rizike (Software Risk Technical Advisor) može da se preuzme sa ftp lokacije ftp://usc.edu/pub/soft_engineering/demos/stra.tar.z. O aktuelnim istraživanjima možete da pročitate na lokaciji http://sunset.usc.edu.

PC alati za podršku procenjivanju opisani su i raspoloživi na veb lokaciji http://dec.bournemouth.ac.uk/ESERG koju održava Empirical Software Engineering Research Group, Bournemouth University.

Više kompanija koje proizvode komercijalne alate za upravljanje projektima i procenjivanje troškova, pruža informacije na svojim veb lokacijama. Quantitative Software Management, proizvođač paketa SLIM za procenu troškova, nalazi se na lokaciji http://www.qsm.com. Slično tome, Software Productivity Research nudi paket pod nazivom Checkpoint. Informacije mogu da se pronađu na lokaciji http://www.spr.com. Computer Associates je razvio veliku grupu alata za upravljanje projektima uključujući Estimacs za procenu troškova i Planmacs za planiranje. Potpuni opis proizvoda dat je na lokaciji http://www.cai.com/products.

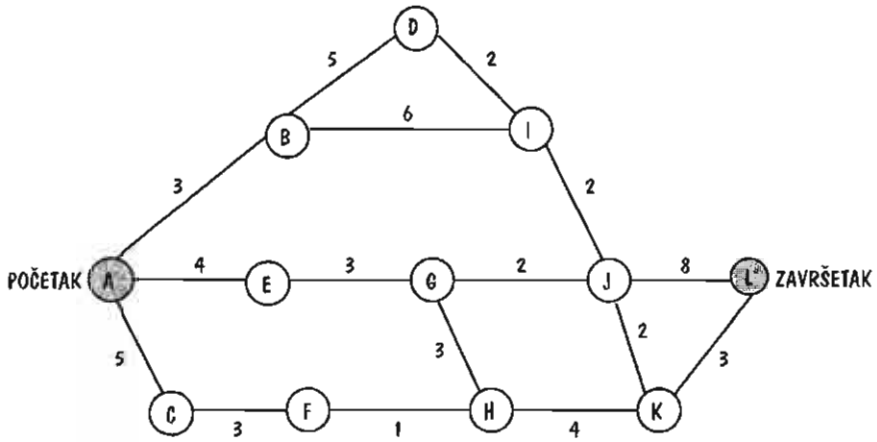
Software Technology Support Center u bazi Hill Air Force Base u Ogdenu, Juta, izdaje bilten pod nazivom *CrossTalk* koji izveštava o ocenjivanju metoda i alata. Njihove smernice za uspešno prikupljanje i upravljanje mogu da se pronađu na lokaciji http://stsc.hill.af.mil/stscdocs.html. Osnovna veb stranica Centra sadrži skup pokazivača na različite tehnološke oblasti, uključujući i upravljanje projektima i procenu troškova a možete je naći u sklopu lokacije http://stsc.hill.af.mil.

Formiranje tima i interakcije u okviru njega čine suštinu dobrih softverskih projekata. Weinberg (1993) u svom drugom tomu serije o kvalitetu softvera razmatra stilove rada i njihovu primenu pri formiranju tima. Scholtes (1995) obuhvata materijal o tome kako da se postupa sa „teškim" članovima tima.

Upravljanje malim projektima obavezno se razlikuje od upravljanja velikim. Izdanje časopisa *IEEE Computer* iz oktobra 1999. bavi se softverskim inženjerstvom u malom, sa člancima o malim projektima, pritiscima koje donosi vreme na Internetu, kao i o ekstremnom programiranju.

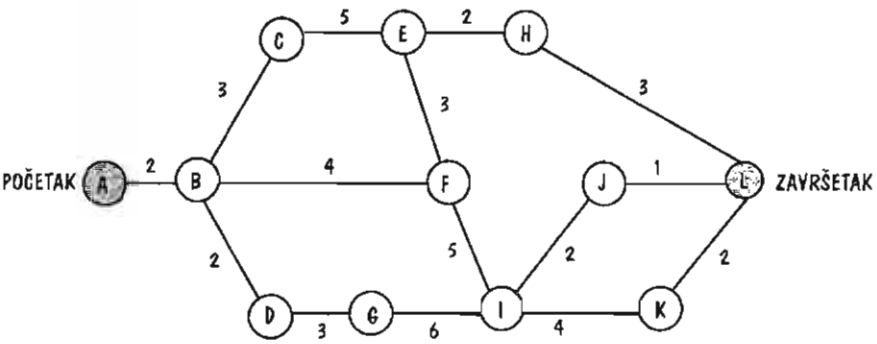
3.14 VEŽBE

1. Pripremate se da ispečete rođendansku tortu sa dve kore i prelivom od čokolade. Opisati projekat pečenja kolača kao strukturu posla. Iz te strukture generisati graf aktivnosti. Šta je kritična putanja?
2. Slika 3.23 predstavlja graf aktivnosti projekta softverskog razvoja. Broj dodeljen svakoj grani grafa ukazuje na broj dana koji su potrebni za dovršavanje aktivnosti po toj grani. Na primer, potrebna su 4 dana za okončanje aktivnosti koja rezultira miljokazom E. Za svaku aktivnost, navesti njene prethodnike, izračunati najranije vreme započinjanja, najkasnije vreme započinjanja, kao i latentno vreme. Zatim, identifikovati kritičnu putanju.



SLIKA 3.23 Graf aktivnosti za vežbu 2.

3. Slika 3.24 predstavlja graf aktivnosti. Pronaći kritičnu putanju.



SLIKA 3.24 Graf aktivnosti za vežbu 3.

4. U projektu softverskog razvoja, koje vrste aktivnosti mogu da se izvode paralelno? Objasniti zašto grafovi aktivnosti ponekad skrivaju međusobne zavisnosti tih aktivnosti.
5. Opisati zašto dodavanje osoblja na projekat koji kasni može da prouzrokuje još veće kašnjenje.
6. Velika vladina agencija želi da sačini ugovor sa preduzećem za razvoj softvera za projekat koji obuhvata 20 000 redova koda. Hardand Software Company koristi Walston-Felixovu tehniku procenjivanja za određivanje broja potrebnih izvršilaca na osnovu vremena potrebnog za pisanje toliko redova koda. Koliko radnih meseci će zahtevati Hardandova procena? Ako je vladina procena veličine za 10% manja (tj. 20 000 redova koda predstavljaju samo 90% stvarne veličine), koliko će dodatnih radnih meseci biti potrebno? U opštem slučaju, ako je vladina procena veličine za k% manja, za koliko mora da se promeni procena u pogledu radnih meseci?
7. Objasniti zašto razvoj pomoćnog programa traje duže nego razvoj aplikativnog programa, a zašto razvoj sistemskog programa traje još duže?

8. Manny's Manufacturing mora da odluči da li da da kupi ili da samostalno razvija softver ili softverski paket za praćenje inventara. Njihovi računarski stručnjaci procenjuju da će kupovina neophodnih programa koštati 325 000 dolara. Ako se program razvija u vlastitom aranžmanu, svaki programer košta 5 000 dolara mesečno. Koje faktore treba da razmotri ova kompanija prilikom donošenja odluke? U kom slučaju je bolji samostalni razvoj, a kada kupovina?
9. Brooks tvrdi da dodavanje ljudi na projekte koji kasne prouzrokuje još veće kašnjenje (Brooks 1975). Neke tehnike za procenu rokova izgleda da ukazuju da dodavanje ljudi na projekat može da skrati vreme razvoja. Da li se radi o protivrečnosti? Da li je to slučajnost ili ne?
10. Mnoga istraživanja pokazuju da su dva osnovna uzroka kašnjenja projekta: izmene zahteva (naziva se nestalnost ili nestabilnost zahteva) i plate radnika. Analizirajte modele troškova, koji su razmatrani u ovom poglavlju, kao i one koje možda koristite na vašem poslu, i odredite koji modeli poseduju faktore troškova koji odražavaju uticaj pomenutih razloga kašnjenja.
11. Čak i u slučaju studentskih projekata, postoje značajni rizici da se projekat ne završi na vreme. Analizirati studentski projekat softverskog razvoja i navesti rizike. Šta je izloženost riziku? Koje tehnike možete da koristite za ublažavanje svakog rizika?
12. Mnogi rukovodioci projekata planiraju rokove na osnovu produktivnosti programera na prethodnim projektima. Ta produktivnost se često iskazuje u jedinicama veličine po jedinici vremena. Na primer, jedna organizacija može da proizvede 300 redova koda na dan ili 1 200 aplikativnih poena mesečno. Da li je podesno da merimo produktivnost na ovaj način? Diskutovati o merenju produktivnosti u zavisnosti od sledećih pitanja:
    - Različiti programski jezici mogu da zahtevaju različit broj redova koda za istu implementaciju projektnog rešenja.
    - Produktivnost u redovima koda ne može da se meri sve dok ne počne implementacija.
    - Programeri mogu da strukturiraju kôd radi dostizanja ciljeva produktivnosti.

4 Evidentiranje zahteva



U ovom poglavlju obradivaćemo
- izvođenje zahteva;
- modelovanje zahteva;
- pregledanje zahteva radi osiguravanja njihovog kvaliteta;
- dokumentovanje zahteva za projektne timove i timove testira.

U prethodnim poglavljima, posmatrajući razne modele procesa, uočili smo nekoliko ključnih koraka za uspešan razvoj softvera. Svaki od predloženih procesnih modela softverskog razvoja uključuje i posebne aktivnosti čiji je cilj evidentiranje zahteva, tj. razumevanje osnovnih problema i potreba naručioca. Prema tome, shvatanje funkcije i namene sistema započinje ispitivanjem zahteva. U ovom poglavlju videćemo razne tipove zahteva i njihove različite izvore, pri čemu ćemo razmatrati kako da razrešimo protivrečne zahteve. Detaljno ćemo opisati razne notacije za modelovanje i metode za specifikaciju zahteva, sa primerima automatskih i ručnih tehnika. Ti modeli nam pomažu da razumemo zahteve i dokumentujemo njihove odnose. Kada se zahtevi dovoljno dobro shvate, naučićemo kako da ih pregledamo da bismo bili sigurni da su ispravni i potpuni. Na kraju poglavlja naučićemo kako da odaberemo metod specifikacije zahteva koji odgovara razmatranom projektu, na osnovu veličine, obima projekta i kritičnosti njegove misije.

Analiza zahteva obuhvata mnogo više od jednostavnog evidentiranja onoga što naručilac želi. Kao što ćemo videti, neophodno je pronaći zahteve oko kojih možemo da se usaglasimo sa naručiocem i za koje možemo da formulišemo test procedure. Prvo razmotrimo šta su to zahtevi, zašto su važni (o tome pročitajte u podsetniku 4.1) i kako sarađujemo sa korisnicima i naručiocima da bismo ih definisali i dokumentovali.

PODSETNIK 4.1 ZAŠTO SU ZAHTEVI VAŽNI?

Najteži pojedinačni segment u izgradnji softvera jeste precizno odlučivanje šta da se gradi. Nijedan drugi deo aktivnosti na konceptualnom nivou nije tako težak kao uspostavljanje detaljnih tehničkih zahteva, uključujući sve sprege sa ljudima, mašinama i drugim softverskim sistemima. Nijedan deo aktivnosti ne može u tom stepenu obogaljiti rezultujući sistem, kao loše formulisani zahtevi. Nijedan drugi deo nije toliko teško naknadno ispravljati.

(Brooks 1987)

Godine 1994. grupa Standish ispitala je ishode više od 8 000 softverskih projekata u preko 350 kompanija. Rezultati su bili „za plakanje". Trideset pet procenata softverskih projekata je otkazano pre nego što su završeni. Još gore, u velikim kompanijama, samo je 9% projekata isporučeno na vreme i u okviru budžeta, dok je u malim kompanijama 16% doseglo te kriterijume (Standish 1994). Od tada postoji više sličnih izveštaja pri čemu je neizbežna činjenica da razvojne organizacije imaju problema sa isporukom odgovarajućeg sistema na vreme i u okviru budžeta.

Da bi razumela zašto je to tako, grupa Standish (1995.) je tražila od učesnika u istraživanju da objasne uzroke neuspeha projekata. Nakon toga, ona je objavila prikaz deset faktora koji se nalaze na vrhu lestvice:

1. nekompletni zahtevi (13,1%);
2. nedovoljno učešće korisnika (12,4%);
3. nedostatak resursa (10,6%);
4. nerealna očekivanja (9,9%);
5. odsustvo podrške rukovodstva (9,3%);
6. izmene zahteva i specifikacija (8,7%);
7. odsustvo planiranja (8,1%);
8. sistem više nije bio potreban (7,5%).

Imajmo u vidu da je u skoro svim tim slučajevima uključen neki deo procesa iskazivanja, definisanja i upravljanja zahtevima. Nedostatak pažnje prilikom razumevanja, dokumentovanja i upravljanja zahtevima može da dovede do niza problema. Primeri su gradnja sistema koji rešava pogrešan problem, gradnja sistema koji ne funkcioniše kao što je očekivano ili gradnja sistema koji korisnici ne razumeju i teško koriste.

Osim toga, greške u zahtevima mogu da budu vrlo skupe ako nisu otkrivene i ispravljene u ranim fazama procesa razvoja. Boehm i Papaccio (1988) izveštavaju da ako tokom procesa definisanja zahteva otkrivanje i otklanjanje problema u zahtevima košta 1 dolar, odgovarajuća ispravka košta 5 dolara u fazi dizajna, 10 dolara tokom kodiranja, 20 dolara tokom pojedinačnog testiranja a više od 200 dolara nakon isporuke sistema. Stoga vredi odvoniti vreme na razumevanje problema i njegovog konteksta da bi se na samom početku dobili ispravni zahtevi.

4.1 PROCES VEZAN ZA ZAHTEVE

Naručilac koji od nas zahteva gradnju novog sistema ima određeno poimanje o tome šta sistem treba da radi. Naručilac često želi da automatizuje zadatke koji se obavljaju

ručno, kao što je elektronsko plaćanje računa, umesto ručnog ispisa čekova. Ponekad naručilac želi da poboljša ili proširi postojeći ručni ili automatizovani sistem. Na primer, telefonski sistem za naplatu koji je klijentima obračunavao samo lokalnu telefonsku uslugu i međugradske pozive, mogao bi da se unapredi preusmeravanjem poziva, poziva na čekanju i drugim novim svojstvima. Naručioci sve češće i žele proizvode koji rade nešto što se do tada nisu radili, kao na primer: prilagođavanje elektronskih novosti interesovanjima korisnika, promena geometrije krila aviona u toku leta ili nadgledanje šećera u krvi dijabetičara i automatsko kontrolisanje doziranja insulina. Bez obzira na to da li je određena funkcionalnost stara ili nova, predloženi softverski sistem poseduje namenu, obično izraženu ciljevima ili željenim ponašanjem.

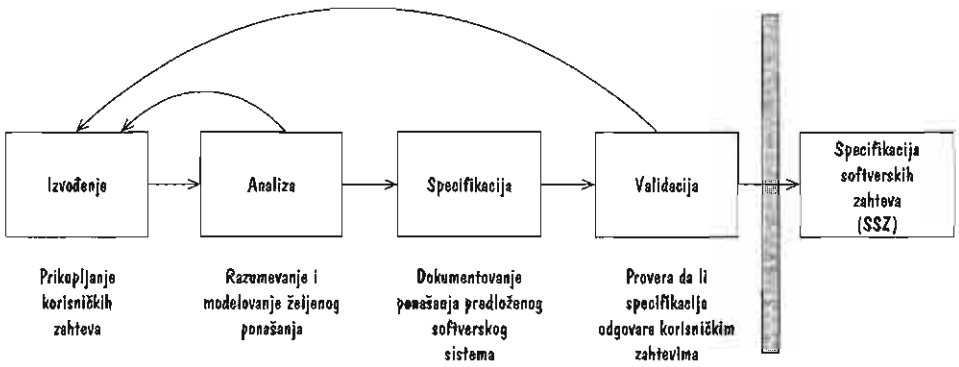
Zahtev predstavlja izraz željenog ponašanja. Zahtev se bavi objektima ili entitetima, stanjima u kojima oni mogu biti, kao i funkcijama koje se izvode radi promene stanja ili osobina objekta. Na primer, pretpostavimo da gradimo sistem za generisanje platnih listića za kompaniju našeg naručioca. Jedan od zahteva mogao bi da bude da se listići izdaju svake dve nedelje. Drugi zahtev mogao bi da bude mogućnost direktne uplate depozita za svakog radnika sa datim ili višim nivoom primanja. Naručilac može da zahteva pristup sistemu za obračun zarada sa više različitih lokacija unutar kompanije. Svi ti zahtevi predstavljaju specifične opise funkcija ili karakteristika koje se bave opštom namenom sistema, a to je generisanje platnih listića. Stoga tražimo zahteve koji identifikuju ključne entitete („radnik je osoba koju plaća kompanija"), entitete koji nameću ograničenja („radnik ne može biti plaćen za više od 40 sati rada nedeljno") ili definišu odnose između entiteta („radnika X nadgleda radnik Y, ako je Y ovlašćen da izmeni zaradu koju X ostvaruje").

Napomenimo da nijedan od tih zahteva ne određuje način implementacije sistema. Nema naznaka o sistemu za upravljanje bazama podataka koji treba da se koristi, da li će se koristiti klijent-server arhitektura, koliko memorije treba da ima računar ili koji će programski jezik biti korišćen za implementaciju sistema. Opisi, koji su specifični za implementaciju, ne spadaju u zahteve (osim ako ih naručilac nije striktno propisao). Cilj faze definisanja zahteva je razumevanje problema i potreba naručioca. Prema tome, zahtevi su okrenuti naručiocu i problemu, a ne rešenju ili implementaciji. Često kažemo da zahtevi označavaju *kakvo* ponašanje naručilac želi, bez izražavanja *kako* će to ponašanje biti ostvareno. Rasprava o rešenju je preuranjena sve dok se problem jasno ne definiše.

Korisno je zahteve opisati kao interakciju između pojava u realnom svetu, bez referenciranja pojava na nivou sistema. Na primer, zahtevi u vezi sa naplatom treba da referenciraju klijente, usluge koje se plaćaju, periode naplate i iznose, bez pominjanja sistemskih podataka ili procedura. Ovaj pristup upražnjavamo delom zbog toga da bismo došli do suštine potreba naručioca jer nekada iskazane potrebe ne odražavaju stvarne. Pored toga, problem koji naručilac ima je obično najlakše iskazati u kategorijama njegovog poslovanja. Drugi razlog zbog kog upražnjavamo takav pristup je da bismo projektantima omogućili maksimalnu fleksibilnost prilikom donošenja odluke o načinu ispunjavanja zahteva. Tokom faze **specifikacije**, odlučićemo koje zahteve će projektovani softverski sistem da ispuni (nasuprot zahtevima koji realizuju hardverski uređaji posebne namene, drugi softverski sistemi, operateri ili korisnici). Tokom faze **dizajna**, formulisaćemo plan implementacije navedenog ponašanja.

Na slici 4.1 je prikazan proces evidentiranja zahteva za predloženi softverski sistem. Izvršilac tih zadataka obično je **analitičar zahteva** ili **analitičar sistema**. Analitičar zahteva prvo radi sa naručiocima na izvođenju zahteva: postavlja pitanja, ispituje aktuel-

no ponašanje ili demonstrira slične sisteme. Nakon toga zahteve evidentiramo u sklopu modela ili prototipa. Ova vežba nam pomaže da bolje razumemo zahtevano ponašanje i obično otvara dodatna pitanja o tome šta naručilac želi da se u određenim situacijama dogodi (na primer, šta se dešava ako radnik napusti preduzeće u sred obračunskog perioda). Kada se postigne dobro razumevanje zahteva, posao se nastavlja izradom specifikacije, u kojoj se odlučuje koji će delovi zahtevanog ponašanja biti implementirani u softveru. Tokom validacije, proverava se da li specifikacija odgovara onome što naručilac želi da vidi u finalnom proizvodu. Aktivnosti analize i validacije mogu da otkriju probleme ili propuste u modelu ili specifikaciji, što prouzrokuje ponovne posete naručiocu i revidiranje modela i specifikacije. Krajnji ishod tog procesa je specifikacija softverskih zahteva (*Software Requirements Specification*, SRS), koja se koristi za komunikaciju sa drugim članovima razvojnog tima (koji rade na dizajnu, testiranju i održavanju), o tome kako konačni proizvod treba da se ponaša. Podsetnik 4.2 razmatra kako korišćenje agilnih metoda utiče na proces definisanja zahteva i na rezultujuće dokumente koji sadrže zahteve. U nastavku ovog poglavlja detaljnije istražujemo proces definisanja zahteva.



SLIKA 4.1 Proces evidentiranja zahteva

4.2 IZVOĐENJE ZAHTEVA

Izvođenje zahteva je posebno važan deo procesa. Moramo da koristimo razne tehnike za utvrđivanje šta korisnici i naručioci stvarno žele. Ponekad automatizujemo ručni sistem, tako da je utvrđivanje šta sistem stvarno radi lako. Međutim, često moramo da radimo sa korisnicima i naručiocima da bismo razumeli problem koji je potpuno nov. Taj zadatak se retko svodi na jednostavno postavljanje pravih pitanja radi prikupljanja zahteva koje naručilac ima u svojoj glavi. U ranoj fazi projekta, zahtevi su nedovoljno formulisani i nedovoljno dobro shvaćeni. Naručilac ne može uvek, prilikom opisivanja, dovoljno precizno iskazati šta hoće ili šta mu stvarno treba, a mi nismo uvek u stanju da dovoljno dobro shvatimo tuđe poslovne probleme. Naručilac poznaje svoj posao, ali ne može uvek strancima da opiše svoje poslovne probleme. Njegovi opisi sadrže žargone i pretpostavke koje nam možda nisu bliski. Slično tome, kao projektantima, poznata su nam računarski zasnovana rešenja, ali ne uvek i to kako će moguća rešenja uticati na poslovne aktivnosti

naručioca. Mi, takođe, posedujemo vlastiti žargon i pretpostavke, mislimo da svi govore istim jezikom, a u stvari različiti ljudi podrazumevaju potpuno različita značenja iste reči. Do zajedničkog dogovora sa svima o tome koji su zahtevi, dolazimo jedino raspravom o zahtevima sa nekim ko je zainteresovan za sistem, objedinjavanjem tih različitih pogleda u koherentni skup zahteva, kao i zajednički pregled sa zainteresovanim subjektima dokumenata koji sadrže zahteve. (Alternativno gledište dato je u podsetniku 4.3.) Ako ne možemo da se usaglasimo oko zahteva, projekat je osuđen na propast.

PODSETNIK 4.2 AGILNO MODELOVANJE ZAHTEVA

Kao što smo napomenuli u poglavlju 2, analiza zahteva igra veliku ulogu kod odlučivanja da li da se koriste agilne metode kao osnova softverskog razvoja. Ako su zahtevi u čvrstoj sprezi i kompleksni, ili ako je verovatno da će budući zahtevi i proširenja prouzrokovati značajne izmene u arhitekturi sistema, tada je možda bolje da se upustimo u naporan proces koji apostrofira modelovanje unapred. Kod napornog procesa, razvojni tim odlaže kodiranje sve dok se ne modeluju i analiziraju zahtevi, formuliše arhitektura koja odražava zahteve i odaberе detaljni dizajn rešenja. Za svaki od tih koraka neophodan je model, a modeli su povezani i koordinisani sa ciljem da projektovano rešenje u celosti implementira formulisane zahteve. Taj pristup najviše odgovara razvoju uz oslonac na veliki tim, gde dokumentacija pomaže učesnicima u razvoju da sinhronizuju svoj rad, kao i kod sistema gde je bezbednost kritična, gde su ispravnost funkcionisanja i bezbednost sistema mnogo važniji nego datum njegovog završetka.

Međutim, u slučaju problema gde su zahtevi nedovoljno definisani, primena takvog pristupa unosi konfuziju, jer zahteva ažuriranje modela sa svakom izmenom zahteva. Kao alternativni pristup, agilne metode inkrementalno prikupljaju i implementiraju zahteve. Inicijalno izdanje implementira samo suštinske zahteve, kao što je definisano poslovnim ciljevima zainteresovanih subjekata. Kako, prilikom korišćenja sistema ili sa boljim razumevanjem problema, iskrsavaju novi zahtevi, oni se implementiraju u sledećim izdanjima sistema. Inkrementalni razvoj omogućava „ranu i kontinuiranu isporuku korisnog softvera" (Beck i dr. 2001) i prilagođava se hitnim ili zakasnelim zahtevima.

Ekstremno programiranje (XP) dovodi agilni proces definisanja zahteva do krajnosti, u tome da se sistem gradi prema zahtevima koji su trenutno definisani, bez planiranja ili projektovanja potencijalnih budućih zahteva. Povrh toga, XP se odriče tradicionalnog dokumentovanja zahteva i umesto toga kodira zahteve u vidu test-slučajeva koje eventualna implementacija mora da prođe. Berry (2002a) ukazuje na to da fleksibilnost agilnih metoda ima i stranu medalje, jer otežava unošenje izmena u sistem, koje su neophodne ako se želi dodavanje, brisanje ili izmena zahteva. Mogu se javiti i dodatni problemi. XP koristi test-slučajeve za specifikaciju zahteva, tako da loše napisan test-slučaj može dovesti do nesporazuma opisanih u ovom poglavlju.

PODSETNIK 4.3 UPOTREBA TAČKE GLEDIŠTA ZA RUKOVANJE NEDOSLEDNOSTIMA

Iako većina softverskih inženjera teži doslednim zahtevima, Easterbrook i Nuseibeh (1996) dokazuju da je često poželjno da se tolerišu i čak podstiču nedoslednost tokom postupka definisanja zahteva. Oni to tvrde jer je u ranoj fazi procesa definisanja zahteva besmisleno pokušavati da se razreše nedoslednosti, iz razloga što razumevanje date oblasti i potrebe vremenom evoluiraju kod zainteresovanih subjekata. Rano rešavanje nedoslednosti je skupo i često nepotrebno (a može i spontano da nastupi kako zainteresovani subjekti revidiraju svoja gledišta). Ono može biti takođe i kontraproduktivno ako se proces razrešavanja usmerava na postizanje dogovora umesto na suštinske uzroke nedoslednosti (na primer, zainteresovani subjekti ne razumeju datu oblast).

Umesto toga, Easterbrook i Nuseibeh predlažu da se kroz proces razvoja softvera pogledi zainteresovanih subjekata dokumentuju i održavaju kao odvojene **tačke gledišta** (Nuseibeh i dr. 1994). Analitičari zahteva definišu pravila doslednosti koja treba da se primene na tačke gledišta (tj. kako objekti, stanja i prelazi stanja su jedne tačke gledišta odgovaraju sličnim entitetima sa druge tačke gledišta; ili kako jedna tačka gledišta unapređuje drugu). Tačke gledišta se analiziraju (tako je moguće automatski) da bi se utvrdilo da li one poštuju pravila doslednosti. Ako se pravila narušavaju, nedoslednosti se evidentiraju kao deo tačaka gledišta, tako da drugi učesnici u razvoju softvera ne bi greškom implementirali osporeno gledište. Zapisane nedoslednosti se ponovo proveravaju svaki put kada se modifikuje tačka gledišta na koju su se one odnosile, da bi se videlo da li one u njima i dalje postoje. Takođe se i pravila doslednosti periodično proveravaju, da bi se utvrdilo da li su ona narušena evolucijom tačaka gledišta.

Ishod ovog pristupa je dokument sa zahtevima koji sadrži sve poglede svih zainteresovanih subjekata u svim trenucima. Nedoslednosti se naglašavaju ali se ne bavimo njima sve dok ne bude dovoljno informacija da se donese adekvatna odluka. Na taj način, izbegavamo da se prerano posvetimo zahtevima ili odlukama vezanim za dizajn.

Ko su u stvari zainteresovani subjekti? Ispostavlja se da ima više ljudi koji mogu dati doprinos zahtevima novog sistema:

- *Klijenti su oni koji finansiraju razvoj softvera*: Finansirajući razvoj, klijenti su na neki način, presudni zainteresovani subjekti, pa imaju konačnu reč o tome šta proizvod treba da radi (Robertson i Robertson 1999).
- *Kupci, su oni koji kupuju softver nakon što je razvijen*: Nekada su kupac i korisnik isti; u nekom drugom slučaju kupac je poslovni rukovodilac zainteresovan za poboljšanje produktivnosti svojih radnika. Moramo u dovoljnoj meri da razumemo potrebe kupaca, da bismo napravili proizvod koji će oni kupovati i smatrati korisnim.
- *Korisnici, su oni koji poznaju postojeći i koji će koristiti budući sistem*: Oni su stručni za rad postojećeg sistema, poznaju njegove najkorisnije odlike, kao i delove sistema koji zahtevaju poboljšanje. Možda ćemo želeti da se posavetujemo takođe i sa posebnim interesnim grupama korisnika, da bismo razumeli njihove posebne potrebe, kao što su korisnici sa invaliditetom, osobe kojima nisu bliski računari ili ih nerado koriste, stručni korisnici i drugi.

- *Stručnjaci iz konkretne oblasti primene, su oni kojima je problem koji softver treba da automatizuje blizak*: Na primer, konsultovaćemo stručnjaka za finansije ako gradimo finansijski paket, ili meteorologa ako softver treba da modeluje vremenske prilike. Te osobe mogu da učestvuju u definisanju zahteva, ili poseduju znanja o karakteristikama okruženja u kojima će proizvod funkcionisati.
- *Istraživači tržišta, koji sprovode istraživanja radi određivanja budućih trendova i potencijalnih potreba korisnika*: Oni mogu da pretpostave ulogu kupca ako se softver razvija za široko tržište, pri čemu nije identifikovan nijedan poseban kupac.
- *Advokati ili revizori, kojima su bliski vladini, bezbednosni i pravni zahtevi*: Na primer, možemo da se posavetujemo sa poreskim stručnjacima da bismo se osigurali da paket za obračun zarada poštuje poreske propise. Možemo takođe da konsultujemo stručnjake za standarde relevantne za funkcionisanje proizvoda.
- *Softverski inženjeri i drugi tehnolozi*: Ovi stručnjaci osiguravaju da je proizvod tehnički i ekonomski izvodljiv. Oni mogu da informišu naručioca o novim hardverskim i softverskim tehnologijama i preporuče nove funkcionalnosti koje koriste prednosti tih tehnologija. Oni mogu takođe da procene troškove i vreme potrebno za razvoj proizvoda.

Svi zainteresovani subjekti imaju poseban pogled na sistem i način na koji sistem treba da funkcioniše i ti pogledi su neretko protivrečni. Analitičari zahteva moraju da razumeju svaki pogled i da formulišu zahtev na način koji odražava interese svih učesnika. Na primer, naručilac može da specifikuje da sistem izvodi određeni zadatak, ali naručilac nije uvek i korisnik predloženog sistema. Korisnik možda hoće da zadatke izvodi na tri načina: u režimu učenja, u režimu početnika i u ekspertskom režimu. To razdvajanje omogućava korisniku da postepeno uči i ovladava sistemom. Mnogi sistemi za obradu teksta implementirani su na taj način, tako da se neiskusni korisnici mogu postepeno prilagoditi novom sistemu. Međutim, konflikti mogu da nastanu kada lakoća korišćenja podrazumeva sistem koji radi sporije nego što to dozvoljavaju zahtevi u pogledu vremena odziva.

Takođe, različiti učesnici mogu da očekuju različite nivoe detalja u dokumentaciji zahteva, a u tom slučaju zahtevi treba da budu različito prirеđeni za različite osobe. Pored toga, korisnici i učesnici u razvoju mogu da imaju predrasude (ispravne ili pogrešne) o tome šta druge grupe vrednuju i kako deluju. Tabela 4.1 zbirno prikazuje neke od uobičajenih stereotipa. Ova tabela naglašava ulogu koju interakcija među ljudima igra u razvoju softverskih sistema. Za dobrog analitičara zahteva bitni su izvrsni međuljudski odnosi, kao i solidne tehničke sposobnosti. Veb lokacija ove knjige sadrži sugestije za analizu svake od navedenih razlika u pogledima.

Uz ispitivanje zainteresovanih subjekata, dopunska sredstva za izvođenje zahteva obuhvataju:

- Pregledanje raspoložive dokumentacije, kao što su dokumentovane procedure ili ručni poslovi, kao i specifikacije ili korisnička uputstva automatizovanih sistema.
- Posmatranje postojećeg sistema (ako je to moguće), radi prikupljanja objektivnih informacija o tome kako korisnici izvode svoje zadatke i bolje razumevanje sistema koji treba automatizovati ili izmeniti. Često kada se razvije novi sistem koji je zasnovan na računarima, nastavlja se sa upotrebom starog sistema jer on obezbeđuje neke kritične funkcije koje su projektanti novog sistema prevideli.

Tabela 4.1 Kako jedni druge vide korisnici i projektanti (Scharer 1990)

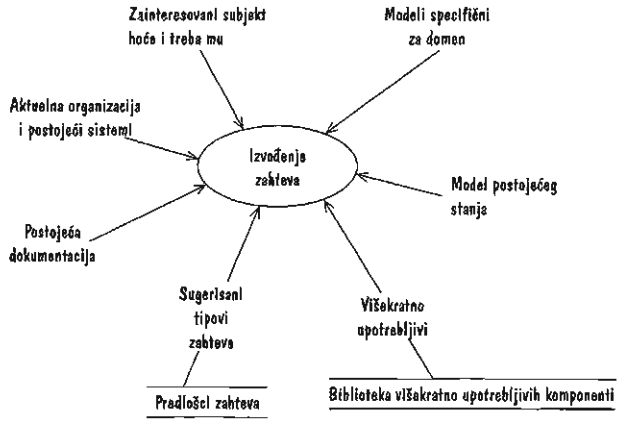
Kako projektanti vidi korisnike	Kako korisnici vide projektante
Korisnici ne znaju šta hoće.	Projektanti ne razumeju operativne potrebe.
Korisnici ne mogu da artikulišu ono šta hoće.	Projektanti ne mogu jasno da prevedu navedene potrebe u uspešan sistem.
Korisnici nisu sposobni da obezbede korisnu formulaciju svojih potreba.	Projektanti postavljaju nerealne standarde za definisanje zahteva.
Korisnici imaju suviše potreba koje su politički motivisane.	Projektanti stavljaju veliki naglasak na tehnička pitanja.
Korisnici sve hoće odmah.	Projektanti uvek kasne.
Korisnici ne mogu da prate rokove.	Projektanti ne mogu brzo da odgovore na legitimno izmenjene potrebe.
Korisnici ne umeju da dodele prioritete svojim potrebama.	Projektanti uvek premašuju budžet.
Korisnici nisu radi da prave kompromise.	Projektanti sve vreme govore „ne".
Korisnici odbijaju da preuzmu odgovornost za sistem.	Projektanti pokušavaju da nam kažu kako da radimo naš posao.
Korisnici nisu posvećeni razvojnim projektima.	Projektanti od korisnika zahtevaju vreme i trud, čak i na uštrb primarnih zaduženja korisnika koja su njemu važnija.

- Učenje zanata od korisnika (Beyer i Holtzblatt 1995), detaljnije učenje o poslovima koje korisnik obavlja dok ih ovaj izvršava.
- Intervjuisanje korisnika ili zainteresovanih subjekata u grupama, tako da oni budu inspirisani idejama drugih.
- Upotreba strategija specifičnih za datu oblast, kao što je tzv. zajednički razvoj aplikacije (*Joint Application Design*) (Wood i Silver 1995) ili PIECES (Wetherbe 1984) kod informacionih sistema. Tako se obezbeđuje da zainteresovani subjekti vode računa o specifičnim tipovima zahteva koji su relevantni za njihovu posebnu situaciju.
- Razmatranje u nevezanoj raspravi sa aktuelnim i potencijalnim korisnicima o tome kako da se poboljša predloženi proizvod.

Model Volere (Robertson i Robertson 1999), prikazan na slici 4.2, za definisanje zahteva predlaže i dodatne izvore, kao što su predlošci i biblioteke zahteva iz sličnih, ranije razvijenih, sistema .

4.3 TIPOVI ZAHTEVA

Kada većina ljudi razmišlja o zahtevima, oni razmišljaju o traženoj *funkcionalnosti*: Koje usluge treba da se obezbede? Koje operacije treba da se izvode? Šta treba da bude reakcija na određeni stimulans? Kako se zahtevano ponašanje menja u funkciji vremena i reakcija na prethodne događaje? **Funkcionalni zahtev** opisuje obavezno ponašanje u funkciji neophodnih aktivnosti, kao što su: reakcije na ulaze i stanja svakog entiteta pre pojave i nakon okončanja svake aktivnosti. Na primer, u slučaju sistema za obračun zarada, funkcionalni zahtevi definišu: koliko često se izdaju platni listići, koji ulaz je neophodan da bi se odštampao platni listić, pod kojim uslovima može da se promeni iznos za isplatu, kao i koji su razlozi uklanjanja radnika sa platnog spiska.



SLIKA 4.2 Izvori mogućih zahteva (Robertson i Robertson 1999)

Funkcionalni zahtevi definišu granice koje omeđuju prostor rešenja našeg problema. Prostor rešenja predstavlja skup mogućih načina izrade softvera koji implementira specifikovane zahteve. Inicijalno, taj skup može da bude vrlo velik. Međutim, u praksi obično nije dovoljno da softverski proizvod ispravno izračunava izlazne podatke, jer postoje i drugi tipovi zahteva na osnovu kojih je moguće jasno razlikovati prihvatljive proizvode od neprihvatljivih. **Zahtev u pogledu kvaliteta**, ili **nefunkcionalni zahtev**, opisuje neke karakteristike kvaliteta koje softversko rešenje mora da poseduje, kao što je, na primer, kratko vreme odziva, lakoća korišćenja, visoka pouzdanost ili niski troškovi održavanja. **Projektno ograničenje** predstavlja unapred donetu odluku, na primer, izbor platforme ili interfejs komponenti, koja ograničava skup mogućih rešenja razmatranog problema. **Procesno ograničenje** predstavlja ograničenje koje se odnosi na tehnike ili resurse koji se mogu koristiti u izgradnji sistema. Na primer, naručilac može da insistira na primeni agilnih metoda, da bi mogao operativno koristiti rane verzije sistema uz kontinuirano proširivanje njihove funkcionalnosti. Prema tome, zahtevi u pogledu kvaliteta, projektna i procesna ograničenja dodatno sužavaju prostor rešenja, praveći razliku između prihvatljivih i dopadljivih rešenja i neupotrebljenih proizvoda. Tabela 4.2 sadrži primere zahteva različitog tipa.

Zahtevi u pogledu kvaliteta ponekad zvuče kao „majčinske" karakteristike koje svi proizvodi treba da poseduju. Nakon svega, ko bi zahtevao softverski sistem koji je spor, neprijateljski, nepouzdan i težak za održavanje. Bolje je zahteve u pogledu kvaliteta posmatrati kao kriterijum projektovanja koji može da se optimizuje i koristi za izbor između različitih načina za implementaciju funkcionalnih zahteva. Ako usvojimo takav pristup, tada zahtevi treba da daju odgovor na sledeće pitanje: Do kog stepena mora proizvod da zadovolji zahteve u pogledu kvaliteta da bi bio prihvatljiv? Podsetnik 4.4 objašnjava kako treba formulisati zahteve u pogledu kvaliteta, da bismo testiranjem mogli utvrditi da li su oni dostignuti.

Rešavanje konflikata

Prilikom pokušaja iskazivanja svih vrsta zahteva relevantnih zainteresovanih subjekata, sigurno ćemo se sresti sa suprotstavljenim idejama o tome kakvi zahtevi treba da budu. Obično je korisno tražiti da naručilac odredi prioritete zahteva. Taj zadatak primorava naručioca da ukaže na najvažnije neophodne osobine ili usluge. Okvirna šema prioriteta mogla bi da klasifikuje zahteve u tri kategorije:

Tabela 4.2 Pitanja za razgraničavanje različitih tipova zahteva

Funkcionalni zahtevi

Funkcionalnost

- Šta će sistem da radi?
- Kada će sistem to da radi?
- Da li postoji više načina rada?
- Koje vrste proračuna ili transformacija podataka moraju da se izvedu?
- Koje su odgovarajuće reakcije na moguće stimulanse?

Podaci

- Kakav treba da bude format ulaznih i izlaznih podataka?
- Da li neki podaci moraju da budu sačuvani u nekom vremenskom periodu?

Projektna ograničenja

Fizičko okruženje

- Gde će biti locirana oprema?
- Da li postoji jedna ili više lokacija?
- Da li postoje neka ograničenja u pogledu okruženja, kao što je temperatura, vlažnost ili magnetna interferencija?
- Da li postoje neka ograničenja u pogledu veličine sistema?
- Da li postoje ograničenja u pogledu napajanja, grejanja, ili klimatizacije?
- Da li postoje ograničenja u pogledu programskih jezika zbog postojećih softverskih komponenti?

Interfejsi

- Da li se ulaz preuzima iz jednog sistema ili iz više različitih sistema?
- Da li se izlaz prosleđuje jednom sistemu ili većem broju različitih sistema?
- Da li je propisan način formatizovanja ulaznih i izlaznih podataka?
- Da li je propisan nosilac podataka?

Korisnici

- Ko će da koristi sistem?
- Da li će biti više vrsta korisnika?
- Koji je nivo veštine svakog korisnika?

Procesna ograničenja

Resursi

- Koji su materijali, osoblje ili drugi resursi potrebni za gradnju sistema?
- Koje veštine moraju posedovati učesnici u razvoju?

Dokumentacija

- Koliko je dokumentacije potrebno?
- Treba li da bude na računaru, u obliku knjige ili oboje?
- Kome je namenjen svaki tip dokumentacije?

Standardi

Zahtevi u pogledu kvaliteta

Performansa

- Da li postoje ograničenja u pogledu brzine izvršavanja, vremena odgovora ili propusnosti?
- Čime će se meriti efikasnost korišćenja resursa i vreme odgovora?
- Koliki je protok podataka kroz sistem?
- Koliko se često podaci primaju i šalju?

Upotrebljivost i ljudski faktori

- Koja vrsta obuke će biti potrebna za svaki tip korisnika?
- Koliko će lako korisnik da razume i upotrebljava sistem?
- Koliko će teško korisnik da na neodgovarajući način koristi sistem?

Bezbednost

- Da li pristup sistemu ili informacijama mora da bude kontrolisan?
- Da li podaci jednog korisnika moraju da budu izolovani od podataka drugih korisnika?
- Da li korisnički programi moraju da budu izolovani od drugih programa ili od operativnog sistema?
- Da li treba da se preduzmu mere obezbeđenja od krađe ili vandalizma?

Pouzdanost i raspoloživost

- Da li sistem mora da detektuje i izoluje greške?
- Koje je propisano srednje vreme između otkaza?
- Da li postoji maksimalno dopušteno vreme za ponovno pokretanje sistema nakon otkaza?
- Koliko često se prave rezervne kopije?
- Da li se rezervne kopije moraju čuvati na drugoj lokaciji?
- Da li moraju da se preduzmu mere obezbeđenja od požara ili poplave?

Mogućnost održavanja

- Da održavanje uključuje samo ispravljanje grešaka ili poboljšanja sistema?
- Kada i na koje načine može sistem da se izmeni u budućnosti?
- Koliko će biti lako dodavanje novih svojstava u sistem?
- Koliko će biti lako prenošenje sistema sa jedne platforme (računar, operativni sistem) na drugu?

Preciznost i tačnost

- Koliko tačne moraju da budu izračunate vrednosti podataka?
- Do kog stepena preciznosti moraju da se obavljaju proračuni?

Vreme do isporuke/cena

- Da li postoji definisano vreme potrebno za razvoj?
- Da li postoji ograničenje iznosa novca koji se može potrošiti na razvoj, hardver ili softver?

PODSETNIK 4.4 OMOGUĆAVANJE TESTIRANJA ZAHTEVA

Pišući o dobrom projektovanju, Alexander (1979) nas ohrabruje da omogućimo testiranje zahteva. Pod time on podrazumeva da, jednom kada je zahtev postavljen, moramo biti u stanju da procenimo da li ga predloženo rešenje zadovoljava. To procenjivanje mora da bude objektivno, tj. zaključak o tome da li je zahtev zadovoljen ne sme da zavisi od toga ko vrši procenu.

Robertson i Robertson (1997) ukazuju da mogućnost testiranja (koju oni nazivaju „merljivost") može da se razmatra prilikom iskazivanja zahteva. Cilj je kvantifikacija stepena do kojeg svaki zahtev treba da bude zadovoljen. Ovaj **kriterijum uskladenosti** formira objektivne standarde za prosuđivanje da li predloženo rešenje zadovoljava zahteve. Kada se takav kriterijum ne može lako definisati, tada je zahtev verovatno višeznačan, nekompletan ili neispravan. Na primer, naručilac može da iskaže zahtev u pogledu kvaliteta na sledeći način:

Informacija o kvalitetu vode mora da bude dostupna trenutno.

Kako možemo da testiramo da li proizvod zadovoljava zahtev? Naručilac verovatno ima jasnu ideju o tome šta znači izraz „trenutno", pri čemu taj pojam mora da se opiše u zahtevima. Treba da preciznije definišemo šta znači pojam „trenutno":

Informacija o kvalitetu vode mora da se dobije u roku od 5 sekundi od postavljanja upita.

Ova druga formulacija zahteva može objektivno da se testira, jer ako, na primer, napravimo niz upita, tada možemo da proverimo da li sistem obezbeđuje odgovarajući rezultat u roku od 5 sekundi od postavljanja svakog upita.

Relativno je lako odrediti kriterijum uskladenosti za zahteve u pogledu kvaliteta koji su kvantitativne prirode (na primer performansa, veličina, preciznost, tačnost, vreme do isporuke). Šta je sa subjektivnijim zahtevima kao što su upotrebljivost ili mogućnost održavanja? U tim slučajevima, razvojni tim koristi fokus grupe ili metriku za procenjivanje kriterijuma uskladenosti:

- 75% korisnika treba da oceni da li je novi sistem jednako upotrebljiv kao i postojeći.
- nakon obuke, 90% korisnika treba da bude u stanju da obradi novi nalog u roku od 5 minuta;
- moduli će enkapsulirati najviše jedan tip podataka;
- greške u promenama treba da se isprave u roku od 3 nedelje od trenutka prijavljivanja.

Kriterijumi uskladenosti koje nije moguće oceniti pre isporuke konačnog proizvoda su mnogo zahtevniji:

- Sistem neće biti van funkcije više od ukupno 5 minuta u toku jedne godine.
- Srednje vreme između dva otkaza neće biti krače od godinu dana.

U tim slučajevima postoje dva rešenja: možemo proceniti atribute kvaliteta sistema (postoje tehnike za procenu pouzdanosti sistema i broja grešaka po redu koda), ili možemo oceniti isporučeni sistem tokom njegovog rada i pri tome prihvatiti plaćanje penala ako sistem ne funkcioniše na obećani način.

Zanimljivo je da sve ono što može da se meri biva i urađeno. Drugim rečima, kriterijum će verovatno biti ispunjen, osim ako je nerealan. Ključno je da se zajedno sa naručiocem odredi pre svega

način na koji će se utvrditi da li isporučeni sistem zadovoljava postavljene zahteve. Robertsonovi sugerišu tri načina koji omogućavaju testiranje zahteva:

- Odrediti kvantitativni opis svih priloga za način i prideva, tako da značenje kvalifikatora bude jasno i nedvosmisleno.
- Umesto zamenica koristiti nazive entiteta.
- Proveriti da li je svaka imenica definisana samo jednom u specifikaciji zahteva.

Alternativni prilaz, koga zastupa škola QFD (*Quality Function Deployment*) (Akao 1990), sastoji se u predstavljanju zahteva u pogledu kvaliteta kao funkcionalnih zahteva posebne namene i svođenju testiranja zahteva u pogledu kvaliteta na testiranje njima pridruženih funkcionalnih zahteva. Ovaj pristup je pogodniji za neke zahteve u pogledu kvaliteta, dok za druge nije. Na primer, zahtevi u vezi sa radom u realnom vremenu mogu da se opišu kao dodatni uslovi ili ograničenja u pogledu dešavanja zahtevane funkcionalnosti. Drugi zahtevi u pogledu kvaliteta, kao što su bezbednost, mogućnost održavanja i performansa, mogu da se zadovolje usvajanjem postojećih projektnih rešenja i protokola koji su posebno razvijeni radi optimizacije određenih zahteva u pogledu kvaliteta.

1. zahtevi koji apsolutno moraju da se zadovolje (*suštinski* zahtevi);
2. zahtevi koji su veoma poželjni, ali nisu nužni (*poželjni* zahtevi);
3. zahtevi koji se mogu ispuniti ali se mogu i izostaviti (*opcioni* zahtevi).

Na primer, sistem za naplatu preko kreditnih kartica mora da bude sposoban da prikaže spisak trenutnih dugovanja, da ih sabere i zahteva plaćanje do određenog datuma; to su *suštinski* zahtevi. Ali sistem za naplatu može takođe da razdvoji dugovanja po načinu naručivanja, da bi naručiocu pomogao u razumevanja šablona za kupovinu. Analiza načina naručivanja je *poželjan* zahtev, koji verovatno nije neophodan. Na kraju, sistem može da štampa potraživanja u crnoj, a dugovanja u crvenoj boji, što bi bilo korisno, ali je *opciono*. Dodeljivanje prioriteta zahtevima njihovom kategorizacijom pomaže svim učesnicima da bolje shvate šta je stvarno potrebno. To je posebno korisno ako je projekat razvoja softvera ograničen sa aspekta vremena ili drugih resursa. Ako će definisani sistem biti skup, ili zahtevati mnogo vremena za razvoj, tada opcioni zahtevi mogu da se izostave, a poželjni analiziraju sa ciljem eliminisanja ili odlaganja realizacije za kasnije verzije.

Dodela prioriteta može da posebno pomogne prilikom razrešenja konflikata između zahteva u pogledu kvaliteta. Često su dešava da su dva atributa kvaliteta u konfliktu tako da je nemoguće optimizacijom oba zadovoljiti. Na primer, pretpostavimo da se od sistema zahteva da se lako održava i da ima kratko vreme odziva. Projektno rešenje koje olakšava održavanja uz oslonac na razdvajanje nadležnosti i enkapsulaciju, može da radi sporije čime se narušava performansa. Slično tome, fino podešavanje sistema tako da radi posebno dobro na jednoj platformi, utiče na prenosivost na druge platforme, dok bezbednosni sistemi nužno kontrolišu pristup i time ograničavaju raspoloživost nekim

korisnicima. Naglasak na bezbednosti, pouzdanosti, robusnosti, upotrebljivosti ili performansama utiče na mogućnost održavanja, budući da uključivanje tih karakteristika uvećava složenost rešenja i umanjuje njegovu koherentnost. Dodeljivanje prioriteta zahtevima u pogledu kvaliteta primorava naručioca da odabere one faktore kvaliteta softvera koji su njemu najvažniji, što nam pomaže da obezbedimo razumno, ako ne i optimalno, rešenje za postavljene zahteve u pogledu kvaliteta.

Možemo takođe da izbegnemo optimizaciju više suprotstavljenih zahteva u pogledu kvaliteta, identifikujući i zadovoljavajući kriterijum usklađenosti, koji uspostavlja jasne testove prihvatanja za te zahteve (pročitajte podsetnik 4.4). Šta ako ne možemo da zadovoljimo kriterijum usklađenosti? Onda je možda došao trenutak za pregovaranje i ponovno ocenjivanje pogleda svih zainteresovanih subjekata. Pregovaranje nije lako, jer zahteva sposobnost, strpljenje i iskustvo u pronalaženju uzajamno prihvatljivih rešenja. Srećom, zainteresovani subjekti se retko razilaze u shvatanju koji problem rešava projektovani softverski sistem. Sukobi se obično javljaju oko različitih pristupa ili ograničenja vezanih za rešavanje problema (na primer, neki zainteresovani subjekti mogu da insistiraju na upotrebi drugačijih sistema za rukovanje bazama podataka, drugih algoritama za šifriranje, drugačijih korisničkih interfejsa ili drugih programskih jezika). Još je ozbiljnije ako se zainteresovane strane ne slažu oko prioritetnih zahteva, ili poslovnih pravila koja treba da budu ugrađena u sistem. Na primer, univerzitetski odseci mogu da zahtevaju različita pravila ocenjivanja studenata u svojim programima, dok bi administracija univerziteta radije želela ujednačenost i jednoobraznost. Rešavanje podrazumeva tačno utvrđivanje razloga nepopustljivosti zainteresovanih subjekata u vezi sa nekim pristupom, pravilom ili nivoom prioriteta (na primer, možda su posebno zainteresovani za drugačije regulisanje troškova, bezbednosti, brzine ili kvaliteta), a tada je neophodno poraditi na postizanju dogovora o fundamentalnim zahtevima. Uz efikasno pregovaranje, zainteresovani subjekti će bolje razumeti i više ceniti fundamentalne potrebe drugih, pa će težiti rešenjima koja će zadovoljiti sve strane. Takva rešenja se obično značajno razlikuju od originalnih pogleda koje su na početku imali zainteresovani subjekti.

Dve vrste dokumenata o zahtevima

Zahteve koriste različiti ljudi za različite namene. Analitičari zahteva i njihovi klijenti koriste zahteve za iskazivanje stepena razumljivosti ponašanja sistema. Projektanti tretiraju zahteve kao ograničenja vezana za utvrđeno prihvatljivo rešenje. Tim za testiranje na osnovu zahteva formuliše grupu *završnih testova*, koji će se koristiti da bi se naručiocu pokazalo da mu se isporučujesistem koji je i naručen. Zahtevi su pomoć timu za održavanje, na osnovu koje se ovaj tim stara da unapređenja sistema (ispravke grešaka, nove osobine) ne utiču na originalnu namenu sistema. Ponekad samo jedan dokument može da posluži za sve navedene namene i time dovede do uzajamnog razumevanja naručioca, analitičara zahteva i razvojnog tima. Često su međutim potrebna dva dokumenta: *definicija zahteva* namenjena poslovnom auditorijumu, kao što su kupci, naručioci i korisnici i *specifikacija zahteva* namenjena tehničkom auditorijumu, kao što su projektanti, timovi za testiranje i rukovodioci projekata.

Tu razliku ćemo ilustrovati na jednostavnom primeru koji su prikazali Jackson i Zave (1995). Razmotrimo softverski kontrolisanu obrtnu kapiju koja je postavljena na ulazu zoološkog vrta. Kada se u kapiju ubaci novčić, ona se otključava, omogućavajući posetio-

cu da je gurne i uđe u zoološki vrt. Kada se obrtna rampa dovoljno zarotira da omogući jedan prolaz, ona se ponovo zaključava i sprečava da sledeća osoba uđe bez plaćanja.

Definicija zahteva predstavlja kompletan spisak svega što naručilac želi da postigne. Dokument sadrži neophodne zahteve, opis entiteta koji pripadaju okruženju u kojem će predloženi sistem biti instaliran, kao i željena ograničenja u vezi sa entitetima, u vezi sa nadgledanjem tih entiteta, ili u vezi sa transformacijom tih entiteta. Svrha predloženog sistema jeste realizacija tih zahteva (Zave i Jackson 1997). Prema tome, zahtevi su potpuno iskazani u smislu okruženja, i opisuju kako će predloženi sistem uticati na okruženje. Primer obrtne kapije postavlja dva zahteva: (1) niko ne treba da uđe u zoološki vrt bez plaćanja ulaza; (2) sistem ne treba da sprečava svaki plaćeni ulaz[1]. Definiciju zahteva obično zajedno pišu naručilac i analitičar, i ona predstavlja ugovor koji sadrži opis funkcionalnosti koju razvojni tim prihvata da isporuči naručiocu.

Specifikacija zahteva iskazuje zahteve u vidu specifikacije ponašanja predloženog sistema. Specifikacija je takođe iskazana u smislu okruženja, s tom razlikom što referencira jedino one entitete kojima može da se pristupi iz sistema posredstvom njegovog interfejsa. Na taj način, granica sistema čini eksplicitnim one entitete iz okruženja u koje sistem ima uvid ili koje direktno kontroliše. Ta razlika je prikazana na slici 4.3, gde su zahtevi definisani bilo gde u domenu okruženja, uključujući i interfejs sistema, i gde je specifikacija ograničena jedino na presek domena okruženja i domena sistema. Da bismo uočili razliku, posmatrajmo zahtev da niko ne može da uđe u zoološki vrt bez plaćanja ulaznice. Ako obrtna rampa poseduje otvor za kovani novac i može da detektuje ubacivanje ispravnog apoena, moguće je utvrditi da li je ulaznica plaćena. Nasuprot tome, koncept samog događaja ulaska može da bude izvan okvira sistema. Zato se zahtev mora ponovo opisati da bi se događaj ulaska realizovao isključivo na osnovu događaja i stanja koje obrtna rampa *može* da detektuje i kontroliše: da li je rampa otključana i da li detektuje pritisak na kapiju prilikom ulaska posetioca:

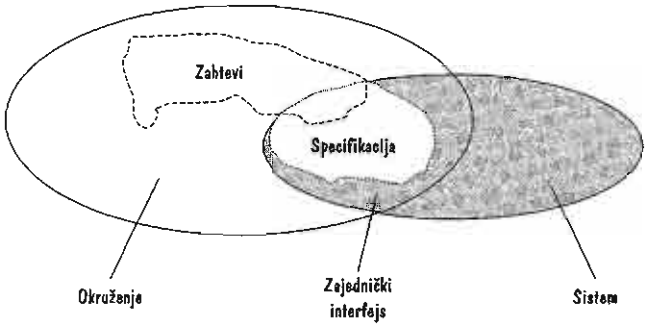
> *Kada posetilac primeni određeni nivo sile na otključanu obrtnu kapiju, rampa se automatski rotira za jedan polukrug i završava u poziciji zaključano.*

Na taj način, specifikacija u stvari predstavlja rafinisanu definiciju originalnog zahteva.

Specifikaciju zahteva piše analitičar, a koriste je ostali učesnici u razvoju softvera. Analitičar mora obezbediti da ne dođe do gubitka informacije ili izmena u postupku pretvaranja zahteva u specifikaciju. Mora se uspostaviti jednoznačna korenspondencija između svakog zahteva koji pripada dokumentu definicije zahteva i zahteva u specifikacionom dokumentu.

---

1 Još intuitivniju predstavu drugog zahteva da svakome ko plati treba dozvoliti da uđe u zoološki vrt, jednostavno nije moguće realizovati. Ne postoji način da sistem spreči da neki spoljašnji činilac onemogući ulazak u zoološki vrt posetiocu koji je platio ulaznicu. Na primer, drugi posetilac može da prođe kroz otključanu obrtnu kapiji pre posetioca koji je platio, zoološki vrt može da se zatvori pre nego što kroz kapiju prođe posetilac koji je platio, posetilac može da odluči da ode pre ulaska itd. (Jackson i Zave 1995).



SLIKA 4.3 Zahtevi prema specifikaciji

4.4 KARAKTERISTIKE ZAHTEVA

Da bi krajnji proizvod bio uspešan, važno je da zahtevi budu visoko kvalitetni, tj. ono što nije specifikovano, obično se i ne realizuje. Kasnije u ovom poglavlju razmatramo način validacije i verifikacije zahteva. U međuvremenu ćemo navesti poželjne osobine koje će biti predmet kasnije provere.

1. *Da li su zahtevi ispravni?* I mi i naručilac treba da pregledamo dokumentovane zahteve, da bismo se uverili da li su oni usklađeni sa našim shvatanjem zahteva.
2. *Da li su zahtevi dosledni?* Da li postoje suprotstavljeni zahtevi? Na primer, ako jedan zahtev navodi da maksimalno 10 korisnika može da koristi sistem u jednom trenutku, a drugi da u određenim situacijama može da postoji 20 istovremenih korisnika, tada se za ta dva zahteva kaže da su nedosledni. U opštem slučaju, dva zahteva su **nedosledna** ako ih je nemoguće istovremeno zadovoljiti.
3. *Da li su zahtevi nedvosmisleni?* Zahtevi su višeznačni ako više osoba koje analiziraju dokument sa zahtevima dođe do različitih ali ispravnih interpretacija. Pretpostavimo da kupac sistema za upravljanje satelitom zahteva tačnost koja je dovoljna za planiranje misije. Zahtevi nam ne govore kakva je podrška potrebna za planiranje misije. Naručilac i razvojni tim mogu da imaju veoma različite poglede na stepen tačnosti koji je potreban. Dalje razmatranje značenja pojma „planiranje misije" može da rezultuje postavljanjem još preciznijeg zahteva: „Prilikom identifikacije položaja satelita, poziciona greška treba da bude manja od 15,24 metara duž orbite, a manja od 9,144 metara van orbite". Ako dobijemo ovaj detaljniji zahtev, moguće je testirati pozicionu grešku i tačno utvrditi da li postavljeni zahtev jeste ili nije zadovoljen.
4. *Da li su zahtevi kompletni?* Skup zahteva je **kompletan** ako specifikuje zahtevano ponašanje i izlaz za sve moguće kombinacije ulaza u svim mogućim stanjima i pod svim mogućim ograničenjima. Za sistem za obračun zarada neophodno je definisati šta se događa kada radnik uzme neplaćeno odsustvo, dobije povišicu, ili traži akontaciju. Kažemo da su zahtevi **eksterno kompletni** ako su sva stanja, promene stanja, ulazi, proizvodi i ograničenja opisani u sklopu nekog zahteva. Opis zahteva je **interno kompletan** ako ne postoje nedefinisani pojmovi unutar zahteva.

5. *Da li su zahtevi izvodljivi?* Da li uopšte postoji rešenje za potrebe naručioca? Na primer, pretpostavimo da naručilac želi da korisnici mogu da pristupe glavnom računaru koji je lociran više hiljada kilometara daleko i da vreme odziva za udaljene korisnike bude jednako kao i za lokalne korisnike (čije su radne stanice direktno povezane sa glavnim računarom). Pitanja izvodljivosti često iskrsavaju kada naručilac postavlja dva ili više zahteva u pogledu kvaliteta, kao što je jeftin sistem koji analizira ogromne količine podataka i produkuje rezultate analize u nekoliko sekundi.
6. *Da li je svaki zahtev relevantan?* Ponekad zahtevi nepotrebno ograničavaju razvojni tim ili uključuju funkcionalnosti koje nisu u direktnoj vezi sa potrebama korisnika. Na primer, general može da odluči da novi softverski sistem tenka treba da omogući vojnicima da šalju i primaju elektronsku poštu, iako je osnovna namena tenka da savladava neravan teren. Treba da nastojimo da tu „eksploziju mogućnosti" držimo pod kontrolom i pomognemo zainteresovanim subjektima da se usredsrede na suštinske i poželjne zahteve.
7. *Da li je zahteve moguće testirati?* Zahtevi se mogu **testirati** ako je moguće formulisati testove koji jasno demonstriraju da li konačni proizvod zadovoljava te zahteve. Razmotrimo kako možemo da testiramo zahtev da sistem obezbeđuje odgovore na upit u realnom remenu. Mi ne znamo šta je to „odgovor u realnom vremenu". Međutim, ako je definisan kriterijum koji kaže da sistem treba da odgovori na upit u roku ne dužem od dve sekunde, onda tačno znamo kako da testiramo reagovanje sistema na upite.
8. *Da li zahtevi mogu da se prate?* Da li su zahtevi organizovani i jedinstveno označeni u cilju lakšeg referenciranja? Da li svaka stavka u definiciji zahteva ima odgovarajuću stavku u specifikaciji zahteva, i obrnuto?

Ove karakteristike možemo da posmatramo kao funkcionalne zahteve i zahteve u pogledu kvaliteta koji prate skup zahteva vezanih za proizvod. One mogu da nam pomognu da odredimo kada smo prikupili dovoljno informacija, a kada je potrebno da saznamo više o značenju određenog zahteva. Stepen do kojeg želimo da ih zadovoljimo direktno utiče na sveobuhvatnost rešenja i tip informacija koje sakupljamo u procesu iskazivanja zahteva, kao i na izbor jezika za specifikaciju zahteva i provere koje je potrebno izvršiti u cilju validacije i verifikacije zahteva.

4.5 NOTACIJE MODELA

Karakteristika svake inženjerske discipline je ponovljivost, kao na primer, u slučaju tehnika za razvoj uspešnih i bezbednih proizvoda predstavljenih u poglavlju 2. Druga osobina je postojanje standardne notacije za modelovanje, dokumentovanje i razmenu odluka. Modelovanje može da nam pomogne da temeljno razumemo zahteve, sugerišući pitanja koja je potrebno postaviti. Praznine u sklopu modela ukazuju na nepoznato ili višeznačno ponašanje. Višestruki, suprotstavljeni izlazi, za isti ulaz, otkrivaju nedoslednosti u zahtevima. Kako se model razvija, ono šta sami ne poznajemo, i ono šta ne zna naručilac postaje sve očiglednije. Model nije moguće kompletirati bez razumevanja predmeta koji

se modeluje. Ponovnim navođenjem zahteva u potpuno različitoj formi u odnosu na originalne zahteve primoravamo naručioca da pažljivo ispita naš model u cilju validacije njegove ispravnosti.

Ako analiziramo literaturu, videćemo da, na prvi pogled, postoji gotovo beskonačan skup notacija i metoda za specifikaciju, i da se stalno na tržište izbacuju nove. Ali ako odemo korak nazad i zanemarimo detalje, primetićemo da su mnoge notacije slične. Uprkos brojnosti notacija, verovatno postoji manje od deset osnovnih paradigmi za iskazivanje koncepata, ponašanja i osobina problema.

Ovaj odeljak se fokusira na sedam osnovnih notacionih platformi, koje mogu da se na različite načine primene u sklopu pojedinačnih koraka u procesu razvoja. Razmatranje svake od njih započinjemo opisom platforme, tipova problema i opisa za koje je posebno pogodna. Nakon toga navodimo par konkretnih primera notacija za određene platforme. Kada upoznamo platforme, učenje i korišćenje nove notacije postaje jednostavnije zbog razumevanja njenih odnosa sa postojećim notacijama.

Međutim, savetuje se oprez. Potrebno je da budemo posebno oprezni sa terminologijom koju koristimo prilikom modelovanja zahteva. Većina notacija koje se koriste za specifikovanje zahteva, zasnovana je na uspešnim projektantskim metodama i notacijama, što znači da su referentne notacije primerene dizajnu a ne zahtevima, i upućuju na postupak dizajnom orijentisanog modelovanja. Odlučivanje u procesu formulisanja zahteva zasnovano je na različitim motivima pa se terminologija različito tumači. Na primer, prilikom modelovanja zahteva raspravljamo o *raščlanjivanju*, *apstrakciji* i *razdvajanju nadležnosti*, što su u stvari izvorne tehnike za projektovanje modularnih rešenja. Dekompoziciju specifikacije zahteva vršimo u skladu sa razdvojenim nadležnostima radi pojednostavljenja rezultujućeg modela, čineći ga na taj način razumljivijim i čitljivijim. Nasuprot tome, dekompoziciju dizajna vršimo radi poboljšanja kvaliteta atributa sistema (modularnost, mogućnost održavanja, performansa, rok isporuke itd). Zahtevi imenuju i ograničavaju te atribute, dok dekompozicija nema uticaja na taj aspekt specifikacije. Prema tome, iako koristimo izraze *dekompozicija* i *modularnost* i u specifikaciji i u dizajnu, odluke koje donosimo u pogledu dekompozicije u svakoj od faza su različite jer su im ciljevi različiti.

U sklopu ovog odeljka ilustrovaćemo pojedinačne notacije modelovanjem različitih aspekata prethodno opisanog problema obrtne rampe (Jackson i Zave 1995), i pogledaćemo jednu biblioteku. Biblioteka vodi evidenciju o tekstovima i drugim materijalima, njihovom izdavanju kao i druge informacije o članovima. Popularni naslovi se *rezervišu*, što znači da su za njih periodi izdavanja kraći nego kod drugih knjiga i materijala, a kazne za kašnjenje su veće nego za one koji ne podležu rezervisanju.

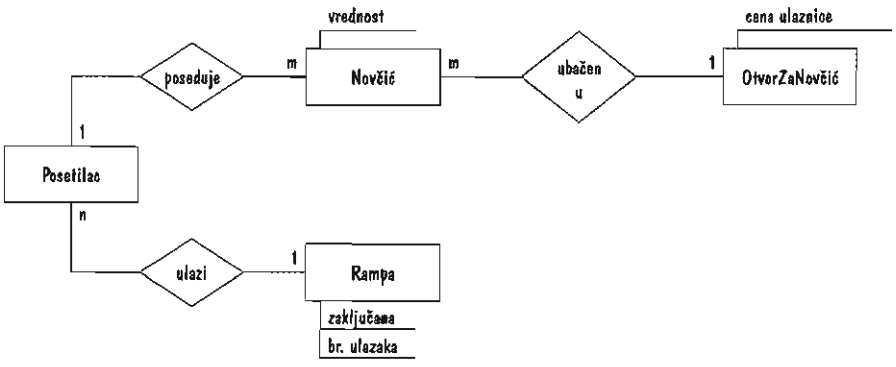
Dijagrami odnosa između entiteta

U ranoj fazi definisanja zahteva korisno je izgraditi konceptualni model problema koji identifikuje relevantne objekte ili entitete, njihov izgled (definisanjem njihovih atributa) i međusobne odnose. Takav model imenuje osnovne elemente problema. Oni se zatim ponovo koriste u drugim opisima zahteva (možda uz oslonac na druge notacije) koji definišu kako će se objekti, njihovi atributi i međusobni odnosi menjati tokom funkci-

onisanja predloženog sistema. Prema tome, konceptualni model pomaže da se različiti pogledi na zahteve i različiti opisi zahteva povežu.

Dijagram odnosa između entiteta ili ER dijagram (*Entity-Relationship diagram*) (Chen 1976) je popularna grafička notaciona platforma za predstavljanje konceptualnih modela. Kao što ćemo videti u poglavlju 6, oni predstavljaju osnovu većine objektno orijentisanih notacija za predstavljanje zahteva i dizajna, gde se ona koristi za modelovanje odnosa između objekata iz domena problema ili za modelovanje strukture softverskog proizvoda. Ta notacija je takođe popularna za opisivanje šeme baze podataka (tj. za opisivanje logičke strukture zapisa smeštenih u bazi podataka).

ER dijagrami se zasnivaju na tri osnovna elementa: entitet, atribut i odnos (relacija), koji se kombinuju radi specifikovanja elemenata problema i njihovih veza. Na slici 4.4 prikazan je ER dijagram obrtne rampe. Entitet, predstavljen pravougaonikom, je skup (ponekad ga nazivamo klasa) objekata iz realnog sveta koji imaju zajedničke osobine i ponašanja. Na primer, u svetu postoji mnogo `Novčića` ali za potrebe modelovanja obrtne rampe, sve `Novčiće` smatramo identičnim po svim elementima (na primer, veličina, oblik i težina) osim možda po njihovoj novčanoj `vrednosti`. **Odnos** je predstavljen linijom koja spaja dva entiteta, sa rombom na sredini koji definiše tip odnosa. **Atribut** je obeležje entiteta koji opisuje podatke ili svojstva pridružena entitetu. Na primer, kod problema obrtne rampe, više nas zanima `Novčić` koji se ubacuje u `OtvorZaNovac` obrtne rampe (odnos) i da li njegova novčana vrednost odgovara `ceni ulaznice` u zoološki vrt (poređenje vrednosti atributa). Varijantna ER notacija sadrži dodatne konstrukte, kao što su atributi po odnosima, odnosi „jedan-prema-više", „više-prema-više", specijalne odnose kao što je odnos nasleđivanja, atributi klase kao proširenja atributa individualnih entiteta. Na primer, model obrtne rampe prikazuje kardinalnost odnosa (takozvanu „arnost"), a ta karakteristika govori da će obrtna rampa propuštati više `Posetilaca`. Sofisticiranije notacije poseduju koncept **promenljivog entiteta**, čija pripadnost ili odnosi sa primercima drugih entiteta mogu da se menjaju tokom vremena. Na primer, u ER dijagramu koji predstavlja porodicu, članovi porodice i njihovi međusobni odnosi se menjaju prilikom stupanja u brak, dobijanja dece i smrti. Prema konvenciji, entiteti i odnosi se postavljaju tako da se odnosi čitaju sa leva na desno ili odozgo nadole.



SLIKA 4.4 Dijagram odnosa između entiteta za problem obrtne rampe

ER dijagrami su popularni jer formulišu sveobuhvatan pregled problema kojim se bavimo (tj. ti oni opisuju sve učesnike), a i zbog toga što je taj pogled relativno stabilan kada dolazi do promena zahteva. Izmena zahteva je mnogo izvesnije posledica izmene ponašanja jednog ili više entiteta nego promena skupa entiteta. Iz ta dva razloga, ER dijagram se koristi za modelovanje problema u ranoj fazi procesa definisanja zahteva.

Jednostavnost ER-a može da zavara, jer ju je, u stvari, prilično teško ispravno koristiti u praktičnoj primeni. Nivo detalja modelovanja nekog problema nije uvek očigledan čak i kad postoje samo tri glavne notacione konstrukcije. Na primer, da li rampa i otvor za novčić treba da se modeluju kao posebni entiteti, ili da budu predstavljeni još apstraktnijim entitetom obrtne rampe? Takođe će biti teško doneti odluku koji podaci predstavljaju entitete i koji podaci predstavljaju atribute. Na primer, da li brava treba da bude entitet? Postoje argumenti za svaki od tih izbora i protiv svakog od njih. Primarni kriterijum za donošenje odluka jeste da li opredeljenje za neku mogućnost pojašnjava opis, kao i da li dati izbor bez potrebe ograničava projektne odluke.

Primer: UML dijagrami klasa

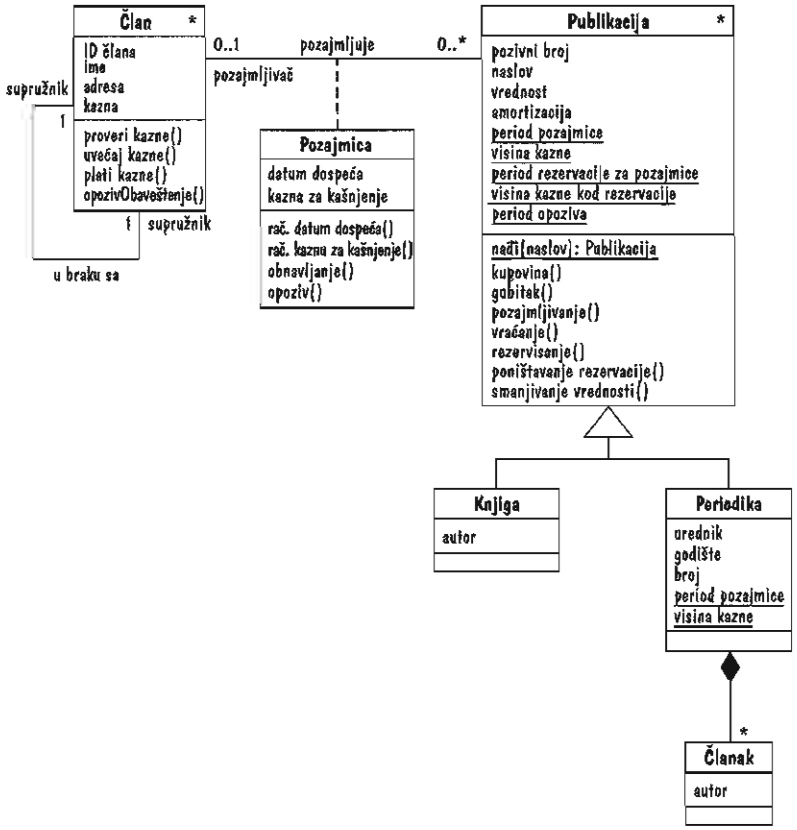
ER notacija se često koristi u dosta složenijim pristupima. Na primer, tzv. „objedinjeni jezik modelovanja" **UML** (**Unified Modeling Language**) (OMG 2003) predstavlja skup notacija koje se koriste za dokumentovanje specifikacija i dizajna softvera. U poglavlju 6 koristićemo UML za opisivanje objektno orijentisanih specifikacija i dizajna. UML je prvobitno bio namenjen objektno orijentisanim sistemima, pa zato on sistem predstavlja oslanjajući se na objekte i metode. Objekti predstavljaju ekvivalent entitetima, i organizovani su u klase koje poseduju hijerarhiju nasleđivanja. Svaki objekat obezbeđuje metode koji izvode akcije nad promenljivim objektima. Kada se objekti izvršavaju, oni šalju poruke u cilju pozivanja metoda drugih objekata, potvrđivanja akcija i prenosa podatka.

Predvodnik u grupi modela u UML specifikaciji je **dijagram klasa**, sofisticirani ER dijagram koji povezuje klase (entitete) u sklopu specifikacije. Premda se u literaturi o UML-u dijagrami klasa tretiraju pre svega kao dizajnerska notacija, moguće ih je koristiti kao notaciju za konceptualno modelovanje, gde klase predstavljaju realne entitete problema koji se modeluje. Moguće je da klasa u konceptualnom modelu, kao na primer klasa `Kupac`, odgovara implementacionoj klasi, kao što je `KupacZapis`, ali to ne mora uvek da bude slučaj. Zadatak projektanata softvera je da, na osnovu specifikacije dijagrama klasa, konstruišu pogodni projektni model strukture implementacionih klasa.

U opštem slučaju, vrste entiteta iz realnog sveta koje želimo da predstavimo dijagramom klasa uključuju: aktere (članove, operatore, osoblje); složene podatke koji treba da se skladište, analiziraju, transformišu ili prikazuju; ili zapise tranzijentnih događaja (na primer, poslovne transakcije, telefonske razgovore). Entiteti u sklopu problema biblioteke obuhvataju ljude, kao što su članovi i bibliotekari, zatim stavke u inventaru biblioteke, kao što su knjige i periodika, i na kraju transakcije izdavanja knjige.

Slika 4.5 prikazuje jednostavan UML dijagram klasa biblioteke. Svaki pravougaonik je **klasa** koja predstavlja kolekciju sličnih tipova entiteta. Na primer, jedna klasa predstavlja sve knjige iz biblioteke. Klasa poseduje: **ime**; skup **atributa** koji predstavljaju promenljive koje poprimaju različite vrednosti tokom vremena i kod različitih entiteta; i skupa **operacija** nad atributima klase. Pod „jednostavnom promenljivom" podrazumevamo promenljivu čije su vrednosti suviše jednostavne da bi činila klasu. Zbog toga adresu `Člana` modelujemo kao atribut, verovatno u formi jednog niza ili više nizova

znakova, dok podatke o `Clanu` vezane za kreditnu karticu, kreditnu instituciju, broj kreditne kartice, datum isteka ili adresu za slanje računa, modelujemo kao posebnu klasu (nije prikazana na slici). Uočite da mnogi atributi koje očekujemo u dijagramu klase biblioteke nedostaju (tj. zvučni zapisi i filmovi), ili su neprecizno definisani (tj. periodika, gde se ne pravi razlika između dnevnih novina i časopisa), a takođe su izostavljene i neke operacije (tj. postupak popravke knjiga ili nadoknade gubitka). Ovakva nepreciznost je tipična za početne konceptualne dijagrame. Ideja je da definišemo dovoljno atributa i operacija, na dovoljnom nivou detalja, tako da svako ko čita specifikaciju može da shvati šta klasa predstavlja i koje su njene odgovornosti.



SLIKA 4.5 UML model klasa za problem biblioteke

UML takođe omogućava onome ko pravi specifikaciju da definiše atribute i operacije, tako da su oni pridruženi samo klasi, a ne instancama klase. **Atribut u opsegu klase**, na slici predstavljen podvučeno, je vrednost kojoj mogu da pristupe sve instance klase. U dijagramu bibliotečkih klasa, atributi kao što su period rezervacije knjigae i visina kazne kod rezervacije, predstavljaju vrednosti zajedničke za sve publikacije u sistemu rezervacija. Stoga u ovom modelu, bibliotekar može da postavi i modifikuje rok izdavanja za klasu stavki (tj. knjige, periodiku, rezervisane stavke), ali ne i za individualne stavke. Slično tome, **operacija u opsegu klase**, prikazana na slici podvučeno, jeste operacija koju izvodi apstraktna klasa, a ne instanca klase, a operacija se izvršava nad novom

instancom ili skupom instanci. Uobičajene operacije u opsegu klase su `kreiranje()`, `traženje()` i `brisanje()`. Veza između dve klase, odnosno njihova **asocijacija**, ukazuje na odnos između klasnih entiteta. Asocijacija može da predstavlja interakcije ili događaje koji uključuju objekte u povezanim klasama, kao kada `Član` pozajmljuje `Publikaciju`. Alternativno, asocijacija može da predstavlja odnos dve klase u kojem je jedna klasa vlasništvo ili element druge klase, kao što je odnos između `Člana` i njegove `Kreditne kartice`. Ovaj drugi tip asocijacije nekada predstavlja tzv. **agregaciju**, ili odnose tipa „A poseduje B", kao u našem primeru. Agregacija se grafički predstavlja kao asocijacija sa belim rombom na jednom kraju. Klasa koja se nalazi na kraju gde je romb, jeste agregatna i uključuje ili poseduje instance klase (klasa) na drugom kraju (krajevima) asocijacije. **Kompozicija** je poseban tip agregacije, u kojoj su instance složene klase fizički sastavljene od instanci klasa koje predstavljaju sastavne delove (na primer, bicikl se sastoji od točkova, zupčanika, pedala, upravljača). Ova asocijacija se predstavlja kao agregacija sa crnim rombom. Kod modela biblioteke, svaka `Periodika`, kao što su dnevne novine i časopisi, sastavljena je od `Članaka`.

Asocijacija sa trouglom na jednom kraju predstavlja **generalizaciju**, a takođe se naziva *podtipska* relacija ili relacija tipa *„A je B"*. Klasa koja se nalazi na kraju asocijacije gde je trougao jeste nadređena (matična) klasa u odnosu na klase koje se nalaze na drugom kraju asocijacije, i koje se nazivaju **potklase**. Takva potklasa nasleđuje sve atribute, operacije i asocijacije nadređene klase. Prema tome, ne moramo da izričito odredimo da `Članovi` mogu da pozajmljuju `Knjige`, jer ta asocijacija se nasleđuje iz asocijacije između `Člana` i `Publikacije`. Nasleđeno ponašanje potkalse proširuje se dodatnim atributima, operacijama i asocijacijama. U stvari, dobra indicija o tome da li hoćemo da modelujemo entitet kao novu potklasu, umesto kao instancu u postojećoj klasi, jeste da li su nam zaista potrebni novi atributi, operacije ili asocijacije za modelovanje varijantne klase. U mnogim slučajevima, možemo da modelujemo varijante kao instance klase koje imaju različite vrednosti atributa. Kod problema biblioteke, da li je neka stavka rezervisana ili izdata predstavljamo atributima[2] `Publikacije`, umesto da pravimo potklase `Rezervisano` ili `Pozajmljeno`. Asocijacije imaju svoje oznake; to su obično glagoli, koji opisuju odnose između asociranih entiteta. Krajnja tačka asocijacije može takođe biti označena, čime se opisuje uloga koju entitet igra u asocijaciji. Takva **imena uloga** su korisna za određivanje konteksta entiteta u odnosu na određenu asocijaciju. U primeru biblioteke, možemo da pratimo koji su članovi u braku, tako da možemo da upozorimo jednog od supružnika ako onaj drugi kasni sa vraćanjem knjige. Krajnje tačke asocijacije mogu takođe biti označene **multiplikatorima** koji određuju ograničenja po pitanju broja entiteta ili po broju veza između asociranih entiteta. Multiplikatori mogu biti izraženi zadatim brojem, opsegom brojeva, ili neograničenim brojem (označeno sa „*"). Multiplikator na jednom kraju asocijacije ukazuje koliko instanci te klase može biti povezano sa jednom instancom asocirane klase.

Na osnovu toga u bilo kom trenutku, `Član` može da ne pozajmi nijednu `Publikaciju`, ili da pozajmi jednu ili više njih, ali jednu zasebnu `Publikaciju` može da pozajmi najviše jedan `Član`.

2 U kasnijim primerima modelovaćemo stanje pozajmice i stanje rezervacije stavki kao stanja konačnog automata (slika 4.9), a ta informacija je uključena u detaljni model klasa biblioteke (slika 4.18).

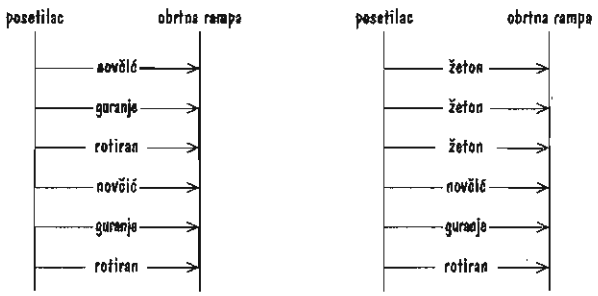
Klasa `Pozajmica` u modelu biblioteke je **asocijativna klasa**, koja dovodi u vezu atribute i operacije sa asocijacijom. Asocijativne klase se koriste za okupljanje informacija koje ne mogu da budu dodeljene samo jednoj ili drugoj klasi. Na primer, atributi `Pozajmice` nisu osobine onog ko uzima pozajmicu ili stavke koja se izdaje, već su pre osobine transakcije izdavanja ili ugovora. Asocijativna klasa ima u asocijaciji tačno jednu instancu po vezi, tako da je modelovanje `Pozajmice` u vidu asocijativne klase ispravno samo ako modelujemo trenutno stanje inventara biblioteke (tj. da modelujemo samo aktuelne pozajmice). Ako umesto toga želimo da održavamo istorijat svih transakcija izdavanja knjiga, tada (zatoa što član može jednu stavku da pozajmi više puta), modelujemo `Pozajmicu` kao potpuno označenu klasu.

Tragovi događaja

Iako su ER dijagrami korisni kod pružanja sveobuhvatnog pogleda na problem koji se modeluje, pogled je pretežno strukturni, prikazujući odnose između entiteta, ne govoreći pri tome ništa o njihovom ponašanju. Potrebne su nam druge notacione platforme za opis zahteva sa aspekta ponašanja sistema.

Trag događaja je grafički opis niza događaja koji se razmenjuju između entiteta u stvarnosti. Svaka uspravna linija predstavlja vremenska osa određenog entiteta, čije ime se pojavljuje na vrhu linije. Svaka vodoravna linija predstavlja događaj ili interakciju između dva entiteta, što se obično shvata kao poruka koju jedan entitet prosleđuje drugom. Vreme teče odozgo nadole, pa ako se jedan događaj pojavljuje iznad drugog događaja, onda to znači da se on desio pre donjeg događaja. Svaki graf prikazuje samo jedan **trag**, predstavljajući samo jedno od više mogućih ponašanja. Slika 4.6 prikazuje dva traga za slučaj obrtne rampe. Trag na levoj strani predstavlja tipično ponašanje, dok trag na desnoj strani prikazuje izuzetno ponašanje, odnosno šta se dešava kada Posetilac pokuša da se ušunja u zoološki vrt, ubacujući bezvredni novčić (`žeton`) u otvor za novčić.

Tragovi događaja su popularni kod projektanata i naručilaca, jer poseduju relativno preciznu semantiku, sa izuzetkom vremenskih odnosa, ali su jednostavni i laki za razumevanje. Dobar deo jednostavnosti potiče od dekompozicije opisa zahteva na scenarije. Pri tome se svaki scenario posmatra (modeluje, čita, pokušava da razume) kao poseban trag. Zbog istih osobina, tragovi događaja nisu efikasni za dokumentovanje ponašanja.



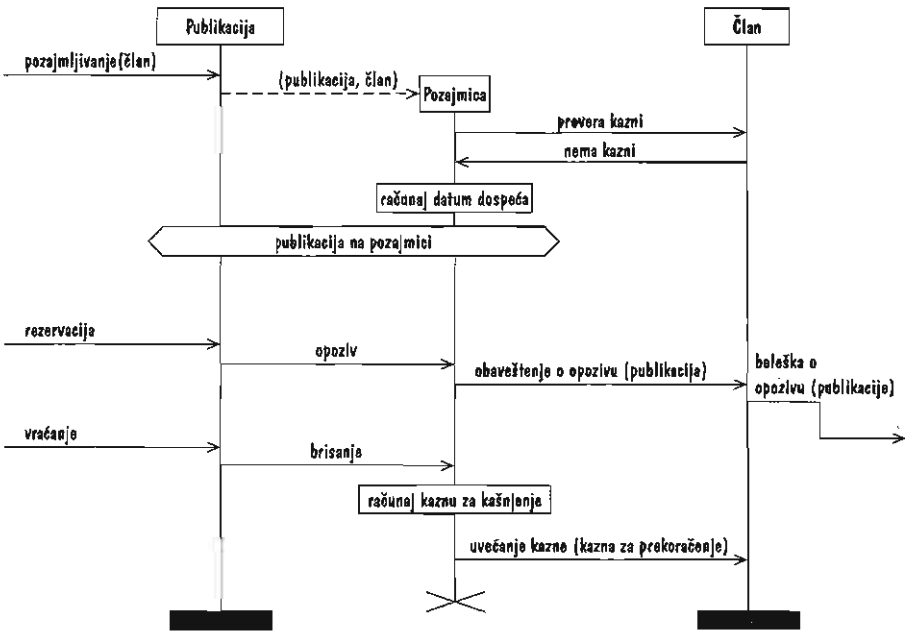
SLIKA 4.6 Tragovi događaja za slučaj obrtne rampe

Tragove ne koristimo za definisanje kompletnog opisa zahtevanog ponašanja, iz razloga što je broj scenarija koje je neophodno predstaviti velik. Tragovi su najkorisniji u početnoj

fazi projekta kada je neophodno postići konsenzus oko ključnih zahteva i pomoći učesnicima u razvoju pri identifikaciji važnih entiteta problema koji se modeluje.

Primer: dijagram redosleda poruka

Dijagram redosleda poruka - MSC (*Message Sequence Chart*) (ITU 1996) predstavlja poboljšanu notaciju traga događaja, sa mehanizmima za kreiranje i uništavanje entiteta, određivanje akcija i tajmera i sklapanje tragova. Slika 4.7 prikazuje dijagram sekvence poruka (MSC) transakcije izdavanja knjiga za slučaj biblioteke. Svaka uspravna linija predstavlja **entitet** učesnika, dok je **poruka** prikazana kao strelica od entiteta pošiljaoca do entiteta primaoca, a oznaka na strelici definiše naziv poruke i parametre, ako ih ima. Strelica poruke može imati krivinu nadole (na primer, `obaveštenje o opozivu`) da bi se predstavio protok vremena od trenutka slanja poruke do njenog prijema. Entiteti mogu nastajati i nestajati na tragu događaja. Isprekidana strelica, nekada sa naznačenim parametrima, predstavlja događaj *stvaranja* novog entiteta, dok krstići na kraju linije života entiteta predstavljaju kraj njegovog izvršavanja. Nasuprot tome, popunjeni pravougaonik na kraju linije predstavlja kraj specifikacije entiteta, bez značenja kraja izvršavanja. **Akcije**, kao što su pozivanje operacije ili menjanje vrednosti promenljivih, definisane su obeleženim pravougaonicima, postavljenim na liniji izvršenja entiteta, u tački na tragu gde se akcija dešava. Prema tome, u našem MSC modelu za izavanje knjiga iz biblioteke, zahtevi za pozajmicu se šalju `Publikaciji` koja se izdaje, a entitet `Publikacija` je odgovoran za stvaranje entiteta `Pozajmica` koji upravlja podacima specifičnim za izdavanje, kao što je `rok na koji se knjiga izdaje`. Rezervisanje stavke koja je izdata rezultuje u `opozivu` te stavke. Vraćanje pozajmljene stavke okončava pozajmicu, ali ne pre nego što se izračuna `kazna prekoračenja`, ako je ima, u slučaju vraćanja stavke posle `datuma dospeća`.



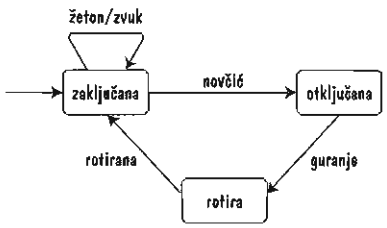
SLIKA 4.7 Dijagram redosleda poruka (MSC) za transakcije izdavanja knjiga iz biblioteke

Postoje mehanizmi za sastavljanje i prečišćavanje dijagrama redosleda poruka (MSC). Na primer, važna stanja u evoluciji entiteta mogu biti navedena kao uslovi, koji se predstavljaju u vidu obeleženog šestougaonika. Pored toga možemo definisati mali skup podtragova između uslova, i izvesti različite tragove za dostizanje tačaka gde su stanja entiteta identična. Na primer, postoji više scenarija za prelaz iz stanja `publikacija na pozajmici` i kraja tranzicije pozajmice. Ti scenariji su: član obnavlja pozajmicu jedanput, član obnavlja pozajmicu dva puta, član vraća publikaciju, član izveštava da je publikacija izgubljena. Svaki od tih scenarija može da se doda na prethodni trag, a to je da član uspešno pozajmljuje publikaciju. Takve odlike sastavljanja i pročišćavanja pomažu da smanjimo broj MSC-ova koji su potrebni za potpunu specifikaciju problema. Međutim ove osobine ne rešavaju u potpunosti problem eksplozije tragova, tako da se dijagrami redosleda poruka (MSC) obično koriste za modelovanje ključnih scenarija umesto za specifikaciju celokupnog problema.

Konačni automati

Notacija zasnovana na konačnim automatima koristi se za predstavljanje skupa tragova događaja u sklopu jedinstvenog modela. **Konačni automat** je grafički opis svih komunikacija između sistema i njegovog okruženja. Svaki čvor, koji se naziva **stanje**, predstavlja postojan skup uslova koji se održavaju u intervalu između pojave događaja. Svaka grana, koju nazivamo **prelaz stanja**, predstavlja promenu koje je događaj proizveo u ponašanju ili u uslovima. Svaki prelaz iz jednog stanja u drugo obeležen je pobudnim događajem, a ponekad i izlaznim događajem, kome prethodi simbol „/", koji se generiše kada se dogodi prelaz.

Konačni automati su korisni za definisanje dinamičkog ponašanja i za opis načina promene ponašanja kao odgovor na prethodne događaje. Konačni automati su posebno pogodni za modelovanje načina na koji se odgovori sistema na isti ulaz *menjaju* tokom izvršavanja. Prelazi u drugo stanje koji proističu iz nekog datog stanja definisani su skupom događaja koji mogu pobuditi te prelaze i odgovarajućim odgovorom na taj skup događaja. Prema tome, kada je obrtna rampa (prikazana na slici 4.8) u stanju `otključano`, njeno ponašanje je različito od ponašanja kada je u stanju `zaključano`.



SLIKA 4.8 Model konačnog automata za slučaj obrtne rampe

U datom slučaju, ponašanje odgovara različitim ulaznim događajima. Ako se desi neki neočekivani događaj (na primer, ako korisnik pokušava da `gura` obrtnu rampu kada je automat u stanju `zaključano`), taj događaj biće ignorisan i odbačen. Ovo drugo ponašanje mogli bismo da tačno opišemo kao prelaz iz stanja `zaključano` u stanje `zaključano`, a pobudu predstavlja događaj `guranje`. Međutim, uključivanje takvih prelaza koji nemaju nikakav efekat, može da napravi pometnju u modelu. Prema

tome, najbolje je ograničiti upotrebu autoprelaza i koristiti samo one prelaze koji imaju vidljivi efekat, kao na primer, neki izlazni događaj.

Putanja kroz automat, od početnog stanja automata pa preko prelaza iz stanja u stanje, predstavlja trag uočljivih događaja u datom okruženju. Ako je automat **deterministički**, što znači da za svako stanje i događaj postoji jednoznačan odgovor, onda putanja kroz automat predstavlja *baš onaj* trag događaja koji će se desiti, i definiše redosled ulaznih događaja koji predstavljaju pobude za prelaze iz stanja u stanje na putanji. Primeri tragova u našoj specifikaciji obrtne rampe su:

```
novčić, guranje, rotirano, novčić, guranje, rotirano, ...
žeton, žeton, žeton, novčić, guranje, rotirano, ...
```

što odgovara tragu događaja sa slike 4.6.

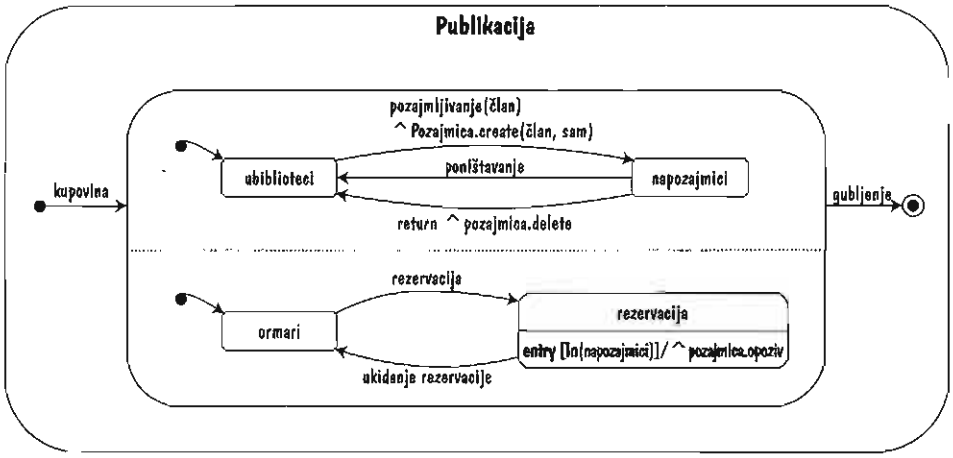
Možda ste se sreli sa konačnim automatima u nekim drugim predmetima iz računarskih nauka. U predavanjima o teoriji računara, konačni automati se koriste kao automati koji prepoznaju nizove znakova u regularnim jezicima. Na neki način, specifikacije pomoću automata služe sličnoj nameni kao i kod pravih automata. Pomoću automata se specifikuje redosled ulaznih i izlaznih događaja koje predloženi sistem treba da realizuje. Prema tome, specifikaciju automata vidimo kao kompaktno predstavljanje skupa željenih, uočljivih, tragova događaja, upravo kao što je konačni automat kompaktna predstava skupa znakovnih nizova koje prepoznaje automat.

Primer: UML-ovdijagram prelaza iz stanja u stanje

UML-ov dijagram prelaza iz stanja u stanje prikazuje dinamičko ponašanje objekata neke UML klase. UML-ov dijagram klasa pruža statički pogled visokog nivoa na dati problem, zavisno od obuhvaćenih entiteta i njihovih odnosa. Međutim, ovaj model ništa ne govori o ponašanju entiteta ili promeni njihovog ponašanja u odgovor na ulazne događaje. Dijagram prelaza između stanja prikazuje kako instance klase treba da menjaju stanja i kako njihovi atributi treba da menjaju vrednosti prilikom međusobne interakcije objekata. Oni predstavljaju dopunu dijagrama redosleda poruka (MSC), s time da MSC prikazuje događaje koji se odvijaju između entiteta, bez razmatranja ponašanja svakog od entiteta, dok dijagram prelaza između stanja prikazuje kako zaseban entitet reaguje na ulazne događaje i pri tome generiše izlazne događaje.

UML model predstavlja skup konkurentnih dijagrama, svaki za po jednu instancu, tj. objekat, koji međusobno komuniciraju preko poruka (OMG 2003). Svaka klasa u UML dijagramu klasa ima pridruženi dijagram prelaza, koji određuje dinamičko ponašanje objekata te klase. Slika 4.9 prikazuje UML dijagram prelaza stanja za klasu `Publikacija` iz modela klasa za biblioteku.

UML-ov dijagram prelaza iz stanja u stanje ima bogatu sintaksu, pri čemu je veći deo preuzet iz Harelovog koncepta grafova stanja (Harel 1987), uključujući hijerarhiju stanja, istovremeno izvršavanje i komunikaciju između automata. Hijerarhija stanja se koristi da se smanji gustina grafa tako što se sva stanja sa zajedničkim prelazima sakupljaju u **nadstanja**. Nadstanje možemo da posmatramo kao podautomat sa svojim skupom stanja i prelaza.

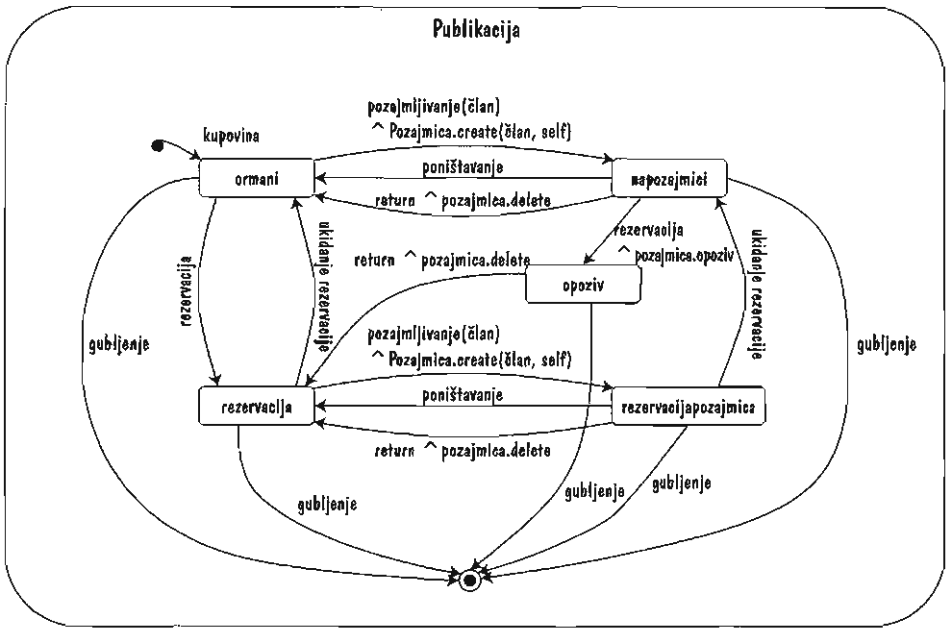


SLIKA 4.9 UML-ov dijagram prelaza stanja za klasu `Publikacija`

Prelaz čije je odredišno stanje nadstanje, ponaša se kao prelaz na podrazumevano prvobitno stanje nadstanja, označen je strelicom koja ide od unutrašnjeg crnog kruga nadstanja. Prelaz čije je izvorno stanje nadstanje, igra ulogu skupa prelaza, po jedan za svako od unutrašnjih stanja nadstanja. Na primer, u grafu stanja `Publikacije`, prelaz čiju pobudu vrši događaj `gubljenje`, može biti omogućen od strane bilo kog unutrašnjeg stanja nadstanja. Taj prelaz se završava u krajnjem stanju i označava kraj životnog veka objekta.

Nadstanje može stvarno da sadrži više konkurentnih podautomata, odvojenih isprekidanom linijom. UML-ov dijagram prelaza za stanja `Publikacije` obuhvata dva podautomata: jedan koji ukazuje na to da li publikacija jeste ili nije izdata, a drugi da li publikacija jeste ili nije rezervisana. Za podautomate kažemo da rade **konkurentno**, što znači da instanca `Publikacije` može u bilo kom trenutku da primi i odgovori na događaje od interesa na jednom, ili na oba podautomata. U opštem slučaju, konkurentni konačni automati se koriste za modelovanje odvojenih, nepovezanih potponašanja, čime se olakšava razumevanje i razmatranje svakog potponašanja. Poređenje dijagrama za prelaz stanja `Publikacije` sa slike 4.10 sa njegovim ekvivalentom u kome nisu iskorišćene hijerarhija stanja ili konkurentnost, pokazuje da ovaj drugi deluje nesređeno i da ima mnogo ponavljanja. Napomenimo da nesređeni dijagram prelaza stanja poseduje stanje za svaku kombinaciju stanja sa slike 4.19 (`orman = Publikacija` je u biblioteci i nije rezervisana, `napozajmici = Publikacija` je na pozajmici i nije rezervisana itd)[3].

3 Nesređeni dijagram prelaza stanja takođe poseduje stanje `opoziv` koje pokriva slučaj gde je rezervisana publikacija već pozajmljena i potrebno je da se izvrši opoziv. To ponašanje ne može da se modeluje kao prelaz iz stanja `napozajmici` u stanje `rezervacijapozajmica`, jer stanje `rezervacijapozajmica` ima prelaz `poništavanje` (koje se koristi da spreči zahtev pozajmice ako `Član` ima neizmirene kazne), što bi bilo neprimereno u toj situaciji. Specijalni slučaj je modelovan na slici 4.9, gde se testira **ulaz** (ključna reč **entry** objašnjena je u nastavku) prema stavci rezervacija, tj. da li je konkurentni podautomat (ključna reč **In**) u stanju `napozajmici`, nakon čega se emituje događaj `opoziv` ako je iskaz tačan.



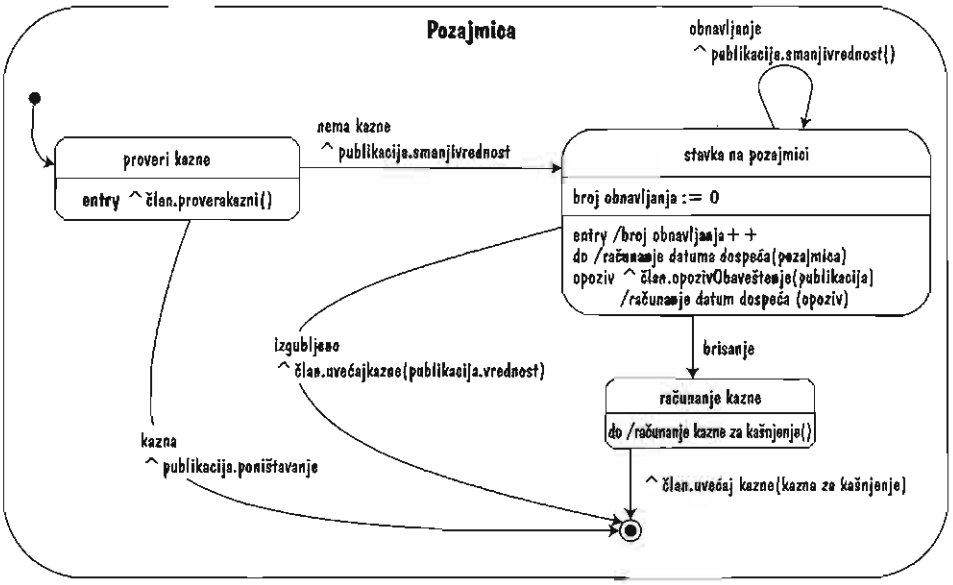
SLIKA 4.10 Nesređeni UML-ov dijagram prelaza stanja za klasu `Publikacija`

Prelazi stanja označeni su odgovarajućim događajima i uslovima koji aktiviraju te prelaze, kao i njihovim sporednim efektima. Oznake prelaza stanja imaju sledeću sintaksu:

```
događaj(argumenti) [uslov] /akcija* ^Objekat.događaj(argumenti)*
```

gde pobudni **događaj** predstavlja poruku koja može da sadrži parametre. **Uslov** za omogućavanje, definisan u uglastim zagradama, predstavlja predikatski iskaz nad vrednostima atributa objekta. Ako se prelaz izvrši, njegove **akcije**, ispred kojih prethodi kosa crta (/), definišu vrednosti dodeljene atributima objekta. Zvezdica „*" ukazuje na to da prelaz može imati proizvoljno mnogo akcija. Ako se prelaz izvrši, može da generiše proizvoljno mnogo **izlaznih događaja**, `/^Objekat.događaj`, gde ispred svakog stoji znak „^". Izlazni događaj može da sadrži parametre i namenjen je određenom `Objektu` ili svim objektima. Na primer, u nesređenom grafu stanja `Publikacija` (slika 4.10), prelaz u stanje opoziva je omogućen ako je publikacija u stanju `napozajmici` u trenutku prijema zahteva za `rezervisanje` stavke. Kada se prelaz izvrši, on šalje događaj ka objektu `Pozajmica`, koji dalje obaveštava onoga ko je pozajmio da ta stavka mora biti vraćena u biblioteku pre datuma dospeća. Svi elementi oznake prelaza su opcioni. Na primer, prelaz ne treba da je omogućen od strane ulaznog događaja. Prelaz mogu da omoguće samo uslovi, a ako nema njih, onda je prelaz uvek omogućen.

UML-ov dijagram prelaza stanja za asocijativnu klasu Pozajmica na slici 4.11 ilustruje kako stanja mogu da se objasne pomoću: lokalnih promenljivih (na primer, promenljiva `broj obnavljanja`), akcija i aktivnosti. Promenljive koje su lokalne u odnosu na stanje, deklarišu se i inicijalizuju u centralnom odeljku stanja.



SLIKA 4.11 UML-ov dijagram prelaza stanja za klasu `Pozajmica`

Donji odeljak stanja specifikuje akcije i aktivnosti nad lokalnim promenljivim stanja, kao i nad atributima objekta. Razlika između akcija i aktivnosti je jedva primetna, jer **akcija** je kratka obrada koja se ne može prekinuti, kao što je recimo dodeljivanje izraza promenljivoj ili slanje poruke. Akciju može da pokrene ulazak u stanje ili izlazak iz njega, pri čemu su ti slučajevi označeni ključnim rečima **`entry`** (ulaz) ili **`exit`** (izlaz), nakon kojih sledi proizvoljan broj akcija i generisanih događaja. Pobuda akcije može da se vrši i pojavljivanjem događaja, što se u tom slučaju označava imenom događaja nakon koga sledi proizvoljan broj akcija i generisanih događaja. U dijagramu prelaza stanja `Pozajmice`, promenljiva `broj obnavljanja` se inkrementira svaki put kada se pređe u stanje `stavka izdata`, tj. svaki put kada se obnovi pozajmica. Takođe, događaj `opozivObaveštenje` se šalje `Članu` svaki put kada se u tom stanju primi događaj `opoziv`. Nasuprot akciji, **aktivnost** predstavlja kompleksniju obradu koja se izvršava u nekom vremenskom intervalu i može biti prekinuta, kao što je recimo izvršavanje operacije. Aktivnosti se iniciraju na ulasku u stanje. Kada se prelaz izvršava (uključujući i autoprelaz, sličan onome čije okidanje izvršava `obnavljanje`), redosled primenjivanja akcija je sledeći: prvo se primenjuju izlazne akcije stanja na izvoru prelaza, zatim slede akcije samog prelaza, praćene ulaznim akcijama i aktivnostima novog stanja.

Semantika UML dijagrama i način njihovog uklapanja namerno nisu definisani, čime je onima koji vrše specifikaciju prepušteno da izaberu semantiku koja najbolje odgovara datom problemu. Međutim, mnogi praktičari posmatraju UML dijagrame stanja kao spoj konačnih automata i FIFO (*First-in, First-out*) komunikacionih kanala. Svaki konačni automat objekta poseduje ulazni red čekanja, koji prihvata poruke poslate od strane drugih objekata modela ili iz okruženja. Poruke se skladište u ulaznom redu čekanja, u redosledu pristizanja. U svakom koraku izvršavanja, konačni automat čita poruku sa vrha pripadajućeg ulaznog reda čekanja i pri tome uklanja poruku. Poruka ili izaziva prelaz konačnog automata u drugo stanje, ili se, u suprotnom, odbacuje. Korak se *izvršava do kraja*, što znači da automat nastavlja izvršavanje svih omogućenih prelaza sta-

nja, uključujući i prelaze koji čekaju na okončanje nekih operacija. Prema tome, konačni automat reaguje samo na jednu poruku u jednom trenutku.

Najteži deo u postupku izrade modela konačnog automata je dekompozicija ponašanja objekta na stanja. Neka razmišljanja u vezi sa stanjem uključuju:

- Klase koje odgovaraju mogućem budućem ponašanju, kao što je definisano redosledom ulaznih događaja prihvaćenih od strane konačnog automata. Na primer, svaka iteracija redosleda događaja `novčić, guranje, rotacija` ostavlja obrtnu rampu u položaju `zaključano`, gde čeka na novog posetioca
- Intervali vremena između uzastopnih događaja, kao na primer, vreme od početka do završetka neke operacije
- Imenovane kontrolne tačke u evoluciji objekta, tokom kojih objekat vrši neku obradu (na primer, stanje `računanje kazne`) ili čeka na neki ulazni događaj (na primer, stanje `stavka izdata`)
- Segmenti ponašanja objekta. Na primer: knjiga je izdata ili je u ormanima biblioteke; stavka je na režimu rezervisanja, što znači da može da se pozajmi samo na kratak period, ili nije na režimu rezervisanja.

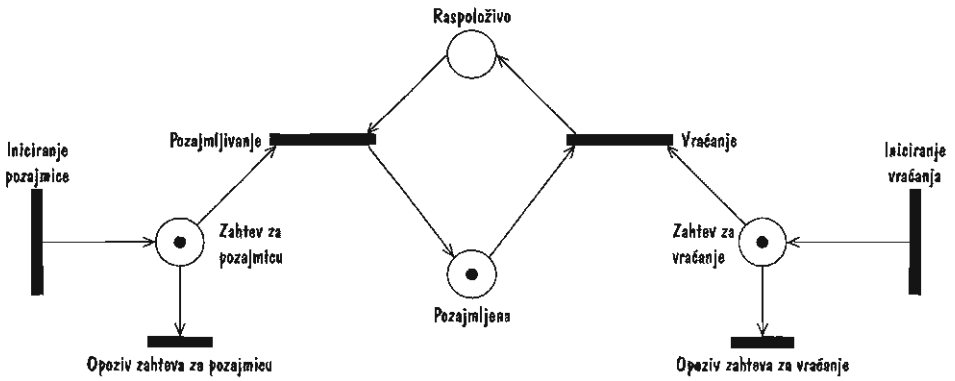
Neke osobine objekta mogu da se modeluju ili u vidu atributa (definisanih u dijagramu klasa) ili u vidu stanja (definisanih u dijagramu prelaza stanja objekta), pri čemu nije uvek očigledno koje predstavljanje je najbolje. Naravno, ako je skup mogućih vrednosti osobina velik (na primer, kazne `Člana` biblioteke), onda je najbolje da se ta osobina modeluje kao atribut. Alternativno, ako događaj, na koji je objekat spreman da reaguje, zavisi od osobine (na primer, da li je knjiga izdata), onda je najbolje da se ta osobina modeluje kao stanje. U protivnom, treba odabrati predstavljanje koje kao rezultat daje najjednostavniji, lako razumljiv, model.

Primer: Petrijeve mreže

UML-ovi dijagrami prelaza stanja lepo modularizuju dinamičko ponašanje problema u ponašanja individualnih instanci klasa, što omogućuje lakše sagledavanje odvojenog ponašanja svake klase u odnosu na specifikaciju celog problema u sklopu jednog dijagrama. Modularizacija, s druge strane, otežava uvid u međusobne interakcije objekata. Posmatranjem pojedinačnog dijagrama prelaza stanja možemo da uočimo trenutak kada neki objekat šalje poruku drugom objektu. Da bismo utvrdili da li poruka koju šalje jedan objekat može biti primljena od strane drugog objekta, moramo da istovremeno analiziramo dijagrame prelaza stanja oba objekta. U stvari, ako želimo da budemo potpuno sigurni, neophodno je utvrditi sva moguća izvršavanja (tragove događaja) oba konačna automata i dokazati da je uvek kada jedan objekat šalje poruku drugom objektu, odredišni objekat spreman da tu poruku primi i na nju reaguje.

Petrijeve mreže (Peterson 1977) predstavljaju oblik notacije prelaza stanja koji se koristi za modelovanje konkurentnih aktivnosti i njihovih interakcija. Krugovi u mreži predstavljaju **mesta** obavljanja aktivnosti ili provere uslova, dok ispunjeni pravougaonici predstavljaju **prelaze**. Usmerene strelice, koje se nazivaju **lukovi**, povezuju prelaz sa *ulaznim mestima* i *izlaznim mestima*. Mesta su popunjena **tokenima**, koji predstavljaju uslove za omogućavanje prelaza. Kada se prelaz *aktivira*, uklanjaju se tokeni iz svih ulaznih mesta i prebacuju u izlazna mesta. Svakom luku može biti dodeljena **vrednost** koja definiše broj tokena koji se uklanjaju iz ulaznog mesta, odnosno broj tokena koji se

ubacuje u svako izlazno mesto nakon aktiviranja prelaza. Prelaz je *omogućen* ako u sklopu njegovih ulaznih mesta postoji najmanje onoliko tokena kolika je vrednost luka, čime nastaju uslovi za stvarno aktiviranje prelaza. Otuda su, na slici 4.12, omogućeni prelazi `Vraćanje`, `Opoziv zahteva za vraćanje` i `Opoziv zahteva za pozajmicu`. Aktiviranje prelaza `Vraćanje` uklanja token iz oba mesta `Zahtev za vraćanje` i `Napozajmici`, i ubacuje token u `Raspoloživo`. **Markiranje** mreže, koje predstavlja raspodelu tokena po mestima, menja se aktiviranjem prelaza. U svakom koraku izvršavanja, markiranje definiše skup omogućenih prelaza. Jedan omogućeni prelaz je *nedeterministički* odabran za aktiviranje, pri čemu aktiviranje tog prelaza proizvodi novo markiranje, koje može da omogući različit skup prelaza. Konkurentno ponašanje možemo da modelujemo kombinujući u jednoj mreži aktivnosti, prelaze i tokene za više entiteta koji se izvršavaju. Konkurentni entiteti se sinhronizuju uvek kada njihove aktivnosti ili mesta predstavljaju ulazna mesta istog prelaza. Ta sinhronizacija osigurava da se sve aktivnosti, koje treba da se dogode pre prelaza, zaista i dogode pre aktiviranja prelaza, pri čemu se ne ograničava redosled u kojem te aktivnosti treba da se dogode.



SLIKA 4.12 Petrijeva mreža za izdavanje knjige iz biblioteke

Osobine konkurentnosti i sinhronizacije posebno su korisne kod modelovanja događaja čiji redosled pojavljivanja nije bitan. Na primer, razmotrimo sobu za hitne intervencije u bolnici. Pre početka tretmana pacijenta, više događaja mora da se desi. Osoblje za trijažu mora da pokuša da pronađe ime i adresu pacijenta i odredi njegovu krvnu grupu. Neko mora da utvrdi da li pacijent diše, a takođe da ispita da li pacijent ima povrede. Ti događaji se ne dešavaju u utvrđenom redosledu, ali se svi moraju desiti pre nego što tim lekara započne detaljno ispitivanje. Nakon što tretman započne (tj. kada se napravi prelaz sa preliminarnog ispitivanja na detaljno ispitivanje), lekari započinju nove aktivnosti. Ortopedi proveravaju da li ima preloma kostiju, hematolog vrši analize krvi, a hirurg stavlja kopče na rane koje krvare. Aktivnosti lekara su međusobno nezavisne, ali nijedna ne može da započne pre prelaza sa preliminarnog ispitivanja. Model sobe za hitne intervencije zasnovan na konačnim automatima može imati jednu od dve varijante: može definisati samo jedan redosled događaja, čime se isključuje više prihvatljivih ponašanja, ili može definistai sve moguće redoslede događaja, što rezultira složenim modelom relativno jednostavnog problema. Model iste sobe za hitne intervencije uz oslonac na Petrijeve mreže elegantno izbegava oba pomenuta problema.

Osnovne Petrijeve mreže su pogodne za modelovanje toka kontrola kroz događaje ili između konkurentnih entiteta. Međutim, ako želimo da modelujemo kontrolu koja zavisi od vrednosti podataka (na primer, izdavanje određene knjige iz zbirke knjiga), neophodno je koristiti notaciju Petrijevih mreža na visokom nivou. Predložen je veći broj proširenja osnovne Petrijeve mreže, u cilju poboljšanja ekspresivnosti notacije, uključujući lukove inhibitore, koji omogućavaju prelaze samo ako u ulaznom mestu nema tokena. U poboljšanja spadaju i prioriteti među prelazima, vremenska ograničenja, kao i strukturisani tokeni koji imaju vrednost.

Za modelovanje problema biblioteke, koji prati informacije i događaje za više članova i publikacija, potrebna nam je notacija Petrijeve mreže koja podržava strukturisane tokene i akcije prelaza (Ghezzi i dr. 1991). Akcija prelaza ograničava koji tokeni u ulaznim mestima mogu da omoguće prelaz i određuje vrednosti izlaznih tokena. Slika 4.13 predstavlja specifikaciju Petrijeve mreže na visokom nivou za problem biblioteke. Svako mesto skladišti tokene različitog tipa. Na primer, `Raspoloživo` smešta token za svaku `stavku` biblioteke koja nije trenutno izdata. Token u `Kaznama` predstavlja *n*-torku (tj. uređeni skup *n* elemenata, ili skraćeno *n*-torka) koja preslikava `člana` u vrednost njegovih neizmirenih bibliotečkih kazni. Token u `Na pozajmici` je drugi tip n-torke koji preslikava `člana` i `stavku` iz biblioteke na `datum dospeća`. Nekoliko predikata prelaza i akcija prikazano je na slici 4.13, kao što je akcija u okviru `Obrade uplate`, gde su ulazi `uplate` i aktuelne `kazne` datog člana, dok je izlaz nova n-torka `Kazne`. Predikati, koji nisu prikazani, tvrde da ako tokeni na ulazu i izlazu prelaza imaju isto ime, tada moraju da imaju i istu vrednost. Tako u prelazu `Uzimanje pozajmice`, gde ***člana*** formira `Zahtev za pozajmicu`, ***člana*** ne može da ima neizmirene `Kazne` i mora da odgovara ***članu*** koji se pojavljuje u generisanoj n-torci `Na pozajmici`. U isto vreme, ***stavka*** koja se pozajmljuje mora da odgovara ***stavci*** u opciji ***Raspoloživo***. U protivnom, te n-torke ne omogućavaju prelaz. Mreža započinje inicijalnim markiranjem n-torki u `Raspoloživo (član, 0)` i n-torki u `Kaznama`. Kako korisnici biblioteke aktiviraju ulaznih prelaza `Plaćanje`, `Iniciranje pozajmice` i `Iniciranje vraćanja`, novi tokeni se uvode u sistem, čime se omogućavaju prelazi u okviru biblioteke, koji dalje aktiviraju i ažuriraju tokene `Kazne`, kao i tokene `Raspoloživo` i `Na pozajmici` itd.

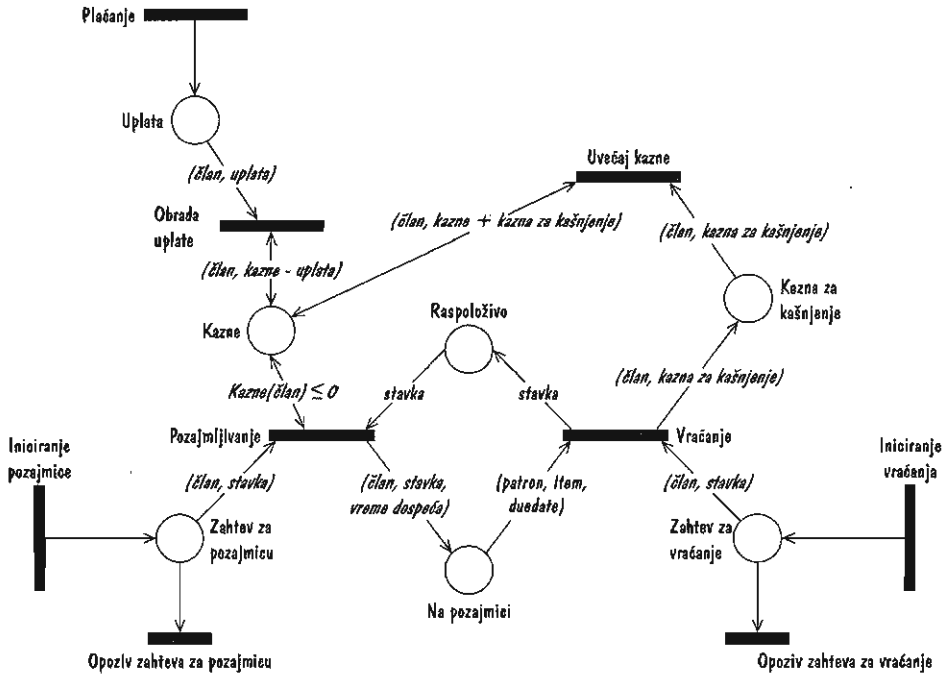
Dijagrami toka podataka

Paradigme notacija koje smo razmatrali do sada promovišu dekompoziciju problema na entitete (ER dijagram), scenarije (trag događaja), kao i na kontrolna stanja (tj. ekvivalentne klase scenarija) (konačne automate). Međutim, zahtevi formulisani u ranoj fazi se obično iskazuju kao:

- zadaci koji treba da se okončaju;
- funkcije koje treba da se izračunaju;
- podaci koji treba da se analiziraju, transformišu ili zabeleže.

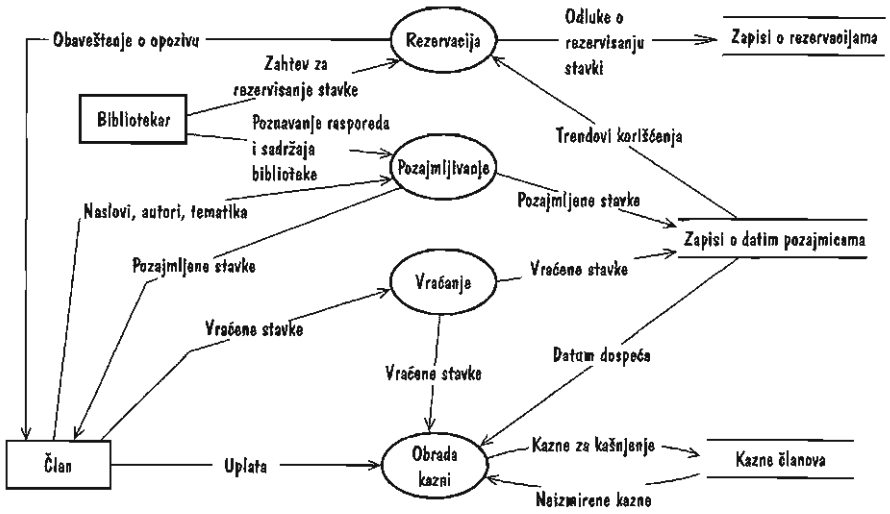
Takvi zahtevi, kada se dekomponuju na entitete, scenarije ili stanja, prelaze u skupove ponašanja na niskom nivou koje je neophodno koordinisati budući da su raspoređeni na više entiteta. Modularna struktura otežava uočavanje funkcija modela na visokom nivou apstrakcije. U primeru biblioteke, nijedan od prethodnih modela ne pokazuje efikasno

sve korake i njihove varijante koje član mora da preduzme da bi pozajmio knjigu. To je razlog popularnosti notacija koje promovišu funkcionalnu dekompoziciju.



SLIKA 4.13 Petrijeva mreža za problem biblioteke

Dijagram toka podataka (*Data-Flow Diagram*, DFD) modeluje funkcionalnost i tok podataka između funkcija. Elipsa predstavlja **proces** ili funkciju koja transformiše podatke. Strelica predstavlja **tok podataka**, pri čemu strelica koja ulazi u elipsu predstavlja ulaz njegove funkcije, a strelica koja iz nje izlazi predstavlja jedan od izlaza funkcije. Slika 4.14 prikazuje dijagram toka podataka na visokom nivou za problem biblioteke. Problem je razbijen na korake, gde rezultati prethodnih koraka ulaze u naredne. Podaci koji nastavljaju da postoje nakon njihovog korišćenja u jednoj obradi (na primer, informacija o neizmirenim kaznama člana), mogu da se sačuvaju u **skladištu podataka**, tj. formalnom skladištu ili bazi podataka, što je na dijagramu predstavljeno sa dve paralelne linije. Izvori ili uviri podataka, predstavljeni pravougaonicima, jesu **učesnici**, tj. entiteti koji obezbeđuju ulazne podatke ili primaju izlazne rezultate. Elipsa može da predstavlja apstrakciju na visokom nivou nekog drugog dijagrama toka koji detaljnije prikazuje kako se izračunava neka apstraktna funkcija. Elipsa na najnižem nivou je funkcija čije efekte, kao što su preduslovi, posledična stanja i izuzeci, možemo da specifikujemo oslanjajući se na drugu notaciju (na primer, tekst, matematičke funkcije, tragovi događaja) u posebnom ali spregnutom dokumentu.



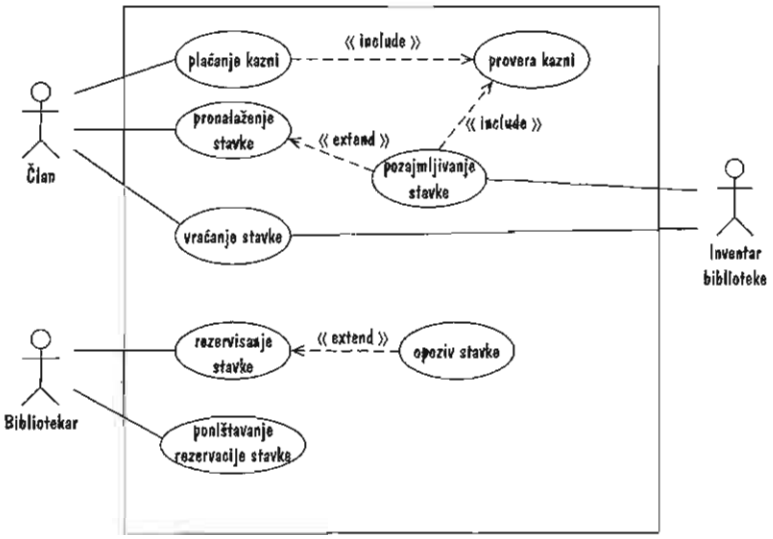
SLIKA 4.14 Dijagram toka podataka za problem biblioteke

Jedna od dobrih strana dijagrama toka podataka jeste da obezbeđuju intuitivne modele funkcija predloženog sistema na visokom nivou apstrakcije i međusobne zavisnosti podataka različitih procesa. Stručnjaci za domene smatraju da ih je jednostavno čitati i razumeti. Međutim, dijagram toka podataka ponekad nije dovoljno jednoznačan za projektanta softvera koji nedovoljno poznaje problem koji modeluje. Posebno zbog postojanja više načina interpretacije DFD procesa koji imaju više ulaznih tokova (na primer proces `Uzimanje pozajmice`) javljaju se dileme: da li su svi ulazi stvarno potrebni za obradu funkcije, da li je potreban samo jedan ulaz, ili je potreban neki podskup ulaza? Slično tome, DFD procesi koji imaju više izlaznih tokova mogu se tumačiti na više načina. Tu se javljaju dileme: da li se svi izlazi generišu svaki put kada se proces izvrši, da li se generiše samo jedan izlaz, ili se generiše neki podskup? Takođe nije očigledno da li dva toka podataka sa istom oznakom imaju iste vrednosti. Na primer, nije jasno da li `Vraćene stavke` koje teku iz `Vraćanja` do `Zapisa o pozajmicama` predstavljaju isto što i `Vraćene stavke` koje teku od `Vraćanja` ka `Obradi kazni`. Iz tih razloga, DFD-ovi su najpogodniji za korisnike kojima je blizak domen primene, kao i u slučajevima kada detalji nisu toliko bitni, recimo u modelima nastalim na početku procesa modelovanja.

Primer: Slučajevi korišćenja

UML-ov dijagram slučajeva korišćenja (OMG 2003) sličan je dijagramu toka podataka na visokom nivou koji opisuje uočljive, korisnički inicirane funkcije, u formi interakcija između sistema i njegovog okruženja. Veliki „pravougaonik" predstavlja granice sistema. Stilizovani likovi opisuju učesnike, ljudska bića i sisteme, dok svaka elipsa unutar pravougaonika predstavlja slučaj korišćenja, tj. neke glavne funkcionalnosti koje se traže i odgovarajuće varijante. Linija između učesnika i slučaja korišćenja ukazuje da učesnik učestvuje u slučaju korišćenja. Slučajevi korišćenja nisu namenjeni modelovanju svih zadataka koje sistem treba da obezbedi. Umesto toga, oni se koriste za specifikovanje

korisničkog pogleda na suštinsko ponašanje sistema. Prema tome, slučajevi korišćenja modeluju samo funkcionalnost sistema koju može da inicira neki učesnik u datom okruženju. Na primer, na slici 4.15, ključna korišćenja biblioteke uključuju pozajmljivanje knjiga, vraćanje pozajmljene knjige, kao i plaćanje bibliotečke kazne.



SLIKA 4.15 Slučajevi korišćenja biblioteke

Svaki slučaj korišćenja obuhvata više mogućih scenarija, od kojih su neki uspešni, a neki ne, ali se svi odnose na neku upotrebu sistema. Van samih dijagrama slučajeva korišćenja, slučajevi korišćenja i njihove varijante detaljno su tekstualno opisani kao tragovi događaja. Svaki slučaj korišćenja identifikuje prethodne uslove i alternativno ponašanje, ako prethodni uslovi nisu zadovoljeni, na primer, traženje izgubljene knjige. Tu su i naknadni uslovi koji sumiraju efekte slučajeva korišćenja, kao i uobičajeni uspešni scenario, koji podrazumeva redosled koraka koje izvodi učesnik ili sistem. Potpuno detaljan slučaj korišćenja određuje sve moguće varijacije svakog koraka u uobičajenom scenariju, uključujući i ispravno ponašanje i greške. On takođe opisuje moguće scenarije koji potiču od ispravnih varijacija i grešaka od kojih je moguć oporavak. Ako postoji redosled koraka koji je zajednički za više slučajeva korišćenja, taj redosled može da se izdvoji kao podslučaj koga osnovni slučaj poziva, slično pozivu procedure. Na dijagramu slučajeva korišćenja, povlačimo isprekidanu liniju od osnovnog slučaja do svakog od njegovih podslučajeva i označavamo te strelice stereotipom `<<include>>` („uključuje")[4]. Slučaju korišćenja može takođe biti pridružen podslučaj koji proširuje njegovu funkcionalnost. Na dijagramu slučajeva korišćenja, povlačimo isprekidanu liniju od podslučaja koji predstavlja proširenje do osnovnog slučaja korišćenja i označavamo je stereotipom `<<extend>>` („proširuje"). Primeri strereotipa dati su na slici 4.15.

4 Na jeziku UML, stereotip predstavlja mehanizam metajezika za proširenje notacije modela, omogućavajući korisniku da notaciju proširi novom <<ključnom reči>>.

Funkcije i odnosi (relacije)

Notacije koje smo do sada razmatrali služile su za predstavljanje i definisanje odnosa (relacija). U tim notacijama koriste se standardni oblici, linije i strelice za prikazivanje entiteta, odnosa i karakteristika obuhvaćenih problemom koji se modeluje. Nasuprot tome, ostale tri notacije koje smo razmatrali imaju jaču matematičku podlogu, i koristimo ih za izradu matematičkih modela zahteva. Matematički zasnovane specifikacije i tehnike projektovanja, zovu se **formalni metodi** ili **pristupi**, i njih koriste mnogi softverski inženjeri koji projektuju sistme kod kojih je bezbednost od izuzetne važnosti, tj. sisteme čiji otkaz može da utiče na zdravlje i bezbednost ljudi koji ih koriste ili koji su u blizini. Na primer, Defence Standard 00-56, nacrt britanskog standarda za gradnju izuzetno bezbednih sistema, zahteva da formalna specifikacija i projektna rešenja koja će se koristiti pokazuju neophodnu funkcionalnost, pouzdanost i bezbednost. Pobornici iznose argumente da su matematički modeli precizniji sa nižim stepenom višeznačnosti nego drugi modeli, i oslanjaju se na sistematičniju i prefinjeniju analizu i verifikaciju. U stvari, mnoge formalne specifikacije se mogu automatski proveriti u odnosu na: doslednost i kompletnost, zatim na determinisanost, dostižnost stanja i tip ispravnosti. Matematički dokazi otkrili su značajne probleme u specifikacijama zahteva, a tu je probleme mnogo lakše ispraviti nego ako se otkriju tokom testiranja. Na primer, Pfleeger i Hatton (1997) podenli su izveštaj o učesnicima u razvoju softvera koji su koristili formalne metode za specificiranje i ocenjivanje kompleksnih zahteva u vezi sa komunikacijom u sklopu sistema za podršku kontroli letenja. Ispitivanje formalne specifikacije na samom početku ukazalo je na glavne probleme koji su ispravljeni mnogo pre nego što je projektovanje počelo. Na taj način se smanjio rizik i uštedelo u vremenu razvoja. Na kraju ovog poglavlja videćemo kako je ispitivanje formalne specifikacije moglo da pomogne da se uoče problemi sa raketom Ariane-5.

Neke formalne paradigme modeluju zahteve ili ponašanja softvera kao skup matematičkih **funkcija** ili **odnosa (relacija),** koji kada su zajedno preslikavaju ulaz sistema u izlaz sistema. Neke funkcije određuju izvršna stanja sistema, dok druge funkcije određuju izlaz. Relacija se koristi umesto funkcija uvek kada se ulazna vrednost preslikava na više izlaznih vrednosti. Na primer, problem obrtne rampe možemo da predstavimo pomoću dve funkcije: jedna funkcija vodi računa o stanju rampe, i preslikava trenutno stanje i ulazni događaj u sledeće stanje, dok druga funkcija određuje izlaz obrtne rampe, zasnovan na trenutnom stanju i ulaznom događaju:

$$SledećeStanje(s,e) = \begin{cases} otključano & s = zaključano \ \ AND \ \ e = novčić \\ rotiranje & s = otključano \ \ AND \ \ e = guranje \\ zaključano & (s = rotiranje \ \ AND \ \ e = rotirano \\ & OR \ \ (s = zaključano \ \ AND \ \ e = žeton) \end{cases}$$

$$Izlaz(s,e) = \begin{cases} zujanje & s = zaključano \ \ AND \ \ e = žeton \\ < ništa > & U \ protivnom \end{cases}$$

Zajedno, gornje funkcije su semantički ekvivalentne grafičkom modelu automata obrtne rampe prikazanom na slici 4.8.

Iz razloga što preslikava svaki ulaz u jedinstveni izlaz, funkcija je po definiciji dosledna. Funkcija koja određuje izlaz za *svaki* jasno različit ulaz, naziva se totalna funkcija i po svojoj definiciji je kompletna. Prema tome, funkcionalne specifikacije se oslanjaju na sistematske i jednostavne testove doslednosti i kompletnosti.

(događaj) pozajmljivanje	T	T	T	N	N	N	N	N
(događaj) vraćanje	N	N	N	T	T	N	N	N
(događaj) rezervacija	N	N	N	N	N	T	T	N
(događaj) poništavanje rezervacije	N	N	N	N	N	N	N	T
knjiga izdata	N	T	-	-	-	N	T	N
knjiga rezervisana	-	-	-	N	T	-	-	-
član.kazne > $0.00	N	-	T	-	-	-	-	-
Preračunavanje datuma dospeća	X						X	
Stavljanje knjige u orman				X				X
Stavljanje knjige na policu sa rezervacijama					X	X		
Slanje obaveštenja o povlačenju							X	
Odbijanje događaja		X	X					

SLIKA 4.16 Tabela odlučivanja na primeru funkcionisanja biblioteke

Primer: Tabele odlučivanja

Tabela odlučivanja (Hurley 1983) predstavlja tabelarni prikaz funkcionalne specifikacije koja preslikava događaje i uslove u odgovarajuće odgovore ili akcije. Kažemo da je specifikacija *neformalna* jer ulazi (događaji i uslovi) i izlazi (akcije) mogu biti izraženi prirodnim jezikom, kao matematički izrazi ili kombinovano.

Slika 4.16. prikazuje tabelu odlučivanja za funkcije `uzimanje knjige iz biblioteke`, `vraćanje`, `rezervisanje` kao i `poništavanje rezervacije`. Svi mogući ulazni događaji (tj. pozivanje funkcija), uslova i akcija navedeni su uz levu ivicu tabele, sa ulaznim događajima i uslovima navedenim iznad horizontalne linije, dok su akcije navedene ispod linije. Svaka kolona predstavlja pravila koja preslikavaju skup uslova u odgovarajuće rezultate. „T" u ćeliji znači da je ulazni uslov ispunjen (tačan), „F" znači da je ulazni uslov netačan, a crtica ukazuje da vrednost uslova nije bitna. „X" na dnu tabele znači da akcija datog reda treba da se izvede uvek kada je ispunjen odgovarajući ulazni uslov. Prema tome, kolona 1 predstavlja situaciju gde član biblioteke želi da pozajmi knjigu, knjiga još uvek nije izdata, a član nema neizmirenih kazni. U toj situaciji odobrava se izdavanje knjige i izračunava se datum kada knjiga treba da se vrati. Slično tome, kolona 7 ilustruje slučaj gde postoji zahtev za rezervisanje knjige, ali je knjiga trenutno izdata. U tom slučaju se knjiga povlači, a datum vraćanja knjige se ponovo preračunava prema povlačenju.

Ovakvo predstavljanje podataka može da proizvede veoma velike tabele, jer je broj uslova koji treba da se razmotri jednak broju kombinacija ulaznih uslova. To jest, ako postoji n ulaznih uslova, postoji $2n$ mogućih kombinacija uslova. Na sreću, mnoge kombinacije preslikavaju se u isti skup rezultata i mogu da se kombinuju u jednoj koloni. Neke kombinacije uslova se uzajamno isključuju (na primer, knjiga ne može da bude izdata i vraćena u istom trenutku). Analizom tabela odlučivanja na ovaj način, možemo da smanjimo njihovu veličinu i da ih učinimo razumljivim.

Šta još možemo da kažemo o specifikaciji zahteva koju izražavamo pomoću tabela odlučivanja? Možemo lako da proverimo da li su razmotrene sve moguće kombinacije uslova, da bismo na osnovu toga odredili da li je specifikacija kompletna. Možemo da ispitamo doslednost tabela, identifikujući višestruke instance istih ulaznih uslova, i da eliminišemo eventualno sukobljavanje izlaza. Možemo takođe da tražimo šablone ponašanja u tabeli da bismo videli stepen korelacije pojedinačnih ulaznih uslova i pojedinačnih akcija. Takvo pretraživanje bi bilo gotovo nemoguće na specifikacijama modelovanim pomoću matematičkih funkcija sa tradicionalnom tekstualnom notacijom.

Primer: Parnasove tabele

Parnasove tabele (Parnas 1992) služe tabelarnom predstavljanju matematičkih funkcija ili odnosa (relacija). Poput tabela odlučivanja, i u Parnasovim tabelama se koriste redovi i kolone za rastavljanje funkcijske definicije u njene različite slučajeve. Svaki zapis u tabeli određuje ulazni uslov koji delimično definiše neki slučaj ili od koga zavisi izlazna vrednost nekog slučaja. Za razliku od tabela odlučivanja, ulazi i izlazi Parnasove tabele jesu čisto matematički izrazi.

Da bismo videli kako funkcionišu Parnasove tabele, razmotrićemo sliku 4.17. Redovi i kolone definišu `Računanje datuma vraćanja`, uobičajene radnje za jednu biblioteku. Informacija je predstavljena kao normalna tabela, što je uobičajeni tip Parnasove tabele. Zaglavlja kolona i redova jesu predikati koji se koriste za određivanje slučajeva, dok u stavkama unetim u tabelu nalaze se mogući rezultati funkcije. Prema tome, svaka stavka uneta u tabelu predstavlja jasno odvojen slučaj u definiciji funkcije. Na primer, ako je događaj obnavljanje pozajmice (zaglavlje kolone) *i* publikacija koja se pozajmljuje je rezervisana (zaglavlje kolone) *i* član koji podnosi zahtev nema neizmirenih kazni (zaglavlje reda), tada se datum dospeća izračunava kao broj dana `publikacija.period pozajmice rezervacije` počev od `Danas`. „X" ukazuje na to da operacija nije važeća pod navedenim uslovima. U drugim specifikacijama, „X" može da znači da je kombinacija uslova neizvodljiva. Obratimo pažnju na to kako su zaglavlja kolona i redova strukturisani da bi bile obuhvaćene sve moguće kombinacije uslova koji mogu da utiču na izračunavanje datuma vraćanja pozajmljene knjige. (Simbol ← ima značenje „ne", tako da `←publikacija.U stanju(rezervacija)` znači da publikacija nije rezervisana.)

Izraz **Parnasove tabele** u stvari se odnosi na skup tipova tabela i strategija pravljenja skraćenica u cilju organizovanja i pojednostavljivanja izraza funkcija i relacija. Postoji još jedan tip tabele koji ćemo spomenuti, a to je *invertovana tabela*, koja je sličnija konvencionalnoj tabeli. U invertovanim tabelama su uslovi slučaja navedeni kao izrazi u zaglavljima redova i u ćelijama tabele, a rezultati funkcija navedeni su u zaglavljima kolona, na vrhu ili na dnu tabele. U opštem slučaju, cilj onoga ko pravi specifikaciju jeste da odabere ili napravi tabelu u takvom formatu koji će na jednostavan i kompaktan način predstaviti funkcije ili odnose. Tabelarna struktura olakšava onima koji pregledaju tabelu da provere da li je specifikacija kompletna (tj. da nema izostavljenih slučajeva) i dosledna (tj. da nema dupliranih slučajeva). Lakše je definiciju funkcije pregledati slučaj po slučaj, nego ispitivati celu specifikaciju odjednom.

Funkcionalna specifikacija koja je izražena pomoću Parnasovih tabela najbolje se dekomponuje na pojedinačne funkcije po izlaznoj promenljivoj. Svaka funkcija definiše vrednost svoje izlazne promenljive za svaki ulazni događaj i za svaki uslov po entitetu ili drugim promenljivima. Prednost strukture ovog modela u odnosu na model automata je u tome što su definicije svake izlazne promenljive lokalizovane u jasno odvojenoj tabeli, umesto da su rasute širom modela kao akcije ili prelazi stanja.

	događaj ∈ {pozajmljivanje, obnavljanje}		događaj = povlačenje
	publikacija.U stanju	publikacija.U stanju	
član.kazna = 0	publikacija.period pozajmice rezervacije	publikacija.period pozajmice	Min(datum dospeća, publikacija. period povlačenaj)
član.kazna > 0	X	X	X

SLIKA 4.17 (Normalna) Parnasova tabela operacije Izračunavanje datuma za vraćanje knjige

Logika

Sa izuzetkom ER dijagrama, notacije koje smo razmatrali do sada bile su zasnovane na modelu i za njih kažemo da su operativne. Notacija je **operativna** ako se koristi za opisivanje problema ili predloženog softverskog rešenja u zavisnosti od ponašanja u datoj situaciji: kako softverski sistem treba da odgovori na različite ulazne događaje, kako obrada treba da teče od jednog koraka ka drugom, i šta sistem treba da proizvede na izlazu pod različitim uslovima. Rezultat je model ponašanja *zasnovan na slučaju,* koji je posebno koristan pri formulisanju odgovora na pitanja o tome šta treba da bude željeni rezultat u pojedinačnim situacijama. Na primer, šta je sledeće stanje ili izlaz sistema za dato trenutno stanje, ulazni događaj, dovršenje procesa i vrednosti promenljivih. Takvi modeli takođe pomažu čitaocu da vizualizuje globalno ponašanje, u formi putanja koje predstavljaju moguće trase izvršavanja u modelu.

Operativne notacije su manje efikasne pri izražavanju globalnih osobina ili ograničenja. Pretpostavimo da modelujemo semafor, i želimo da osiguramo da svetla koja kontrolišu saobraćaj u ukrštenim pravcima nisu nikada zelena u isto vreme, kao i da su svetla u oba smera periodično zelena. Možemo da napravimo operativni model koji implicitno izlaže ta ponašanja tako da sve putanje u modelu zadovoljavaju te osobine. Međutim, za nas je nejasno da li su te osobine zahtevi koje traba zadovoljiti shodno specifikaciji, ili su to slučajne posledice odluka donetih prilikom modelovanja. Tu dilemu nemamo jedino ako je model **zatvoren**, pri čemu se misli na to da model izražava svo željeno ponašanje i da je eventualna implementacija koja izvršava druge funkcije pogrešna.

Nasuprot tome, globalne osobine i ograničenja bolje se izražavaju pomoću opisnih notacija, kao što je logika. **Opisna** notacija jeste notacija koja opisuje problem ili predloženo rešenje zavisno od njegovih osobina ili nepromenljivih ponašanja. Na primer, ER dijagrami su opisni po tome što izražavaju svojstva odnosa između entiteta. **Logika** se sastoji od jezika za izražavanje osobina, i skupa pravila zaključivanja za izvođenje novih, konsekventnih osobina iz navedenih osobina. U matematici, logički izraz[5], koji se zove **formula**, daje rezultat *istinito* ili *neistinito*, zavisno od vrednosti promenljivih koje se pojavljuju u formuli. Nasuprot tome, kada se logika koristi za izražavanje osobina softverskog problema ili sistema, osobina je tvrdnja o problemu ili sistemu koja treba da bude *istinita*. Kao takva, osobinu definišu samo vrednosti onih njenih promenljivih za koje izraz ima vrednost *istinito*.

Postoji više logičkih varijanti koje se razlikuju po tome koliko je izražajna njihova notacija, ili koja pravila zaključivanja koriste. Logika koja se uobičajeno koristi za izražavanje osobina softverskih zahteva je **logika prvog reda** (*first-order logic*, FOL) koja obuhvata sledeće tipove promenljivih: konstante, funkcije, predikate, relacione operatore < i >, jednakost, logičke veznike $\wedge$ (i), $\vee$ (ili), $\leftarrow$ (ne), $\Rightarrow$ (implikaciju) i $\Leftrightarrow$ (logičku ekvivalenciju); kao i kvantifikatore $\exists$ (postoji) i $\forall$ (za svako). Razmotrimo sledeće promenljive za problem obrtne rampe, sa njihovim inicijalnim vrednostima:

```
broj_novčića : integer := 0;                    /* broj ubačenih novčića     */
broj_stavki : integer := 0;                     /* broj polurotacija
                                                   obrtne rampe              */
rampa : {zaključana, otključana} := zaključana;  /* da li je rampa zaključana */
```

5 Logiku možemo da shvatimo kao funkcije koje preslikavaju izraze u skup mogućih vrednosti. Dalje, *n*-arna logika preslikava se u skup *n* vrednosti. Binarna logika preslikava izraze u {istinito, neistinito}, ali u opštem slučaju *n* može biti veće od 2. U ovoj knjizi, pretpostavićemo da je *n* jednako 2, osim ako nije drugačije navedeno.

```
može_da_uđe : boolean := false;          /* da li neko može da uđe     */
ubacivanje_novčić : boolean := false;    /* događaj ubacivanja novčića*/
guranje : boolean := false;              /* obrtna rampa je gurnuta
                                            dovoljno jako da izvrši
                                            jednu polurotaciju         */
```

Slede primeri osobina obrtne rampe u odnosu na te promenljive, izražene logikom prvog reda:

```
broj_novčića ≥ broj_stavki
(broj_novčića > broj_stavki) ⇔ (rampa = otključana)
(rampa = zaključana) ⇔ ←može_da_uđe
```

Zajedno, te formule predstavljaju tvrdnju da broj prolazaka kroz obrtnu kapiju nikada ne prelazi broj ubačenih novčića, a uvek kada broj ubačenih novčića premaši broj ulazaka, rampa se otključava da bi omogućila sledećoj osobi da uđe u ograđeni prostor. Napomenimo da te osobine ne govore ništa o tome kako promenljive menjaju vrednosti, kao, recimo, o načinu uvećavanja broja ubačenih novčića. To opisuje drugi deo specifikacije. Gornje osobine jednostavno osiguravaju da kako god da se promenljive promene, te vrednosti uvek zadovoljavaju ograničenja koja propisuje formula.

Temporalna logika uvodi dodatne logičke veznike kojima se ograničava način na koji promenljive mogu da menjaju vrednost tokom vremena, tj. preciznije, na više tačaka u toku izvršavanja. Na primer, veznici temporalne logike mogu da izraze bliske promene, kao što su dodeljivanje promenljivih (na primer, rezultat događaja `ubacivanje_novčića` je da se promenljiva `broj_novčića` inkrementira za 1). Oni mogu da izraze i buduće vrednosti promenljivih (na primer, nakon događaja `ubacivanje_novčića`, promenljiva `može_da_uđe` zadržava istinitu vrednost sve dok se ne desi događaj `guranje`). To ponašanje možemo da modelujemo pomoću logike prvog reda, dodavanjem parametra vreme u svaku promenljivu modela, uz pitanje o vrednosti promenljive u određenom trenutku. Međutim, temporalna logika omogućava još jezgrovitije predstavljanje promenljivog ponašanja, jer dozvoljava izmene vrednosti promenljivih i uvodi posebne operatore temporalne logike.

Kao i kod opšte logike, postoje mnoge varijante temporalne logike koje se razlikuju u operatorima koje uvode. Sledeći operatori (operatori linearnog vremena) ograničavaju buduće vrednosti promenljivih, tokom jedne putanje izvršavanja:

□ f ≡ f je *istinito* sada i tokom ostatka izvršavanja;

◊ f ≡ f je *istinito* sada ili u nekoj budućoj tački izvršavanja;

○ f ≡ f je *istinito* u sledećoj tački izvršavanja;

f W g ≡ f je *istinito* sve dok je g *istinito*, ali g ne mora nikada da bude istinito.

Sada ćemo vremenske osobine obrtne rampe izraziti pomoću temporalne logike:

```
□ (ubacivanje_novčić ⇒ ○(može_da_uđe W guranje))
□ (ubacivanje_novčić ∧ broj_novčića=n) ⇒ ○(broj_novčića = n+1))
```

Osobine se često koriste za pojačavanje specifikacija zasnovanih na modelu, bilo ograničavanjem dopuštenog ponašanja u sklopu modela, ili jednostavno izražavanjem

redundantnih, ali ne i očiglednih globalnih osobina specifikacije. U prvom slučaju, osobina određuje ponašanje koje nije izraženo u modelu, a željeno ponašanje predstavlja konjunkciju modela i osobine. U drugom slučaju, osobina ne menja specifikovano ponašanje, ali može da potpomogne razumevanje modela izričito navodeći ponašanja koje se inače podrazumevaju. Redundantne osobine takođe pomažu kod verifikacije zahteva, obezbeđujući očekivane osobine modela koje je neophodno proveriti prilikom pregleda.

Primer: Jezik ograničenja objekta (OCL)

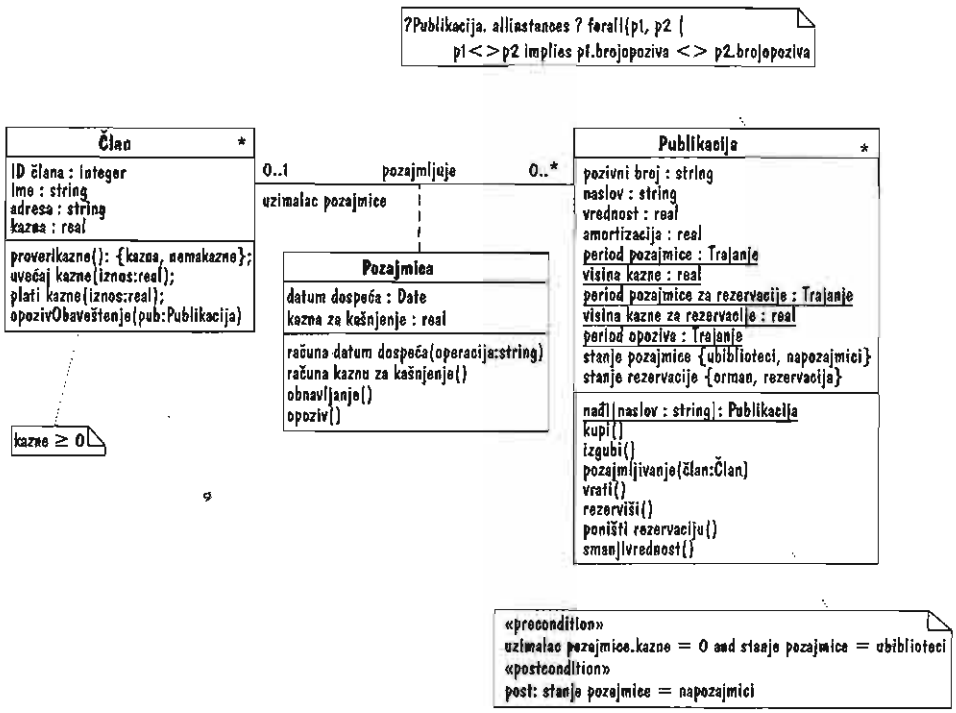
Jezik ograničenja objekta (*Object Constraint Language*, OCL) predstavlja pokušaj stvaranja matematički preciznog jezika ograničenja kojim i oni koji nisu matematičari mogu lako da se izražavaju, da ga čitaju i razumeju, što znači i naručioci. Jezik je posebno projektovan za izražavanje ograničenja u sklopu objektnih modela (tj. ER dijagramima), pri čemu se uvode jezičke konstrukcije za navigaciju od jednog objekta do drugog preko pridruženih putanja, zatim za bavljenje kolekcijama objekata i za formulisanje upita nad tipom objekata.

Delimični model klasa Biblioteka sa slike 4.5 pojavljuje se na slici 4.18, u kojoj su tri klase detaljnije prikazane i označene OCL ograničenjima. Ograničenje koje se nalazi sasvim na levoj strani je invarijanta klase `Član`, i određuje da kazna člana ne može biti negativna (tj. biblioteka uvek vrši izmenu ako uplata člana premaši odgovarajuću kaznu). Ograničenje sasvim na vrhu predstavlja invarijantu klase `Publikacija`, i određuje da je pozivni broj jedinstven. To ograničenje uvodi:

- konstrukciju *sve instance* (*allinstances*), koja vraća sve instance klase `Publikacija`;
- simbol → koji primenjuje atribut ili operaciju na operande sa desne strane na sve objekte na levoj strani;
- konstrukcije *za sve* (*forall*), *i* (*and*) i *podrazumeva* (*implies*), koje odgovaraju operatorima logike prvog reda, koji su prethodno opisani.

Prema tome, ograničenje bukvalno kazuje da za bilo koja dva izdanja, `p1` i `p2`, koje vraća *allinstances*, ako `p1` i `p2` nisu ista izdanja, onda oni imaju različite pozivne brojeve. Treće ograničenje, pridruženo metodi `pozajmljivanje()`, izražava prethodne i naknadne uslove operacija. Jedan od prethodnih uslova tiče se atributa u klasi `Član`, kome se pristupa posredstvom asocijacije `pozajmljivanje`. Objektu klase član možemo pristupiti bilo posredstvom imena uloge, `pozajmljivač`, ili preko imena klase napisanog malim slovima, u slučaju da uz dalji kraj asocijacije nema imena uloge. Ako je kardinalitet asocijacije veći od 0..1, onda navigacija posredstvom asocijacije vraća kolekciju objekata, i tada za pristup atributima objekta koristimo notaciju →, umesto notaciju sa tačkom.

Iako nije originalno projektovan kao deo UML-a, OCL je sada tesno povezan sa UML-om i čini deo UML standarda. OCL može da dopuni mnoge UML modele. Na primer, može da se koristi za izražavanje invarijanti, prethodnih uslova i naknadnih uslova u dijagramima klasa ili invarijanti i uslova prelaza u dijagramima prelaza stanja. OCL može takođe da izrazi uslove vezane za događaje u dijagramima redosleda (Warmer i Kleppe 1999). OCL komentarisanje UML-a zahteva relativno detaljne modele klasa, zajedno sa tipovima atributa, skupom operacija, imenima uloga, kardinalitetima, kao i sa enumeracijom stanja iz klasinog dijagrama prelaza. OCL izrazi mogu da se pojave u UML dijagramu kao UML napomene, ili da budu navedeni u pratećem dokumentu.



SLIKA 4.18 Bibliotečke klase iskomentarisane OCL svojstvima

Primer: Z

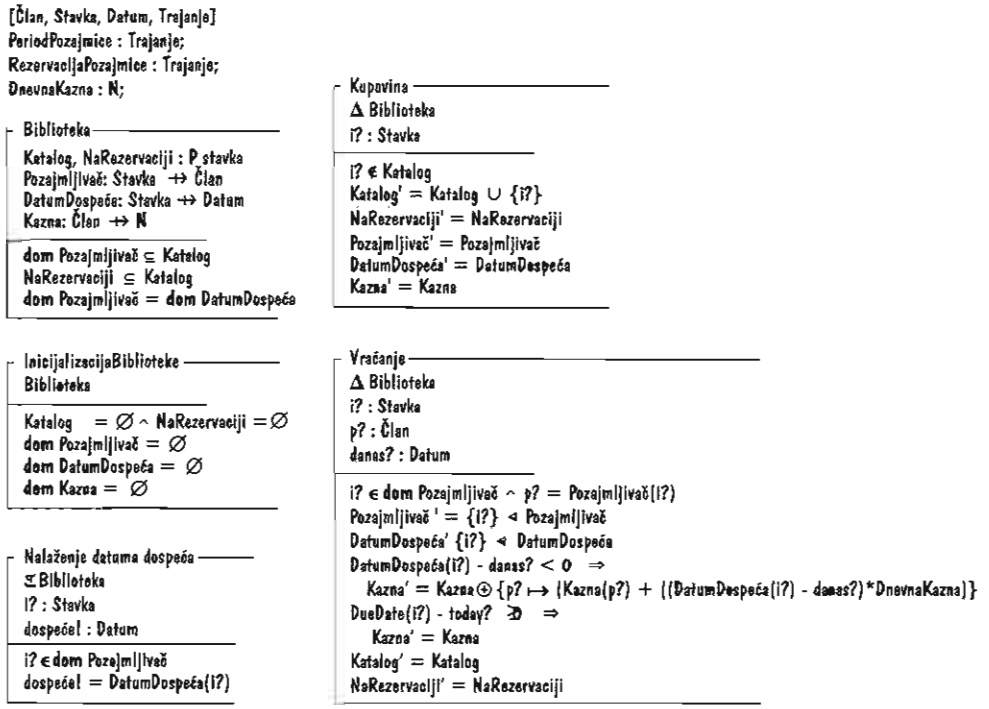
Z (izgovara se „zed") je formalni jezik za specifikaciju zahteva koji definicije promenljivih na osnovu teorije skupova strukturiše u kompletan model apstraktnih tipova podataka za problem i koristi logiku za izražavanje prethodnih i naknadnih uslova za svaku operaciju. Z koristi apstrakcije softverskog inženjerstva za dekompoziciju specifikacije na module kojima je lakše upravljati a koji se nazivaju **šeme** (Spivey 1992). Odvojene šeme specifikuju:

- stanje sistema u smislu tipiziranih promenljivih i njihovih invarijanti vrednosti;
- početno stanje sistema (tj. početne vrednosti promenljivih);
- operacije.

Više od toga, Z pruža preciznost matematičke notacije i sve njene pogodnosti, kao što je mogućnost evaluacije specifikacije pomoću dokazivanja ili automatskih provera.

Slika 4.19 prikazuje deo primera biblioteke u Z notaciji. Na primer, `Član`, `Stavka`, `Datum` i `Trajanje` su osnovni tipovi koji odgovaraju njihovim oznakama iz realnog sveta koje su date u redosledu kojim smo ih naveli. (Videti podsetnik 4.5 za više detalja o označavanju.) Šema `Biblioteka` deklariše da se problem sastoji od `Kataloga` i skupa stavki `NaRezervaciji`, gde su deklarisani kao nadskupovi (**P**) `Stavki`. Te deklaracije znače da vrednosti `Kataloga` i `NaRezervaciji` mogu da se promene tokom izvršavanja na bilo koji podskup `Stavki`. Šema takođe deklariše parcijalna preslikavanja (⇓) koja evidentiraju `Pozajmljivača` i `DatumeDospeća` za podskup `Stavki` koje su na pozajmici, i evidentiraju `Kazne` za podskup `Članova` koji imaju neizmirene kazne. Domen (**dom**)

parcijalnog preslikavanja je podskup entiteta koji se trenutno preslikavaju. Stoga tvrdimo da podskup izdatih stavki treba da bude identičan podskupu stavki koje imaju datum dospeća. Šema `InicijalizacijaBiblioteke` inicijalizuje sve promenljive i funkcije na vrednost prazan skup. Sve preostale šeme odgovaraju operacijama u sklopu biblioteke.



SLIKA 4.19 Parcijalna Z specifikacija problema biblioteke

PODSETNIK 4.5 ZASNIVANJE ZAHTEVA NA REALNOM SVETU

Jacksonov savet (Jackson 1995) je da zasnivanje zahteva na realnom svetu ide dalje od poruke da se zahtevi izražavaju i specifikuju zavisno od pretpostavljenog okruženja sistema. Jackson osporava činjenicu da jedan model zahteva obuhvata primitivne termine koji nemaju formalno značenje (na primer Član, Pozajmica i Danas u našem primeru biblioteke) i da je jedini način za utvrđivanje značenja primitivnog termina njegovo povezivanje sa nekim fenomenom iz realnog sveta. On te opise naziva **odrednicama**, i pri tome razlikuje *odrednice* od *definicija* i *tvrdnji*. Definicije predstavljaju formalna značenja termina zasnovana na značenjima drugih termina koriščenih u modelu (na primer, definicija *knjiga na pozajmici*), dok *tvrdnje* opisuju ograničenja nad terminom (na primer, članovi mogu da pozajme samo one knjige koje nisu trenutno izdate). Ako model treba da ima bilo kakvo značenje u vezi sa nekim ponašanjem iz realnog sveta, njegovi primitivni termini moraju biti jasno i precizno povezani sa realnim svetom, a odgovarajuće odrednice „se moraju održavati kao sustinski deo dokumentacije zahteva" (Zave i Jackson 1997).

Odeljak na vrhu šeme operacija ukazuje na to da li operacija modifikuje (Δ) ili jednostavno generiše upit ($\in$) u stanje sistema, i identifikuje ulaz (?) i izlaz (!) operacije. Odeljak na dnu šeme operacija određuje prethodne i naknadne uslove operacije. U operacijama koje modifikuju stanje sistema, promenljive koje su napisane bez „prim", predstavljaju vrednosti promenljivih pre nego što je operacija izvedena, a promenljive sa „prim" predstavljaju vrednosti koje slede nakon operacije. Na primer, ulaz u operaciju `Kupovina` je nova bibliotečka `Stavka`, a prethodni uslov određuje da `Stavka` već nije u bibliotečkom `Katalogu`. Naknadni uslov ažurira `Katalog` tako da obuhvata i novu `Stavku`, i određuje da druge promenljive biblioteke ne menjaju vrednost (na primer, ažurirana vrednost `Kazni` jednaka je staroj vrednosti `Kazni`). Operacija `Vraćanje` je složenija. Ona kao ulaz uzima bibliotečku `Stavku` koja se vraća, `Člana` koji je uzeo na pozajmicu `Stavku`, kao i `Datum` vraćanja. Naknadni uslovi uklanjaju vraćenu stavku iz promenljivih `Pozajmljivač` i `DatumiDospeća`. Ta ažuriranja koriste Z simbol $\lhd$, „oduzimanje domena", za vraćanje potpreslikavanja odgovarajućih vrednosti pre operacije, sa izuzetkom elemenata čija je vrednost u domenu jednaka `Stavci` koja se vraća. Sledeća dva naknadna uslova jesu ažuriranja promenljive `Kazne`, pod uslovom da se `Član` zadužuje kaznom kod vraćanja `Stavke` nakon datuma dospeća pozajmice. Ta ažuriranja koriste simbol $\sum$, koji „preslikava" vrednost domena „na" opseg vrednosti, dok simbol $\oplus$ „nadjačava" preslikavanje funkcije, obično sa novim „preslikava-na" elementom. Prema tome, ako je datum `danas` kasniji nego `Datum Dospeća` vraćene `Stavke`, onda je `Kazna` koju dobija `Član` nadjačana novom vrednošću koja odražava staru kaznu uvećanu za novu kaznu. Poslednja dva naknadna uslova jesu da se vrednosti promenljivih `Katalog` i `NaRezervaciji` ne menjaju.

Algebarske specifikacije

Sa izuzetkom logike i OCL-a, sve ostale do sada razmatrane notacije rezultuju modelima koji sugerišu određene implementacije. Na primer:

- UML model klasa sugeriše koje klase treba da se pojave u konačnoj (objektno orijentisanoj) implementaciji;
- Specifikacija toka podataka sugeriše kako implementacija treba da se dekomponuje na module koji transformišu podatke;
- Model konačnih automata sugeriše kako reaktivni sistem treba da se dekomponuje na slučajeve;
- Z specifikacija sugeriše kako kompleksni tipovi podataka mogu da se implementiraju zavisno od skupova, redosleda ili funkcija.

Takvo prisustvo implementacije u sklopu specifikacije zahteva može uticati da projektant softvera, zbog pridržavanja *modela* specifikacije, pri formulisanju dizajna izostavi potencijalno bolja rešenja koja bi zadovoljila specifikovano *ponašanje*. Na primer, kao što ćemo videti u poglavlju 6, klase u UML dijagramu klasa mogu biti pogodne za jednostavno i jezgrovito iskazivanje *problema*, ali ista hijerarhija klasa prilikom projektovanja može da dovede do neefikasnog *rešenja* problema.

Potpuno drugačiji način gledanja na sistem je pogled iskazan kategorijama šta se događa kada se izvršavaju kombinacije operacija. Pogled kroz više operacija je glavna ideja koja stoji iza **algebarskih specifikacija**: specifikovanje ponašanja operacija specifikovanjem interakcija između parova operacija umesto modelovanja pojedinačnih ope-

racija. Trag izvršavanja je redosled operacija koje su izvedene od početka izvršavanja. Na primer, u slučaju problema obrtne rampe, posmatraćemo sam novi (*new*) početak rada obrtne rampe, koji je dat redosledom sledećih operacija:

```
new().novčić().guranje().rotacija().novčić().guranje().rotacija()....
```

ili u notaciji pomoću matematičkih funkcija:

```
...(rotacija(guranje(novčić(rotacija(guranje(novčić(new()))))))) ....
```

Specifikacioni aksiomi određuju efekte primenjivanja parova operacija na proizvoljni redosled operacija koje su već izvršene (gde je SEQ neki prefiks redosleda operacija):

```
broj_stavki(novčić(SEQ)) ≡ broj_stavki (SEQ)
broj_stavki(guranje(SEQ)) ≡ broj_stavki (SEQ)
broj_stavki(rotacija(SEQ)) ≡ 1 + broj_stavki (SEQ)
broj_stavki(new()) ≡ 0
```

Prva tri aksioma određuju ponašanje operacije `broj_stavki` kada se primeni na sekvence koje se završavaju odgovarajućim operacijama `novčić`, `guranje` i `rotacija`. Na primer, operacija `rotacija` ukazuje da je sledeći posetilac ušao u zoološki vrt, tako da `broj_stavki` primenjen na `rotacija(SEQ)` treba da bude za jedan veći od operacije `broj_stavki` primenjen na `SEQ`. Četvrti aksiom određuje osnovni slučaj operacije `broj_stavki`, kada se primeni na novu (`new`) obrtnu rampu. Ova četiri aksioma zajedno ukazuju da operacija `broj_stavki` vraća, za datu sekvencu, broj pojavljivanja operacije `rotacija`, bez dodatnog kazivanja o tome kako ta informacija može da se uskladišti ili izračuna. Slični aksiomi bili bi potrebni za specifikovanje ponašanja ostalih parova operacija.

Notacije pomoću algebarskih specifikacija nisu omiljene kod učesnika u razvoju softvera jer može da bude prilično složeno da se za definisani skup operacija konstruiše koncizan skup aksioma koji je potpun i konsistentan i ispravan! Uprkos svojoj složenosti, algebarske notacije su dodate u više formalnih specifikacionih jezika, radi podrške definisanju vlastitih apstraktnih tipova podataka u okviru specifikacija.

Primer: SDL podaci

SDL definicije **podataka** se koriste za pravljenje korisnički definisanih tipova podataka i parametrizovanih tipova podataka na **jeziku za specifikaciju i opisivanje – SDL** (*Specification and Description language*) (ITU 2002). Definicija SDL tipa podataka uvodi tip podataka koji se specifikuje, potpise svih operacija tog tipa podataka, kao i aksiome koji određuju kako parovi operacija vrše interakciju. Slika 4.20 prikazuje delimičnu SDL specifikaciju problema biblioteke, gde se sama biblioteka, tj. katalog publikacija, kao i pozamljivanje svake publikacije i status rezervacije, tretiraju kao kompleksni tipovi podataka. `NEWTYPE` uvodi tip podataka `Biblioteka`. Odeljak `LITERALS` deklariše konstante novog tipa podataka. U tom slučaju, `New` je vrednost prazne biblioteke. Odeljak `OPERATORS` deklariše sve operacije biblioteke, uključujući tipove parametara svakog operatora, kao i tip podataka koji se vraća kao rezultat. Odeljak `AXIOMS` određuje ponašanje parova operacija.

```
NEWTYPE Biblioteka
LITERALS New;
OPERATORS
   kupovina: Biblioteka, Stavka → Biblioteka;
   gubljenje: Biblioteka, Stavka → Biblioteka;
   pozajmljivanje: Biblioteka, Stavka → Biblioteka;
   vraćanje: Biblioteka, Stavka → Biblioteka;
   rezervisanje: Biblioteka, Stavka → Biblioteka;
   poništavanjerezervacije: Biblioteka, Stavka → Biblioteka;
   opoziv: Biblioteka, Stavka → Biblioteka;
   jesteUKatalogu: Library, Item → boolean;
   jesteNaPozajmici: Library, Item → boolean;
   jesteRezervisano: Library, Item → boolean;

/* generatori su New, kupovina, pozajmljivanje, rezervisanje */
```

```
AXIOMS
FOR ALL lib in Biblioteka {
  FOR ALL i, i2 in Stavka {
    gubljenje(New, i) ≡ ERROR;
    gubljenje(kupovina(lib, i), i2) ≡ if i = i2 then lib;
        else kupovina(gubljenje(lib, i2), i);
    gubljenje(pozajmljivanje(lib, i), i2) ≡ if i = i2 then gubljenje(lib. i2);
        else pozajmljivaaje(gubljeaje(lib, i2), i);
    gubljenje(rezervisanje(lib, i), i2) ≡ if i = i2 then gubljenje(lib. i2);
        else rezervisanje(gubljenje(lib, i2), i);
    vraćanje(New, I) ≡ ERROR
    vraćanje(kupovina(lib, i), i2) ≡ if i = i2 then kupovina (lib, I);
        else kupovina(return(lib, I2), I);
    vraćanje(pozajmljivanje(lib, I), I2) ≡ if i = i2 then lib;
        else pozajmljivanje(vraćanje(lib, i2), i);
    vraćanje(rezervisanje(lib, I), i2) ≡ rezervisanje(vraćanje(lib, i2), i);
        ...
    jesteUKatalogu(new, i) ≡ false;
    jesteUKatalogu(kupovine(lib, i), i2) ≡ if i = i2 then true;
        else jesteUKatalogu(lib, I2);
    jesteUKatalogu(pozajmljivanje(lib, i), i2) ≡ jesteUKatalogu(lib, i2);
    jesteUKatalogu(rezervisanje(lib, i), i2) ≡ jesteUKatalogu(lib, i2);
        ...
  }
}
ENDNEWTYPE Biblioteka;
```

SLIKA 4.20 Parcijalna SDL specifikacija podataka za problem biblioteke

Kao što je gore pomenuto, najteži deo konstrukcije algebarske specifikacije je definisanje skupa aksioma koji je potpun i dosledan i odražava željeno ponašanje. Posebno je teško osigurati doslednost aksioma jer su oni veoma međuzavisni, svaki aksiom učestvuje u specifikaciji dve operacije, a svaka operacija je određena skupom aksioma. Prema tome, izmena u specifikaciji obavezno utiče na više aksioma. Iskustvo koje pomaže da smanjimo broj aksioma, pomoću koga se smanjuje rizik od nedoslednosti, jeste da razdvojimo operacije na:

- **generatore**, koji pomažu da napravimo kanonsko predstavljanje definisanog tipa podataka;
- **manipulatore**, koji vraćaju vrednosti definisanog tipa podataka, ali nisu generatori;
- **upite**, koji ne vraćaju vrednosti definisanog tipa podataka.

Skup generatorskih operacija je minimalni skup operacija potreban za konstrukciju bilo koje vrednosti tipa podataka. To jest, svaka sekvenca operacija može biti redukovana na neku kanoničku sekvencu koju čine jedino generatorske operacije, tako da kanonička sekvenca predstavlja istu vrednost kao originalna. Na slici 4.20, odabiramo `New`, `kupovina`, `pozajmljivanje` i `rezervisanje` kao generatorske operacije, jer te operacije mogu da predstavljaju bilo koje stanje biblioteke, s obzirom na sadržaj kataloga biblioteke, kao i stanja pozajmica i rezervacija publikacija. Ova odluka ostavlja `gubljenje`, `vraćanje`, `poništavanje rezervacije` i `obnavljanje` kao manipulatorske operacije, jer to su preostale operacije koje imaju tip rezultata `Biblioteka`, dok na samom kraju preostaju `jesteUKatalogu`, `jesteNaPozajmici` i `jesteRezervisano`, kao naše operacije upita.

Drugi deo heuristike vezan je za obezbeđivanje aksioma koji određuju efekte primene negeneratorskih operacija na kanoničku sekvencu operacija. Kanoničke sekvence

se sastoje samo od generatorskih operacija, i zato ovaj korak znači da aksiome treba obezbeđivati samo za parove operacija, gde je svaki par negeneratorska operacija koja se primenjuje na primenjenu generatorsku operaciju. Svaki aksiom definiše način redukcije sekvence operacija na njen kanonički oblik. Primena manipulatorskih operacija na kanoničke sekvence obično rezultuje manjim kanoničkim sekvencama, jer manipulator obično poništava efekte ranijih generatorskih operacija, kao što je vraćanje pozajmljene knjige. Primena operacije upita, kao što je provera da li je knjiga izdata, vraća neke rezultate bez izmene stanja sistema koje je već kanoničko.

Aksiomi svake negeneratorske operacije definisani su rekurzivno:

1. Postoji osnovni slučaj koji određuje efekte svake negeneratorske operacije nad praznom, novom (`New`) bibliotekom. U našoj specifikaciji biblioteke (slika 4.20), gubljenje knjige iz prazne biblioteke predstavlja grešku (`ERROR`).
2. Postoji rekurzivni slučaj koji određuje efekte dve operacije na zajedničkim parametrima, kao što je kupovina i gubljenje iste knjige. U opštem slučaju, takve operacije utiču jedna na drugu. U tom slučaju, operacije poništavaju jedna drugu, a rezultat je stanje biblioteke, umanjen za dve operacije. Ako pogledamo slučaj gubljenja knjige koja je izdata, mi odbacujemo operaciju `pozajmljivanje` (jer ne postoji potreba da čuvamo zapise o pozajmicama za izgubljene knjige), a na ostatak sekvence primenjujemo operaciju `gubljenje`.
3. Postoji drugi rekurzivni slučaj koji primenjuje dva operatora na različite parametre, kao što je kupovanje i gubljenje različitih knjiga. Takve operacije ne utiču jedna na drugu, a aksiom određuje kako da kombinujemo efekte unutrašnje operacije sa rezultatima rekurzivne primene spoljnje operacije na ostatak stanja sistema. U slučaju kupovine i gubljenja različitih knjiga, zadržavamo efekte kupovine jedne knjige, a rekurzivno primenjujemo operaciju gubljenja na ostatak sekvence operacija koje su do tada izvršene.

Ne postoji potreba da se određuju aksiomi za parove negeneratorskih operacija, jer možemo da koristimo gornje aksiome za redukovanje kanoničkih oblika primene svake od negeneratorskih operacija, pre razmatranja sledeće negeneratorske operacije. Možemo da napišemo aksiome za parove generatorskih operacija. Na primer, možemo da odredimo da uzastopno izdavanje iste knjige predstavljaju grešku (`ERROR`). Međutim, mnoge kombinacije generatorskih operacija, kao što je uzastopno izdavanje različitih knjiga, neće rezultovati redukcijom kanoničkih oblika. Umesto toga, pišemo aksiome pretpostavljajući da mnoge operacije poseduju prethodne uslove koji ograničavaju primenu operacije. Na primer, u specifikaciji biblioteke pretpostavićemo (slika 4.20) da je neispravno `izdavanje` knjige koja je već na pozajmici. Uz tu pretpostavku, efekat vraćanja izdate knjige

```
vraćanje (pozajmljivanje(SEQ, i),i)
```

je da te dve operacije poništavaju jedna drugu, a rezultat je jednak SEQ. I ako ne napravimo tu pretpostavku, tada ćemo da napišemo aksiome tako da operacija `vraćanje` uklanja *sve* odgovarajuće operacije `pozajmljivanja`:

```
vraćanje (New, i) ≡ ERROR;
vraćanje(kupovina(lib, i), i2) ≡ if i=i2 then kupovina(lib, i);
                                 else kupovina(vraćanje(lib, i2), i);
vraćanje(pozajmljivanje(lib, i), i2) ≡ if i=i2 then vraćanje(lib, i2);
                                      else pozajmljivanje(vraćanje(lib, i2) i);
vraćanje(rezervisanje(lib, i, i2) ≡ rezervisanje(vraćanje(lib, i2), i);
```

Prema tome, efekat vraćanja izdate knjige je odbacivanje operacije pozajmljivanje i ponovna primena operacije `vraćanje` na ostatak sekvence, tako da može da ukloni sve nerelevantne pronađene operacije `pozajmljivanja`. Rekurzivni postupak završava se kada se operacija `vraćanje` primeni na odgovarajuću operaciju `kupovanje`, što označava početak postojanja knjige. Ako se operacija primeni na praznu biblioteku, dobija se greška (ERROR). Zajedno, ovi aksiomi određuju da operacija `vraćanja` uklanja iz stanja biblioteke sve tragove da je knjiga bila izdavana, što predstavlja željeno ponašanje. Specifikacija napisana u tom stilu poništava uzastopne kupovine, pozajmljivanja ili rezervisanje iste stavke.

4.6 ZAHTEVI I SPECIFIKACIONI JEZICI

Na ovom mestu se možda pitate kako su se razvili tako brojni tipovi softverskih modela, a da nijedan nije dominantna ili idealna notacija. Situacija je slična načinu rada arhitekte sa skupom nacrta, pri čemu svaki od njih odslikava određeni aspekt projekta zgrade (na primer, konstruktivni elementi, dovod grejanja, električna instalacija, cevi za vodu) ali jedino skup svih nacrta omogućava arhitekti da vizualizuje i saopšti celokupno projektno rešenje. Sve opisane notacije modeluju probleme iz različite perspektive: entitete i odnose, tragove događaja, stanja izvršavanja, funkcije, osobine i podatke. Prema tome, svaka predstavlja mogući izbor za modelovanje određenog aspekta softverskog problema. Uz praksu i iskustvo, naučićete da prosuđujete koje tačke gledišta i notacije najviše odgovaraju razumevanju i saopštavanju datog softverskog problema.

Zbog toga što svaka platforma poseduje vlastitu snagu, kompletna specifikacija može da se sastoji od više modela, gde svaki ilustruje različite aspekte problema. Iz tog razloga, većina praktičnih jezika za formulisanje i specifikaciju zahteva u stvari predstavlja kombinacije različitih notacionih platformi. Razumevanjem odnosa između specifikacionih jezika i notacionih platformi koje ti jezici koriste, počećemo da prepoznajemo sličnosti između različitih jezika i da cenimo suštinske načine u kojima se ti specifikacioni jezici razlikuju. Na kraju ovog poglavlja, razmotrićemo kriterijume za ocenjivanje i izbor specifikacionog jezika.

Unified Modeling Language (UML)

Objedinjeni jezik za modelovanje (***Unified Modeling Language*, UML**) (OMG 2003) jeste jezik koji je najpoznatiji po kombinovanju različitih notacionih platformi. UML standard obuhvata osam grafičkih notacija za modelovanje i jezik OCL za specifikovanje ograničenja. UML notacije koje se koriste pri definisanju i specifikovanju zahteva obuhvataju:

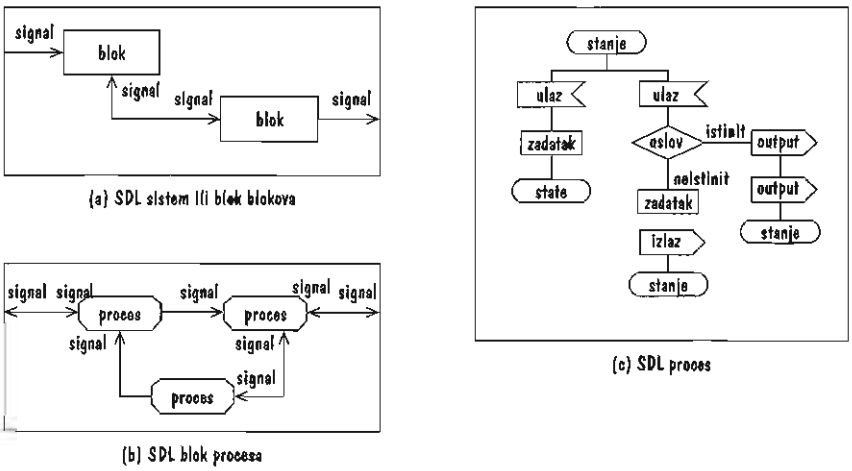
- **Dijagram slučajeva korišćenja (DFD visokog nivoa):** Dijagram slučajeva korišćenja se koristi na početku novog projekta za evidentiranje suštinskih funkcija na najvišem nivou apstrakcije koje treba da obezbedi budući proizvod. Tokom razrade detalja scenarija slučajeva korišćenja, moguće je identifikovati važne entitete koji figurišu u rešavanom problemu.
- **Dijagram klasa (ER dijagram):** Kao što je prethodno pomenuto, dijagram klasa je najvažniji model u grupi UML specifikacija, koji ističe entitete problema i njihove međusobne odnose. Ostatak UML specifikacionog modela obezbeđuje više detalja o tome kako se objekti klasa ponašaju i kako stupaju u interakcije. Sa podizanjem stepena razumevanja modelovanog problema dopunjavamo dijagram klasa detaljima kao što su dodatni atributi, klase atributa, operacije i oznake. U idealnom slučaju novi pogled na problem najčešće inicira izmene na nivou detalja, a ne na nivou entiteta ili njihovih veza.
- **Dijagram sekvence (trag događaja):** Dijagrami sekvenci jesu prvi modeli ponašanja koji opisuju tragove (staze) poruka koje se razmenjuju između instanci klasa. Oni su najpogodniji za dokumentovanje značajnih scenarija koji uključuju više objekata. Prilikom izrade dijagrama sekvenci, tražimo zajedničke podsekvence koje se pojavljuju u više dijagrama. Te podsekvence mogu da pomognu pri identifikaciji stanja (tj. početne i krajnje tačke podsekvence) u sklopu lokalnih ponašanja objekata.
- **Dijagram saradnje (trag događaja):** Dijagram saradnje ilustruje jedan trag ili više tragova događaja koje pokriva dijagram klasa. Dijagram saradnje predstavlja istu informaciju kao i dijagram sekvence. Razlika je u tome što dijagram sekvence, u sklopu razmatranog scenarija, stavlja akcenat na uređenje poruka u vremenu. S druge strane, dijagram saradnje se usredsređuje na odnose među klasama i tretira poruke kao detaljnu razradu tih odnosa, pri čemu se poruke predstavljaju strelicama koje povezuju klase u dijagramu klasa.
- **Dijagram prelaza stanja (model konačnog automata):** UML dijagram prelaza stanja određuje ponašanje instanci jedne klase u specifikovanom dijagramu klasa. Pre formulisanja dijagrama prelaza stanja klase (preciznije, dijagrama prelaza stanja reprezentativnog objekta klase), neophodno je identifikovati stanja u životnom ciklusu objekta, događaje koje ti objekti emituju i primaju od drugih objekata, redosled u kojem se ta stanja i događaji dešavaju, kao i operacije koje objekat poziva. Iz razloga što je takva informacija prilično detaljna, dijagrame prelaza stanja treba formulisati u kasnim fazama formulisanja zahteva, kada su svi detalji problema bolje shvaćeni.
- **OCL osobine (logika):** OCL izrazi predstavljaju svojstva elemenata modela (tj. objekata, atributa, događaja, stanja i poruka). OCL svojstva mogu da se koriste u bilo kom od prethodno opisanih modela, u cilju eksplicitnog navođenja podrazumevanog ponašanja modela ili nametanja ograničenja nad navedenim ponašanjem modela.

Većinu tih notacija razmatrali smo u prethodnim odeljcima, kao primere različitih notacionih platformi. U poglavlju 6 razmotrićemo detaljnije način funkcionisanja UML-a, u pogledu specifikacije i projektovanja, kroz primenu na primere iz realnog sveta.

Jezik za specifikaciju i opisivanje (SDL)

Jezik za specifikaciju i opisivanje (***Specification and Description Language***, **SDL**) (ITU 2002), predstavlja jezik standardizovan od strane Međunarodnog udruženja za telekomunikacije (*International Telecommunication Union*) za precizno specifikovanje ponašanja konkurentnih distribuiranih procesa, koji rade u realnom vremenu i koji međusobno komuniciraju posredstvom neograničenih redova za čekanje na poruke. SDL ima tri grafička dijagrama, pored algebarskih specifikacija za definisanje kompleksnih tipova podataka. To su:

- **SDL dijagram sistema (DFD):** SDL dijagram sistema, prikazan na slici 4.21(a), opisuje specifikacione blokove visokog nivoa i komunikacione kanale koji te blokove povezuju. Kanali su usmereni i obeleženi tipovima signala koji mogu da teku u oba smera. Poruka koja prolazi kroz **kanale** je *asinhrona*, što znači da ne možemo pretpostaviti kada će poslate poruke biti primljene. Naravno, poruke poslate istim kanalom biće primljene u redosledu slanja.
- **SDL blok dijagram (DFD):** Svaki SDL blok može da modeluje skup blokova nižeg nivoa, kao i kanale za kašnjenje poruka koji ih povezuju. Alternativno, može da modeluje i skup procesa na najnižem nivou koji komuniciraju preko signalnih puteva, kao što je prikazano na slici 4.21(b). **Signalni putevi** prosleđuju poruke *sinhrono*, tako da se poruke između procesa u istom bloku primaju trenutno. U stvari, ta razlika u mehanizmima komunikacije jeste činilac koji uzimamo u obzir prilikom odlučivanja kako da dekomponujemo ponašanje. Veoma spregnuti procesi koje je potrebno međusobno sinhronizovati treba da se nalaze u istom bloku.



SLIKA 4.21 Grafičke SDL notacije

- **SDL dijagram procesa (model konačnih automata):** SDL proces, prikazan na slici 4.21(c) predstavlja konačni automat, čiji su prelazi stanja sekvence jezičkih konstrukcija (ulaz, odluke, zadaci, izlazi) koji nastaju i nestaju pri konstrukciji stanja. U svakom koraku izvršavanja, proces uklanja signal sa čela ulaznog reda i poredi signal sa ulaznim konstrukcijama koje prate trenutno stanje procesa. Ako signal odgovara

jednom od ulaza datog stanja, proces izvršava sve konstrukcije koje slede nakon odgovarajućeg ulaza, sve dok izvršavanje ne dostigne sledeće stanje konstrukcije.

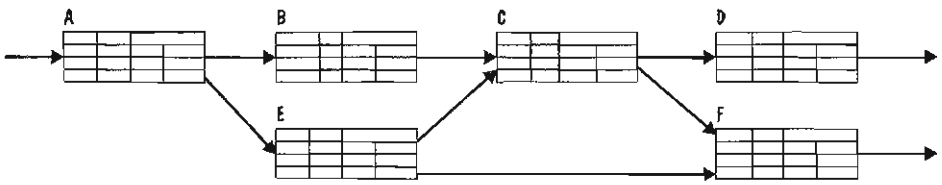
- **SDL tip podataka (algebarska specifikacija):** SDL proces može da deklariše lokalne promenljive, a SDL definicije tipova podataka se koriste za deklarisanje kompleksnih, korisnički definisanih tipova promenljivih.

Poređ toga, SDL specifikacija je često praćena skupom dijagrama sekvenci poruka (*Message Sequence Chart*, MSC) (ITU 1996), gde svaki dijagram ilustruje jedno izvršavanje specifikacije, zavisno od poruka koje se prosleđuju između procesa date specifikacije.

Smanjenje troškova softvera (SCR)

Smanjenje troškova softvera (***Software Cost Reduction***, **SCR**) (Heitmeyer 2002) predstavlja zbirku tehnika koje su projektovane radi podsticanja učesnika u razvoju softvera da primenjuju dobre principe projektovanja u okviru softverskog inženjerstva. SCR specifikacija modeluje softverske zahteve kao matematičku funkciju REQ, koja preslikava **nadgledane promenljive**, tj. promenljive okruženja na koje sistem *reaguje*, u **kontrolisane promenljive**, tj. promenljive okruženja čije vrednosti sistem *postavlja*. Funkcija REQ je raščlanjena na skup tabelarnih funkcija, slično Parnasovim tabelama. Svaka od tih funkcija je odgovorna za postavljanje vrednosti jedne kontrolisane promenljive ili **člana** (promenljiva slična makrou, koju druge definicije funkcija referenciraju).

REQ je posledica sklapanja tih tabelarnih funkcija u mrežu (DFD, kao što je prikazano na slici 4.22), čije ivice odražavaju međuzavisnosti funkcija od podataka. Svaki korak izvršavanja započinje promenom vrednosti jedne od nadgledanih promenljivih. Ta promena se zatim prostire kroz mrežu, u jednom, sinhronizovanom koraku. Funkcije specifikacije se primenjuju u topološkom redosledu koji se oslanja na funkcionalne zavisnosti podataka. Sve funkcije koje referenciraju ažurirane vrednosti promenljivih moraju da se izvrše *nakon* funkcija koje ažuriraju te vrednosti. Prema tome, korak izvršavanja liči na talas ažuriranja promenljivih koji plovi kroz mrežu i koji počinje sa prepoznatom novom vrednošću nadgledane promenljive, zatim sledi ažuriranje krajnjih promenljivih, a potom ažuriranje vrednosti kontrolnih promenljivih.



SLIKA 4.22 SCR specifikacija kao mreža tabelarnih funkcija

Ostale odlike notacija zahteva

Postoje i mnoge druge tehnike modelovanja zahteva. Neke tehnike uključuju mehanizme za pridruživanje stepena neizvesnosti ili rizika uz svaki zahtev. Postoje takođe tehnike koje poseduju mehanizme za praćenje zahteva u ostalim sistemskim dokumentima, kao što su projektna rešenja ili kôd, ili u okviru drugih sistema, kao što je slučaj kada se zahtevi ponovo koriste. Mnoge specifikacione tehnike su automatizovane do nekog stepena,

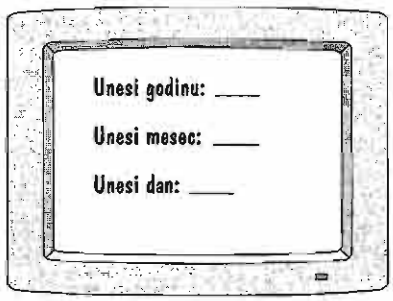
što olakšava crtanje dijagrama, sakupljanje termina i oznaka u rečniku podataka, kao i proveravanje očiglednih nedoslednosti. Kako se dalji razvoj alata nastavlja sa ciljem podrške aktivnostima softverskog inženjerstva, dokumentovanje i praćenje zahteva biće još lakše. Međutim, najteži deo analize zahteva, razumevanje potreba naručioca, moraju da obavljaju ljudi.

4.7 IZRADA PROTOTIPA ZAHTEVA

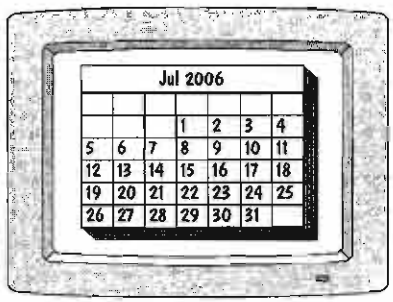
Prilikom pokušaja definisanja zahteva, možemo utvrditi da naručioci nisu sasvim sigurni šta tačno žele ili šta im je stvarno potrebno. Iskazivanje zahteva može da rezultuje „spiskom želja" sa veoma malo detalja ili bez jasne naznake da li je taj spisak potpun. Oprez! Ti isti naručioci, koji su neodlučni u svojim zahtevima, bez teškoća razlikuju isporučeni sistem koji zadovoljava njihove potrebe i sistem koji ih ne zadovoljava; to su naručioci za koje važi krilatica „znaću tek kada budem video" (Boehm 2000). U stvari, većina ljudi nalazi da je lakše detaljno kritikovati postojeći proizvod nego jednako detaljno zamisliti novi. Prema tome, jedan način kako možemo da saznamo detalje jeste da izradimo prototip predloženog sistema i da tražimo povratnu informaciju od potencijalnog korisnika o tome koje aspekte bi voleo poboljšati, koje odlike nisu dovoljno korisne, ili koja funkcionalnost nedostaje. Izrada prototipa može takođe da pomogne pri utvrđivanju izvodljivosti rešenja za naručiočev problem, kao i u traženju opcija za optimizaciju zahteva sa aspekta kvaliteta.

Da bismo videli kako funkcioniše izrada prototipa, pretpostavićemo da realizujemo alat koji prati koliko korisnik vežba svakog dana. Naši naručioci su psiholozi i predavači koji izvode vežbe, a njihovi klijenti su korisnici. Alat treba da pomogne psiholozima i predavačima prilikom rada sa klijentima i da prati kako napreduje obuka njihovih klijenata. Korisnički interfejs alata je važan, jer korisnicima možda nisu bliski računari. Na primer, prilikom unošenja informacija o njihovom načinu vežbanja, korisnik treba da unese datum primene. Predavači nisu sigurni kako korisnički interfejs treba da izgleda, tako da ćemo brzo izraditi prototip da bismo demonstrirali mogućnosti. Slika 4.23 prikazuje prvi prototip, u kojem korisnik mora da unese dan, mesec i godinu. Još zanimljiviji i sofisticiraniji interfejs uključuje kalendar (videti sliku 4.24), gde se korisnik služi mišem za izbor meseca i godine - sistem prikazuje grafikon za taj mesec, a korisnik bira odgovarajući dan u tabeli. Treća alternativa je prikazana na slici 4.25, u kojoj se, umesto kalendara, korisniku predstavljaju tri trake sa klizačem. Kada korisnik mišem pomera klizač levo ili desno, polje na dnu ekrana se menja i prikazuje odabrani dan, mesec i godinu. Taj treći interfejs obezbeđuje najbrže biranje, čak i ako se značajno razlikuje od onoga šta su korisnici navikli da vide. U ovom primeru, izrada prototipa pomaže da odaberemo pravi izgled i funkcije korisničkog interfejsa za interakciju korisnika sa predloženim sistemom. Interfejse prototipova bi bilo teško opisati rečima ili simbolima i oni demonstriraju kako je neke zahteve bolje predstaviti u vidu slika ili prototipova.

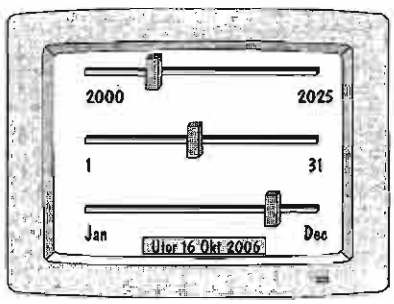
SLIKA 4.23 Prototip sa unosom preko tastature



SLIKA 4.24 Prototip sa kalendarom



SLIKA 4.25 Prototip sa klizačima



Kod izrade prototipa postoje dva pristupa: prototip koji se baca i evolutivni prototip. **Prototip koji se baca** predstavlja softver koji se razvija radi boljeg upoznavanja sa problemom ili predloženog rešenja i koji ne predstavlja deo softvera koji se isporučuje. Taj pristup nam omogućava da napišemo „brz i prljav" softver koji je loše strukturisan, neefikasan, bez provere grešaka, i koji verovatno predstavlja samo fasadu koja ne implementira ništa od željene funkcionalnosti, ali nas brzo dovodi do srži problema ili predloženog rešenja. Kada na naša pitanja dobijemo odgovarajuće odgovore, bacamo prototipski softver i započinjemo inženjerski posao na izradi softvera koji će biti isporučen. Nasuprot tome, evolutivni prototip predstavlja softver koji se razvija ne samo da nam pomogne da odgovorimo na pitanja, već da bi bio ugrađen u finalni proizvod. Zato, moramo da budemo mnogo pažljiviji pri njegovoj izradi, jer taj softver na kraju treba i da zadovolji zahteve u pogledu kvaliteta (na primer, brzinu odziva, modularnost) finalnog proizvoda, jer je kvalitet nemoguće naknadno ugrađivati u finalni proizvod.

Obe tehnike se ponekad nazivaju **brza izrada prototipa** jer uključuju izradu softvera da bi se dobio odgovor na pitanja vezana za *zahteve*. Termin „brza" pravi razliku izrade softverskih prototipova od sličnih postupaka u drugim inženjerskim disciplinama, u kojima prototip obično predstavlja kompletno rešenje, kao što je recimo, prototip automobila ili aviona, koji se grade ručno prema već odobrenim projektnim rešenjima. Namena takvog prototipa je testiranje projektnih rešenja i proizvoda pre automatizacije ili optimizacije ručnih koraka u masovnoj proizvodnji. Nasuprot tome, u slučaju softvera, brzi prototip je delimično rešenje koje se gradi kao pomoćno sredstvo za razumevanje zahteva ili za ocenjivanje alternativnih projektnih rešenja.

Pitanja vezana za zahteve mogu da se istraže ili putem modelovanja ili putem izrade prototipa. Da li je jedan pristup bolji od drugog, zavisi od od toga o kojim se pitanjima radi, kako dobro ona mogu biti izražena u modelima ili u softveru, kao i koliko brzo može da se izradi softverski model ili prototip. Kao što smo već videli, odgovore na pitanja o korisničkom interfejsu možda je lakše dati uz oslonac na prototipove. Dalje, prototip koji implementira određeni broj predloženih svojstava, još efikasnije pomaže korisnicima da odrede prioritete i verovatno identifikuju neka svojstva koja su nepotrebna. S druge strane, odgovore na pitanja o ograničenjima u pogledu redosleda dešavanja ili sinhronizacije aktivnosti, možemo mnogo brže dobiti oslanjajući se na modele. Na kraju, potrebno je izraditi finalnu dokumentaciju o zahtevima neophodnu timovima koji se bave testiranjem i održavanjem, a verovatno i kontrolnim telima, kao i finalni softver koji treba da se isporuči. Prema tome, da li je bolje da modelujemo ili pravimo prototip, zavisi od toga da li je brže i lakše da modelujemo i razvijemo softver iz pročišćenih modela, ili je brže i lakše da izradimo prototip i iz pročišćenog prototipa izvedemo dokumentaciju.

4.8 DOKUMENTOVANJE ZAHTEVA

Bez obzira na to koji metod za definisanje zahteva odaberemo, moramo da održavamo dokumenta u kojima su evidentirani rezultati. Zajedno sa naručiocima ćemo se pozivati na te dokumente tokom razvoja ili održavanja. Stoga zahtevi moraju da budu dokumentovani tako da budu korisni ne samo naručiocu, već i tehničkom osoblju u našem razvojnom timu. Na primer, zahtevi moraju da budu organizovani na takav način da mogu da se prate tokom razvoja sistema. Jasne i precizne ilustracije i dijagrami koji predstavljaju prateću dokumentaciju, treba da budu usaglašeni sa tekstom. Takođe je važan i nivo na kojem su napisani zahtevi, kao što je objašnjeno u podsetniku 1.6.

Definisanje zahteva

Definisanje zahteva predstavlja evidentiranje zahteva uz oslonac na terminologiju naručioca. Radeći sa naručiocem, dokumentujemo šta naručilac može da očekuje od isporučenog sistema:

1. Prvo skiciramo opštu namenu i opseg sistema, uključujući relevantne koristi i ciljeve. Veze sa drugim zavisnim sistemima se takođe uključuju, pri čemu navodimo terminologiju, označavanje i korisne skraćenice.

PODSETNIK 4.6 NIVO SPECIFIKACIJE

Godine 1995, Australijska organizacija za odbranu i tehnologiju (*Australian Defence and Technology Organisation*) objavila je rezultate istraživanja problema u vezi sa specifikacijom zahteva za softver za potrebe ratne mornarice (Gabb i Henderson 1995). Jedan od problema koji je tom prilikom istaknut bio je neujednačen nivo specifikacija. To jest, neki zahtevi su navedeni na suviše visokom nivou, dok su drugi bili suviše detaljni. Ta neujednačenost je obuhvatala nekoliko situacija:

- analitičari zahteva koristili su različite stilove pisanja, posebno prilikom dokumetovanja različitih oblasti sistema;
- razlika u iskustvu među analitičarima dovela je do različitih nivoa detalja zahteva;
- prilikom pokušaja preuzimanja zahteva iz prethodnih sistema, analitičari su koristili različite formate i stilove pisanja;
- zahtevi su bili često prespecifikovani, tako da je analitičar identifikovao određene tipove računara i programskih jezika, pretpostavljao pojedina rešenja ili propisivao neodgovarajuće procese i protokole. Analitičari ponekad pomešaju zahteve i delimična rešenja, dovodeći do „ozbiljnih problema pri projektovanju ekonomičnih rešenja";
- zahtevi su ponekad bili nedovoljno specifikovani, posebno prilikom omisivanja radnog okruženja, održavanja, simulacije za potrebe obuke, administrativne obrade i otpornosti na greške.

Većina ispitanika složila se da ne postoji univerzalno ispravan nivo izrade specifikacija. Iskusni naručioci radije daju specifikacije na visokom nivou apstrakcije, a oni sa manje iskustva češće ulaze u detalje. Ispitanici su dali više predloga, od kojih su neki:

- pisati svaki iskaz tako da sadrži samo jedan zahtev;
- izbegavati da jedan zahtev referencira drugi;
- objediniti slične zahteve.

2. Dalje, opisujemo poreklo i razloge koji stoje iza predloga novog sistema. Na primer, ako sistem treba da zameni neki postojeći pristup, objasnićemo zašto je postojeći pristup nezadovoljavajući. Aktuelni metodi i procedure se dovoljno detaljno skiciraju, tako da možemo da odvojimo elemente kojima je naručilac zadovoljan, od onih u koje se razočarao.
3. Nakon što zapišemo taj pregled problema, opisujemo suštinske karakteristike prihvatljivog rešenja. Taj zapis obuhvata kratke opise suštine funkcionalnosti proizvoda, na nivou slučajeva korišćenja. Zapisi takođe obuhvataju zahteve u pogledu kvaliteta, kao što je pitanje vremena, tačnosti i odgovora na otkaze. U idealnom slučaju dodelićemo prioritete zahtevima i identifikovati one koji mogu da se ostave za kasnije verzije sistema.
4. Kao deo konteksta problema, opisujemo okruženje u kojem će sistem da radi. Navodimo sve poznate hardverske i softverske komponente sa kojima predloženi sistem treba da bude u interakciji. Da bismo se uverili da je korisnički interfejs odgovarajući, pravimo nacrt opšteg radnog iskustva i sposobnosti korisnika siste-

ma, kao što su obrazovanje, iskustvo i tehnička stručnost. Na primer, možemo da osmislimo različite korisničke interfejse za iskusne korisnike i početnike. Pored toga, navodimo eventualna ograničenja vezana za zahteve ili projektna rešenja, kao što su propisi koji se primenjuju, hardverska ograničenja, revizijske provere, regulatorna politika itd.

5. Ako naručilac ima predlog rešenja problema, skiciramo opis predloga. Međutim, treba se podsetiti da je namena dokumenta sa izloženim zahtevima da se razmotri problem, a ne rešenje. Potrebno je da pažljivo ocenimo predloženo rešenje, da bismo odredili da li se radi o projektnom ograničenju koje treba da bude zadovoljeno ili o dodatnoj specifikaciji koja bi mogla da isključi bolja rešenja. Na kraju, ako naručilac postavi ograničenja u pogledu razvoja ili ako postoje neke posebne pretpostavke koje treba da se učine, potrebno ih je uključiti u definiciju zahteva.
6. Konačno, navodimo eventualne pretpostavke vezane za ponašanje okruženja. Posebno opisujemo eventualne uslove iz okruženja koji mogu da prouzrokuju otkaz predloženog sistema, kao i eventualne promene okruženja koje mogu da prouzrokuju izmene zahteva. Podsetnik 4.7 detaljnije objašnjava zašto je važno da dokumentujemo pretpostavke. Pretpostavke treba da budu dokumentovane odvojeno od zahteva tako da projektanti znaju za koje ponašanje su odgovorni prilikom implementacije.

Specifikacija zahteva

Specifikacija zahteva pokriva identičnu oblast kao i definisanje zahteva, ali iz perspektive učesnika u razvoju. Dok su definicije zahteva pisane rečnikom naručioca, pozivajući se na objekte, stanja, događaje i aktivnosti iz sveta naručioca, specifikacija zahteva se iskazuje u kategorijama interfejsa sistema. To postižemo preformulisanjem zahteve tako da referenciraju jedino one objekte iz realnog sveta (stanja, događaje i akcije) na koje predloženi sistem reaguje ili koje aktivira:

1. Prilikom dokumentovanja interfejsa sistema, detaljno opisujemo sve ulaze i izlaze, uključujući i izvore ulaza, odredišta izlaza, opsege vrednosti i formate podataka ulaza i izlaza, protokole podataka koji upravljaju redosledom u kojem određeni ulazi i izlazi moraju da se razmenjuju, formate i organizaciju prozora, kao i eventualna vremenska ograničenja. Treba da napomenemo da je korisnički interfejs retko i jedini interfejs sistema, budući da sistem može da vrši interakciju sa drugim softverskim komponentama (na primer, bazom podataka), hardverom posebne namene, internetom itd.
2. Zatim, ponovo iskazujemo zahtevanu funkcionalnost u kategorijama ulaza i izlaza interfejsa. Pri tome možemo da koristimo funkcionalnu notaciju ili dijagrame toka podataka za preslikavanje ulaza na izlaz, ili logiku za dokumentovanje prethodnih uslova i naknadnih uslova funkcija. Možemo da koristimo konačne automate ili tragove događaja za ilustrovanje sekvenci operacija ili redosleda ulaza i izlaza. Možemo da koristimo dijagram odnosa između entiteta u cilju okupljanja povezanih aktivnosti i operacija u klase. Na kraju, specifikacija treba da je potpuna, što znači da treba da specifikuje izlaz za sve izvodljive sekvence ulaza. Otuda uključujemo provere ispravnosti ulaza i odgovora sistema na izuzetne situacije, kao što je narušavanje preduslova.

PODSETNIK 4.7 SKRIVENE PRETPOSTAVKE

Zave i Jackson (1997) su pažljivo posmatrali probleme u softverskim zahtevima i specifikacijama, uključujući nedokumentovane pretpostavke o ponašanju realnog sveta.

U stvari, postoje dva tipa ponašanja okruženja koja nas zanimaju. To su željeno ponašanje koje treba da realizuje predloženi sistem (tj. zahtevi) i postojeće ponašanje koje predloženi sistem ne menja. Ova druga vrsta ponašanja se često naziva **pretpostavka** ili **domensko znanje**. Mnogi od onih koji evidentiraju zahteve smatraju pretpostavke kao jednostavne uslove pod kojim sistem garantovano radi ispravno. Iako su neophodni, ti uslovi nisu i jedine pretpostavke. Takođe pravimo i pretpostavke o tome kako će se okruženje ponašati kao odgovor na izlaze sistema.

Razmotrimo železničku rampu na ukrštanju puta i skupa železničkih koloseka. Naš zahtev je da se voz i automobili ne sudare na prelazu. Međutim, vozovi i automobili su izvan kontrole sistema. Sve što naš sistem može da uradi jeste da spusti rampu pre dolaska voza i podigne je nakon što voz prođe. Jedini način da rampa spreči sudare je da vozovi i automobili slede određena pravila. Na primer, treba da pretpostavimo da se vozovi kreću nekom maksimalnom brzinom i da je poznato koliko ranije treba da spustimo rampu da bismo osigurali da je ona spuštena dovoljno pre nego što voz, na koji se reagovalo, dođe do prelaza. Međutim, takođe treba da pretpostavimo kako će vozači automobila da reaguju na spuštenu rampu. Moramo da pretpostavimo da automobili neće da zastanu na pružnom prelazu ili da u njega udu dok je rampa spuštena.

3. Konačno, treba da utvrdimo kriterijume usklađenosti svakog zahteva u pogledu kvaliteta koje postavlja naručilac, tako da možemo da ubedljivo demonstriramo da li naš sistem zadovoljava te kriterijume kvaliteta.

Rezultat je dovoljno detaljan opis onoga šta razvojni tim treba da proizvede, u kome se jasno razlikuju prihvatljiva i neprihvatljiva rešenja, ali bez naznaka o načinu projektovanja i implementacije predloženog sistema.

Više organizacija, kao što su IEEE i Ministarstvo odbrane SAD, poseduju standarde koji definišu sadržaj i format dokumenata o zahtevima. Na primer, slika 4.26 prikazuje obrazac zasnovan na preporukama IEEE-a za organizovanje specifikacije softverskih zahteva po klasama ili objektima. IEEE standard sadrži slične obrasce za organizovanje zahteva prema režimu rada, funkcijama, odlikama, kategoriji korisnika itd. Prilikom pripremanja dokumenata za projekte, trebalo bi se držati tih standarda.

1. Uvod u dokument
   1.1 Namena proizvoda
   1.2 Opseg proizvoda
   1.3 Akronimi, skraćenice, definicije
   1.4 Reference
   1.5 Skica ostalog dela SRS-a
2. Opšti opis proizvoda
   2.1 Kontekst proizvoda
   2.2 Funkcije proizvoda
   2.3 Karakteristike korisnika
   2.4 Ograničenja
   2.5 Pretpostavke i zavisnosti
3. Specifični zahtevi
   3.1 Zahtevi spoljašnjih interfejsa
      3.1.1 Korisnički interfejsi
      3.1.2 Hardverski interfejsi
      3.1.3 Softverski interfejsi
      3.1.4 Komunikacioni interfejsi
   3.2 Funkcionalni zahtevi
      3.2.1 Klasa 1
      3.2.2 Klasa 2
      ...
   3.3 Zahtevi u pogledu performanse
   3.4 Projektna ograničenja
   3.5 Zahtevi u pogledu kvaliteta
   3.6 Ostali zahtevi
4. Dodaci

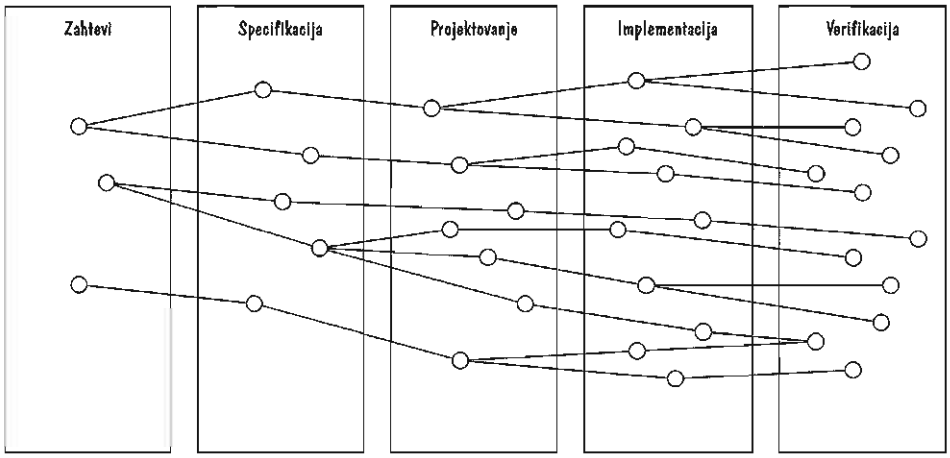
SLIKA 4.26 IEEE standardi za SRS , organizovani po objektima (IEEE 1998)

Upravljanje procesom i mogućnost praćenja zahteva

Neophodno je da postoji direktna korespondencija između zahteva u definicionom i zahteva u specifikacionom dokumentu. Ovo je mesto gde započinju metode upravljanja projektom koje će se koristiti tokom životnog ciklusa. Upravljanje procesom je skup procedura koje prate:

- zahteve koji definišu šta sistem treba da radi;
- projektne module koji se generišu na osnovu zahteva;
- programski kôd koji implementira projektna rešenja;
- testove koji verifikuju funkcionalnost sistema;
- dokumente koji opisuju sistem.

U određenom smislu, upravljanje procesom obezbeđuje niti koje povezuju delove sistema, integrišući dokumente i artefakte koji su nezavisno razvijeni. Te niti nam omogućavaju da koordinišemo razvojne aktivnosti, kao što je prikazano vodoravnim „nitima" između entiteta na slici 4.27. Posebno smo, tokom aktivnosti definisanja zahteva, treba voditi računa o uspostavljanju podudarnosti između elemenata iz definicije zahteva i specifikacije zahteva, tako da pogled naručioca bude povezan sa pogledom razvojnog tima, i to na tako organizovan način da ga je moguće direktno pratiti. Ako ne definišemo te veze, nemamo načina da projektujemo test-slučajeve koji potvrđuju da li kôd zadovoljava postavljene zahteve. U narednim poglavljima, videćemo da upravljanje procesom takođe omogućava da odredimo uticaj izmena, kao i da kontrolišemo efekte paralelnog razvoja.



SLIKA 4.27 Veze između entiteta softverskog razvoja

Da bismo olakšali podudaranje između definicije i specifikacije, uspostavljamo šemu označavanja ili datoteku zapisa za pogodno praćenje zahteva iz dokumenta u dokument. Često, tim za upravljanje procesima postavlja ili proširuje tu šemu označavanja radi povezivanja zahteva sa drugim komponentama i artefaktima sistema. Označavanje softverskih zahteva omogućava nam da uspostavimo unakrsne veze pomoću rečnika podataka i drugih pomoćnih dokumenata. Ako se neka izmena napravi u zahtevima tokom ostalih faza razvoja, izmene mogu da se prate od dokumenta sa zahtevima, preko procesa projektovanja, i celim putem do procedura za testiranje. U idealnom slučaju, posle toga možemo da pratimo sve odlike ili funkcije sistema do uzročnog zahteva i obrnuto.

4.9 VALIDACIJA I VERIFIKACIJA

Podsetimo se da dokumenti o zahtevima služe i kao ugovor između nas i naručioca, jer detaljno opisuju šta treba da isporučimo, ali i kao smernice za projektante, jer detaljno opisuju šta treba da bude napravljeno. Prema tome, pre prosleđivanja zahteva projektantima, zajedno sa naručiocem moramo da budemo apsolutno sigurni da svaka zainteresovana strana zna namere one druge, kao i da su naše namere evidentirane u dokumentima o zahtevima. Za uspostavljanje te sigurnosti, moramo da izvršimo *validaciju* zahteva i *verifikaciju* specifikacije.

U ovom poglavlju koristili smo izraze „verifikacija" i „validacija", bez formalnog definisanja njihovog značenja. Tokom **validacije zahteva**, proveravamo da li definicije zahteva tačno odražavaju definicije zahteva naručioca, odnosno, sve što je potrebno zainteresovanim subjektima. Validacija je prilično složena jer postoji samo nekoliko dokumenata koje možemo da koristimo kao osnovu za argumentaciju da su definicije zahteva ispravne. Kod verifikacije, proveravamo da li je jedan dokument ili artefakt saglasan sa drugim. Prema tome, verifikujemo da li je naš kôd saglasan sa projektnim rešenjem, kao i da li je projektno rešenje usaglašeno sa specifikacijom zahteva. Na nivou zahteva, verifikujemo da li specifikacija zahteva zadovoljava definiciju zahteva. Da rezimiramo, verifikacija osigurava da gradimo sistem *na pravi način*, dok validacija osigurava da gradimo *pravi* sistem!

Validacija zahteva

Naš kriterijum za validaciju zahteva jesu karakteristike koje smo naveli u odeljku 4.4:

- ispravan;
- dosledan;
- nedvosmislen;
- potpun;
- relevantan;
- može se testirati;
- može se pratiti.

Zavisno od tehnika definisanja koje koristimo, neke od navedenih provera (na primer, da li su zahtevi dosledni i da li se mogu pratiti) mogu da se automatizuju. Takođe, moguće je evidentirati uobičajene greške u **listi za proveru**, koju revizori mogu da koriste kao smernice prilikom traženja grešaka. NASA je u okviru Jet Propulsion Laboratory uspešno upotrebljavala liste za proveru prilikom validacije zahteva, o čemu izveštava Lutz (1993). Međutim, veći deo validacionih provera (na primer, da li su zahtevi ispravni, relevantni ili nedvosmisleni, ili da li su zahtevi potpuni) predstavljaju subjektivne provere po tome što one uključuju poređenje definicija zahteva sa zahtevima zainteresovanih subjekata, tj. mentalnim modelom onoga šta oni očekuju da sistem radi. Za te validacione provere, naše jedino uporište je oslanjanje na ocenjivanje dokumenata od strane zainteresovanih subjekata.

Tabela 4.3 navodi neke od tehnika koje mogu da se koriste za validaciju zahteva. Validacija može u najjednostavnijem slučaju da se svede na čitanje dokumenata i izveštavanje o greškama.

TABELA 4.3 Tehnike validacije i verifikacije

Validacija	Prolasci
	Čitanje
	Intervju
	Pregledi
	Liste za proveru
	Model za proveru funkcija i odnosa
	Scenariji
	Prototipovi
	Simulacija
	Formalne inspekcije
Verifikacija	Unakrsno referenciranje
	Simulacija
	Provera doslednosti
	Provera potpunosti
	Provera nedostižnih stanja ili prelaza
Provera	Provera modela
	Matematičko dokazivanje

Možemo od validacionog tima da tražimo da parafira dokument, deklarišući time da je izvršio reviziju dokumenta i da ga odobrava. Nakon parafiranja, zainteresovani subjekti prihvataju delimičnu odgovornost za greške koje se nakon toga nađu u doku-

mentu. Alternativno, možemo da obavimo **prolazak kroz dokumente**, gde jedan od autora dokumenta predstavlja zahteve ostalim zainteresovanim subjektima i od njih traži komentare. Prolaženje kroz dokumente je najefikasnije ako postoji veliki broj raznih zainteresovanih subjekata, a nije realno da od njih zahtevamo da detaljno analiziraju dokumente. Suprotan ekstremni slučaj je validacija, koja može da bude strukturisana kao **formalna inspekcija**, gde revizori uzimaju određene uloge (na primer, izlagač, posrednik) i prate propisana pravila (tj. pravila o tome kako da se ispituju zahtevi, kada se održavaju sastanci i prave pauze, kada da se nastavi inspekcija).

Mnogo češće se validacija zahteva vrši pomoću **ocenjivanja** zahteva. Prilikom ocenjivanja, predstavnici našeg osoblja i osoblja naručioca pojedinačno ispituju dokumente o zahtevima, a zatim se sastaju da bi raspravljali o uočenim problemima. Predstavnici naručioca uključuju one koji će raditi na sistemu, one koji će pripremati ulaz sistema, kao i one koji će koristiti izlaze sistema, pri čemu učešće mogu da uzmu i rukovodioci tih radnika. Mi obezbeđujemo članove razvojnog tima, tima za testiranje i operativnog tima. Radom u grupi, možemo da uradimo više od provere da definicija softverskih zahteva zadovoljava validacioni kriterijum:

1. Pregledamo navedene ciljeve i namenu sistema.
2. Poredimo zahteve sa ciljevima i namenom, da bismo se uverili da su svi zahtevi neophodni.
3. Ocenjujemo okruženje u kojem sistem treba da radi, ispitujući interfejse između predloženog sistema i svih drugih sistema i proveravamo da li su njihovi opisi kompletni i ispravni.
4. Predstavnici naručioca pregledaju tok informacija i predložene funkcije, da bi potvrdili da one tačno odražavaju potrebe i namere naručioca. Naši predstavnici ocenjuju predložene funkcije i ograničenja, da bi potvrdili da su oni realistični i u okviru naših razvojnih mogućnosti. Svi zahtevi se ponovo proveravaju na eventualne propuste, nepotpunost ili nedoslednosti.
5. Ako postoji rizik uključen u razvoj ili stvarno funkcionisanje sistema, možemo da procenimo i dokumentujemo rizike, diskutujemo i poredimo alternative, kao i da postignemo saglasnost o pristupima koji treba da se koriste.
6. Ako sistem treba da se isporuči u fazama, možemo da razgovaramo o testiranju sistema: kako će se vršiti ponovna validacija zahteva sa njihovom evolucijom i promenama; ko treba da obezbedi podatke za testiranje timu testira; koji zahtevi će biti testirani u kojim fazama.

Uvek kada se uoči problem, on se dokumentuje, određuje se uzrok problema, a analitičari zahteva dobijaju zadatak da reše problem. Na primer, validacija može da otkrije da postoji veliko nerazumevanje vezano za način na koji određena funkcija generiše rezultat. Naručioci mogu da zahtevaju da se podaci izražavaju u miljama, dok korisnici žele podatke u kilometrima. Naručilac može da kao cilj postavi pouzdanost ili raspoloživost, za koje projektni tim smatra da ih je nemoguće postići. Ti konflikti treba da budu razrešeni pre nego što počne projektovanje. Da bismo razrešili konflikt, razvojni tim možda ima

potrebu da napravi simulaciju ili prototip, u cilju istraživanja ograničenja izvodljivosti, a nakon toga da radi sa naručiocem na postizanju dogovora o prihvatljivim zahtevima. Podsetnik 4.8 razmatra prirodu i broj problema u vezi sa zahtevima, sa kojima ćemo se verovatno susresti.

Izbor validacione tehnike zavisi od iskustva i želja zainteresovanih subjekata, kao i od usklađenosti tehnika sa notacijama korišćenim u sklopu definicija zahteva. Neke notacije imaju pomoćne alate za proveravanje doslednosti i potpunosti. Takođe postoje alati koji potpomažu proces pregleda, praćenje problema i nalaženje rešenja. Na primer, neke alate koristimo zajedno sa naručiocem da bismo smanjili stepen neizvesnosti u zahtevima. Prateća veb stranica uz ovu knjigu pokazuje alate koji se odnose na zahteve.

PODSETNIK 4.8 BROJ NEDOSTATAKA U ZAHTEVIMA

Koliko razvojnih problema nastaje tokom prikupljanja zahteva? O tome postoje različita mišljenja. Boehm i Papaccio (1988) u članku koji analizira softver u IBM-u i TRW-u, nalaze da većina grešaka nastane tokom projektovanja, pri čemu obično postoje tri projektna nedostatka na svaka dva nedostatka u kodiranju. Oni ukazuju da veliki broj nedostataka inicijalno pripisanih fazi projektovanja može da se izvede iz grešaka vezanih za zahteve. U svojoj knjizi o ekonomici softverskog inženjerstva, Boehm (1981) citira studije Jonesa i Thayera i drugih koji dodeljuju:

- 35% nedostataka projektnim aktivnostima kod projekata od 30 000 do 35 000 isporučenih izvornih instrukcija;
- 10% nedostataka aktivnostima definisanja zahteva i 55% nedostataka projektnim aktivnostima za projekte od 40 000 do 80 000 isporučenih izvornih instrukcija;
- 8% do 10% nedostataka aktivnostima definisanja zahteva i od 40% do 55% nedostataka aktivnostima projektovanja za projekte od 65 000 do 85 000 isporučenih izvornih instrukcija.

Basili i Perricone (1984) u empirijskom istraživanju softverskih grešaka, izveštavaju da je 48% nedostataka otkrivenih u softverskim projektima srednje veličine „pripisano pogrešnim ili loše protumačenim funkcionalnim specifikacijama ili zahtevima".

Belzer (1990) je 8,12% nedostatka iz analiziranih uzoraka pripisao problemima u funkcionalnim zahtevima. On je svojom analizom obuhvatio probleme kao što su: neispravni zahtevi, nelogični i neumereni zahtevi; dvosmisleni, nepotpuni ili prespecifikovani zahtevi; zahtevi koji ne mogu da se verifikuju ili testiraju; loše predstavljeni zahtevi; izmenjeni zahtevi. Međutim, Belzer analizom nije obuhvatio aktivnosti projektovanja. On kaže: „Zahtevi, posebno onako kako su izraženi u specifikaciji (ili često onako kako nisu izraženi specifikacijom, jer nije ni nema) jesu glavni izvor skupih programskih grešaka. Procentualno, ove greške su u rasponu od nekoliko procenata, pa do više od 50%, zavisno od oblasti primene i okruženja. Ono što najviše boli u vezi sa tim greškama je to što one najranije osvajaju sistem i poslednje ga napuštaju. Nije neobično da zahtev sa nedostatkom prođe kroz svo razvojno testiranje, beta testiranje i početnu upotrebu na lokaciji naručioca, i da se otkrije tek nakon instalacije na stotinama lokacija."

Postoji obilje drugih zbirnih statistika. Na primer, Perry i Stieg (1993) zaključuju da su 79,6% nedostataka interfejsa i 20,4% implementacionih nedostataka posledica nekompletnih zahteva ili propusta u zahtevima. Slično tome, *Computer Weekly Report* (1994) uradio je analizu koja je pokazala da se 44,1% svih sistemskih nedostataka događa u fazi specifikacije. Lutz (1993b) analizira greške koje se odnose na bezbednost u dva softverska sistema svemirskih brodova organizacije NASA, pri čemu nalaze da „primarni uzrok nedostataka interfejsa u vezi sa bezbednošću jeste pogrešno razumevanje specifikacija hardverskih interfejsa" (od 48% do 67% takvih rezultata), dok „primarni uzrok funkcionalnih nedostataka u vezi sa bezbednošću jesu greške u prepoznavanju (razumevanju) zahteva" (62% do 79% takvih nedostataka). Šta je pravi procenat u slučaju vašeg razvojnog okruženja? Jedino pažljivo vođenje evidencije može da nam na to da odgovor. Ta evidencija može da se koristi kao osnova za merenje napretka prilikom uvođenja novog načina rada ili korišćenja novih alata.

Verifikacija

Kod verifikacije, želimo da proverimo da li naš dokument sa specifikacijom zahteva odgovara dokumentu definicije zahteva. Ta verifikacija treba da nas uveri da će, ako implementiramo sistem koji zadovoljava specifikaciju, on da zadovoljava i zahteve naručioca. Još češće, to je jednostavno provera mogućnosti praćenja, gde obezbeđujemo da svaki zahtev u dokumentu sa definicijama može biti praćen sve do specifikacije.

Međutim, kod kritičnih sistema želimo da uradimo i više, i da zaista demonstriramo da specifikacija ispunjava zahteve. Tada dokazujemo da specifikacija realizuje sve funkcije, događaje, aktivnosti i ograničenja data u zahtevima. Sama specifikacija je retko dovoljna da pruži tu vrstu argumentacije, jer se ona piše u smislu akcija koje će se izvesti u sistemskom interfejsu, recimo, sila koja se primenjuje na otključanu obrtnu rampu; a mi hoćemo da dokažemo nešto o okruženju izvan tog interfejsa, kao što je, recimo, broj ulazaka u zoološki vrt. Da bismo premostili taj jaz, potrebno je da iskoristimo naše pretpostavke o tome kako se okruženje ponaša, tj. pretpostavke o tome koje ulaze će sistem da prihvata ili kako će okruženje da reaguje na izlaze. Na primer, ako je otključana obrtna rampa gurnuta dovoljnom silom, zarotiraće se za jedan poluokret, propuštajući posetioca u zoološki vrt. Matematički, specifikacija (S) plus naše pretpostavke (P) o okruženju, moraju da budu dovoljni za dokazivanje da zahtevi (Z) važe:

$$S, P \vdash Z$$

Na primer, da bismo prikazali da termostat i peć kontrolišu temperaturu vazduha, potrebno je da pretpostavimo da se temperatura vazduha menja kontinualno a ne skokovito, mada senzori mogu da detektuju diskretne promene vrednosti, a peć koja je u radu podiže temperaturu vazduha. Te pretpostavke mogu da budu očigledne, ali ako je zgrada dovoljno porozna a spoljna temperatura dovoljno niska, onda naša druga pretpostavka neće važiti. U takvom slučaju, radi opreza bi trebalo da postavimo neke granice zahteva: sve dok je spoljna temperatura iznad -100°C, termostat i peć održavaće temperaturu vazduha.

Ovakva upotreba pretpostavki o okruženju u suštini pokazuje zašto je dokumentacija tako važna: oslanjamo se na okruženje da nam ono pomogne da zadovoljimo zahteve naručioca, pri čemu ako su naše pretpostavke o tome kako se okruženje ponaša pogrešne, onda sistem možda neće raditi prema očekivanju naručioca. Ako ne možemo da dokažemo da specifikacija i naše pretpostavke ispunjavaju zahteve naručioca, onda je potrebno ili da izmenimo specifikaciju, proverimo i dopunimo naše pretpostavke o okruženju ili da smanjimo zahteve koje pokušavamo da zadovoljimo. Podsetnik 4.9 razmatra neke tehnike za automatizovanje tih dokaza.

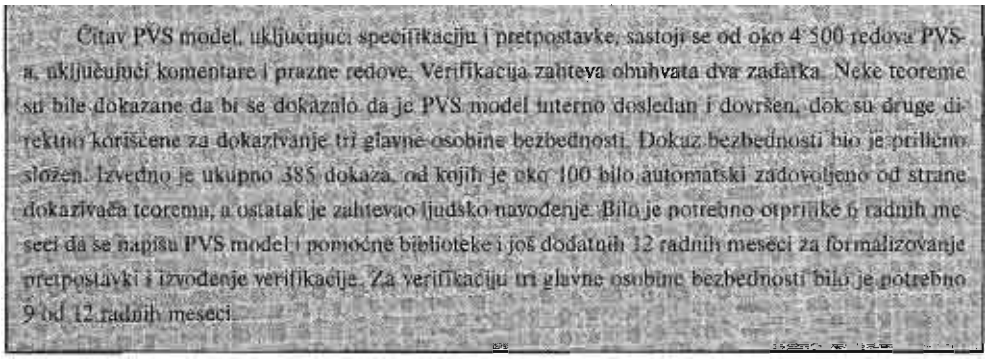
PODSETNIK 4.9 RAČUNARSKI PODRŽANA VERIFIKACIJA

Proveravanje modela je iscrpno pretraživanje prostora izvršavanja specifikacije, radi određivanja da li neka osobina temporalne logike važi prilikom izvršavanja. Program za proveravanje modela proračunava i pretražuje prostor izvršavanja specifikacije, ponekad simbolički obrađujući prostor izvršavanja, a ponekad ga obrađujući u hodu, tokom pretraživanja. Prema tome, i ako je potpuno automatizovana, verifikacija troši značajne računarske resurse. Sreemani i Atlee (1996) koriste za proveru model SMV, za verifikovanje pet osobina SCR specifikacije mornaričke letilice A-7. Njihov SMV model sastojao se od 1251 reda, od kojih je većina bila automatski preneseni iz SCR specifikacije, što je teoretski davalo prostor izvršavanja od $1.3 * 1022$ mogućih stanja. U njihovom proveravanju modeka, istraživači su utvrdili da se za jednu od osobina za koju se mislilo da ne važi ispostavilo suprotno, jer uslovi pod kojima ne bi važila jednostavno nisu bili dostižni. Oni su takođe otkrili da osobina bezbednosti nije važila. Prema specifikaciji, bilo je moguće da sistem za dopremanje oružja koristi zastarele podatke prilikom praćenja pozicije cilja, dok su navigacioni senzori zaključili da se očitavaju nerazumne vrednosti.

Program **dokazivač teorema** koristi kolekciju ugrađenih teorija, pravila zaključivanja i postupke odlučivanja da bi se odredilo da li skup dokazanih činjenica logički iziskuje neke činjenice koje se nisu dokazane. Mnogi sofisticirani dokazivači teorema zahtevaju ljudsku asistenciju prilikom skiciranja strategije dokazivanja. Dutertre i Stavridou (1997) koristili su dokazivač teorema PVS za verifikovanje nekih funkcionalnih zahteva i zahteva u pogledu bezbednosti avionskog sistema. Na primer, pretpostavka koja dovodi u vezu napadni ugao krila WSPOS u trenutku $t + eps$ i napadnu komandu krila CMD u trenutku t, u slučaju da nijedna blokada nije aktivna, u PVS-u se izražava na sledeći način:

```
no_wingstall: AXIOM
    CMD(t) < WSPOS(t + epsilon) AND
    WSPOS(t + eps) < WSPOS(t) + eps * ws_max_rate
OR
    CMD(t) > WSPOS(t + eps) AND
    WSPOS(t + eps) > WSPOS(t) - eps * ws_min_rate
```

РЕПУБЛИКА СРПСКА
Природно-математички
факултет Бања Лука


Čitav PVS model, uključujući specifikaciju i pretpostavke, sastoji se od oko 4 500 redova PVS-a, uključujući komentare i prazne redove. Verifikacija zahteva obuhvata dva zadatka. Neke teoreme su bile dokazane da bi se dokazalo da je PVS model interno dosledan i dovršen, dok su druge direktno korišćene za dokazivanje tri glavne osobine bezbednosti. Dokaz bezbednosti bio je prilično složen. Izvedno je ukupno 385 dokaza, od kojih je oko 100 bilo automatski zadovoljeno od strane dokazivača teorema, a ostatak je zahtevao ljudsko navođenje. Bilo je potrebno otprilike 6 radnih meseci da se napišu PVS model i pomoćne biblioteke i još dodatnih 12 radnih meseci za formalizovanje pretpostavki i izvođenje verifikacije. Za verifikaciju tri glavne osobine bezbednosti bilo je potrebno 9 od 12 radnih meseci.

Kada su završeni validacija i verifikacija zahteva, softversi inženjer i naručilac bi trebalo da budu prilično zadovoljni specifikacijom zahteva. Uz razumevanje šta naručilac želi, softverski inženjer može da nastavi sa projektovanjem sistema. U međuvremenu, naručilac ima u rukama dokument koji tačno opisuje šta isporučeni sistem treba da radi.

4.10 MERENJE ZAHTEVA

Postoje mnogi načini za merenje karakteristika zahteva, tako da prikupljene informacije dosta govore o procesu definisanja zahteva i kvalitetu samih zahteva. Merenja se obično usredsređuju na tri oblasti: proizvod, proces i resurse (Fenton i Pfleeger 1997). Broj zahteva u specifikaciji i definiciji zahteva može da nam pruži osećaj verovatne veličine sistema koji se razvija. Videli smo u poglavlju 3 da je za modele za izračunavanje uloženog rada neophodno proceniti veličinu proizvoda, i tada se veličina zahteva može koristi kao ulaz za takav model. Pored toga, veličina zahteva i procena napora mogu da se prate kroz proces razvoja. Kako projektovanje i razvoj vode do dubljeg razumevanja problema i rešenja, lako mogu iskrsavati novi zahtevi koji nisu bili očigledni u samom početku, kada je pravljena specifikacija zahteva.

Slično tome, možemo da merimo broj izmena zahteva. Veliki broj izmena ukazuje na neku nestabilnost, ili na naše nedovoljno razumevanje šta bi sistem trebalo da radi ili kako bi sistem trebalo da se ponaša, a takođe i sugeriše da treba da preduzmemo radnje u cilju smanjenja stope izmena. Praćenje izmena nastavlja se i tokom razvoja, a sa promenom sistemskih zahteva, može se ocenjivati uticaj izmena.

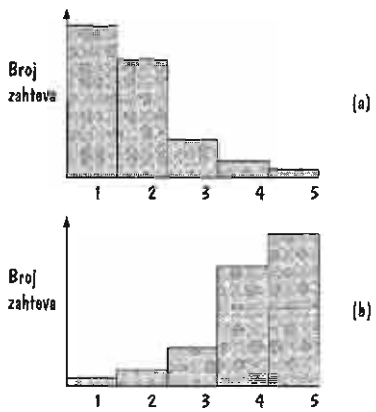
Gde je to moguće, veličina zahteva i merenje izmena mogu da se evidentiraju prema tipu zahteva. Takva metrika je zasnovana na kategoriji i kazuje nam da li se izmena ili neizvesnost u zahtevima događa na nivou celog sistema, ili se zadržava samo na određenim vrstama zahteva, kao što su, na primer, korisnički interfejs ili zahtevi u pogledu baze podataka. Ta informacija nam pomaže da odredimo da li je potrebno da usredsredimo našu pažnju na određene tipove zahteva.

Zahtevi služe projektantima i testerima, pa merenja treba da odraze njihovu ocenu zahteva. Korisno je, na primer, tražiti od projektanata da ocene svaki zahtev na skali od 1 do 5:

1. Vi (projektant) razumete u potpunosti ovaj zahtev, projektovali ste u prošlosti na osnovu sličnih zahteva i nećete imati problema da iz tog zahteva izvedete projektno rešenje.
2. Postoje elementi zahteva koji su novi za vas, ali nisu radikalno različiti od zahteva koje ste uspešno projektovali u prošlosti.
3. Postoje elementi zahteva koji su veoma različiti od zahteva koje ste projektovali u prošlosti, ali vama je zahtev razumljiv i mislite da možete da pomoću njega napravite dobro projektno rešenje.
4. Postoje delovi zahteva koje ne razumete, a vi niste sigurni da možete da napravite dobro projektno rešenje.
5. Uopšte ne razumete zahtev i ne možete da napravite projektno rešenje za njega.

Možete da napravite sličnu šemu rangiranja sa pitanjima testerskom timu, da bi se videlo koliko dobro oni razumeju svaki zahtev i koliko su uvereni da su sposobni da smisle pogodne testove za svaki zahtev. U oba slučaja, profili rangiranja mogu da služe kao grubi indikatori da li su zahtevi napisani na odgovarajućem nivou detalja. Ako projektni tim i tim za testiranje uglavnom dobijaju profile ocenjene sa 1 ili 2, kao što je prikazano na slici 4.28(a), onda su zahtevi dobro oblikovani i mogu da se proslede projektnom timu. Međutim, ako ima mnogo ocena 4 i 5, kao što je prikazano na slici 4.28(b), onda mora da se uradi revizija zahteva, a te revizije moraju ponovo da se ocene, da bi se dobili bolji profili, pre nego što se nastavi sa projektovanjem. Iako je ocenjivanje subjektivno, generalni trendovi moraju da budu jasni, a ocene moraju da obezbede korisne povratne informacije za nas same, kao za i naručioca.

Takođe, za svaki zahtev je dobro napisati napomenu o tome kada je izvršeno ocenjivanje, kada je ono implementirano kao projektno rešenje, kada je implementirano kao programski kôd i kada je zahtev testiran. Sve to govori o napretku, odnosno koliko smo blizu dovršenju projekta. Tim za testiranje takođe može da meri temeljnost svojih test slučajeva s obzirom na zahteve, kao što ćemo videti u poglavljima 8 i 9. Možemo da merimo broj zahteva pokrivenih svakim od test slučajeva, kao i broj zahteva koji su testirani (Wilson 1995).



SLIKA 4.28 Merenje gotovosti zahteva

4.11 IZBOR TEHNIKE ZA SPECIFIKACIJU

U ovom poglavlju predstavljeni su primeri nekoliko tehnika za specifikovanje zahteva, mada pored njih postoji još mnogo tehnika koje se mogu koristiti prilikom projektovanja. Svaka tehnika ima svoje dobre osobine, od kojih neke više a neke manje odgovaraju određenom projektu. Prema tome, važno je da za svaki projekat imamo skup kriterijuma na osnovu kojih odlučujemo koja tehnika najbolje odgovara.

Razmotrićemo sada neka od pitanja koja treba da budu uključena u takav skup kriterijuma. Pretpostavimo da gradimo računarski upravljan sistem za izbegavanje sudara između letilica. Letilice treba da bude opremljene radarskim senzorima. Na svakoj treba da postoji podsistem koji će pratiti druge letilice u blizini, detektovati kada se neka opasno približi i uspostaviti komunikaciju sa tom letelicom, pregovarati s njom o manevrima za izbegavanje sudara (ne sme se dogoditi da letilice nezavisno jedna od druge biraju manevre koji će ih skrenuti sa kursa sudara, ili ih zadržati na tom kursu), i koji će na kraju dati instrukcije svom navigacionom sistemu da izvrši dogovorene manevre. Podsistem sveke letilice izvodi svoju analizu podataka i procedure donošenja odluka na letačkim računarima, ali zajednički pristupa planovima letenja sa ostalim avionima i prenosi sve podatke i finalne manevre na centralnu lokaciju, radi dalje analize. Jedna od ključnih karakteristika sistema za izbegavanja sudara jeste da je to distribuiran i reaktivni sistem. To jest, to je **reaktivni sistem** zato što podsistem svakog aviona neprestano nadgleda i reaguje na položaj druge letelice. To je **distribuirani sistem** zato što su funkcije sistema distribuirane između više letelica. Zbog kompleksnosti ovóg sistema izuzetno je važno da zahtevi budu specifikovani precizno i u potpunosti. Interfejsi moraju da budu dobro definisani, a komunikacije moraju da budu koordinisane, tako da svaki podsistem može da pravovremeno donosi odluke. Nisu sve specifikacione tehnike jednako podesne za sve tipove problema. Na primer, testiranje opisanog podsistema biće teško jer, iz bezbednosnih razloga, većina testova neće moći da se obavi u realnom okruženju. Pored toga, biće teško da se detektuju i ponove tranzijentne greške. Prema tome, možda ćemo dati prednost tehnici koja omogućuje simulaciju, ili koja olakšava iscrpnu ili automatsku verifikaciju specifikacije. Tehnike koje automatski proveravaju specifikaciju ili sistem u pogledu doslednosti i potpunosti, posebno su podesne da uhvate greške na koje nije lako ukazati na neki drugi način.

Još opštije, ako sistem postavlja zahteve u pogledu rada u realnom vremenu, potrebna nam je specifikaciona tehnika koja podržava pojam vremena. Eventualna potreba za faznim razvojem znači da ćemo pratiti zahteve kroz nekoliko međusistema, koji ne samo da komplikuju praćenje zahteva, već i povećavaju verovatnoću da će zahtevi da se promene tokom života sistema. Kada korisnici budu radili sa međuverzijama sistema, uočiće možda da je potrebno uvesti nove odlike ili izmeniti postojeće. Prema tome, potreban nam je sofisticirani metod koji lako može da prihvata izmene. Ako želimo da zahtevi imaju sve željene karakteristike, navedene ranije u ovom poglavlju, onda tragamo za metodom koji treba da nam pomogne da revidiramo zahteve, pratimo izmene, unakrsno povezujemo podatke i funkcionalne stavke i analiziramo zahteve u odnosu na što je moguće više karakteristika.

Ardis je sa svojim kolegama (Ardis 1996) predložio skup kriterijuma za ocenjivanje specifikacionih metoda. Oni za svaki kriterijum formiraju spisak pitanja koja treba da nam pomognu da odredimo koliko dobro određeni metod zadovoljava taj kriterijum. Nji-

hovi kriterijumi su bili namenjeni tehnikama ocenjivanja određenih reaktivnih sistema, mada su kao što ćemo videti, mnogi od njih prilično opšti:

- **Primenljivost:** Može li tehnika da opiše probleme i rešenja iz realnog sveta na prirodan i realističan način? Ako tehnika pravi pretpostavke o okruženju, da li su te pretpostavke razumne? Da li je tehnika kompatibilna sa drugim tehnikama koje će se koristiti u okviru projekta?
- **Mogućnost implementacije:** Može li specifikacija da se lako pročisti ili prevede u implementaciju? Koliko je teško to prevođenje? Da li je ono automatizovano? Ako je tako, da li je generisani kôd efikasan? Da li je generisani kôd na istom jeziku kao onaj koji se koristi u ručno proizvedenim delovima implementacije? Da li postoji čist, dobro definisan interfejs između koda koga je generisala mašina i ostatka koda?
- **Mogućnost testiranja/simulacije:** Može li specifikacija da se koristi za testiranje implementacije? Da li se svaka stavka u specifikaciji može testirati kroz implementaciju? Da li je moguće izvršiti specifikaciju?
- **Mogućnost provere:** Da li je specifikacija čitljiva za one koji nisu projektanti, kao što je recimo naručilac? Mogu li stručnjaci iz date oblasti (tj. stručnjaci za problem za koji se pravi specifikacija) da provere tačnost specifikacije? Da li postoje programi za automatsku proveru specifikacije?
- **Mogućnost održavanja:** Da li specifikacija može da bude korisna prilikom pravljenja izmena sistema? Da li se specifikacija lako menja tokom evolucije sistema?
- **Modularnost:** Da li metod dozvoljava dekompoziciju velike specifikacije na manje delove koje je lakše napisati i razumeti? Mogu li izmene da se načine na manjim delovima bez ponovnog pisanja celokupne specifikacije?
- **Nivo apstrakcije/izražajnosti:** Koliko blisko i izražajno objekti, stanja i događaji u jeziku odgovaraju stvarnim objektima, akcijama i uslovima u domenu problema? Koliko je koncizna i elegantna rezultujuća specifikacija?
- **Ispravnost:** Da li jezici ili drugi alati olakšavaju proveravanje nedoslednosti ili višeznačnosti u specifikaciji? Da li je semantika specifikacionog jezika precizno definisana?
- **Mogućnost verifikacije:** Možemo li formalno da demonstriramo da specifikacija zadovoljava zahteve? Može li verifikacioni proces da bude automatizovan, i ako je tako, da li je ta automatizacija jednostavna?
- **Bezbednost izvršavanja:** Ako kôd može da se automatski generiše iz specifikacije, da li se kôd bez posledica degradira pod neočekivanim uslovima izvršavanja, kao što je prekoračenje opsega?
- **Zrelost alata:** Ako tehnika specifikovanja ima pomoćne alate, da li su ti alati visokog kvaliteta? Da li je na raspolaganju i obuka za upotrebu tih alata? Koliko je veliko predznanje korisnika za upotrebu tih alata?
- **Proizvoljnost:** Da li specifikacija može da bude nekompletna ili da dopušta neodređenost?
- **Krivulja učenja:** Može li novi korisnik da nauči koncepte, sintaksu, semantiku i heuristike date tehnike?
- **Zrelost tehnike:** Da li je tehnika sertifikovana ili standardizovana? Da li postoji korisnička grupa ili velika baza korisnika?

- **Modelovanje podataka:** Da li tehnika uključuje predstavljanje podataka, odnosa ili apstrakcija? Da li su mehanizmi modelovanja podataka sastavni deo tehnike?
- **Disciplina:** Da li tehnika primorava one koji je koriste da pišu dobro strukturisane i razumljive specifikacije, koje se dobro ponašaju?

U slučaju našeg posebnog problema, prvi korak pri izboru specifikacione tehnike jeste odrediti koji je od navedenih kriterijuma posebno važan. Prioriteti kriterijuma su različiti za različite probleme. Ardis je sa svojim kolegama bio zainteresovan za razvoj telefonskih komutacionih sistema, pa su oni za taj sistem procenjivali koliko je koji kriterijum koristan u razvoju reaktivnih sistema. Nisu uzimali samo efekte koje je kriterijum imao na aktivnosti u vezi sa zahtevima, već takođe i njegove efekte na druge aktivnosti unutra životnog ciklusa. Tabela 4.4 prikazuje rezultate njihovog ocenjivanja. Drugi korak za izbor specifikacione tehnike jeste ocenjivanje svake od potencijalnih tehnika u smislu datog kriterijuma. Na primer, Ardis je sa svojim kolegama ocenio Z tehniku kao jaku u modularnosti, apstrakciji, mogućnosti verifikacije, proizvoljnosti, zrelosti tehnike i modelovanju podataka; adekvatnu u pogledu primenljivosti, mogućnosti proveravanja i održavanja, ispravnosti, zrelosti alata, krivulje učenja i discipline; slabu u mogućnosti implementacije, testiranja i simulacije. Neke od njihovih ocena, kao što je ona da Z ima nasleđenu podršku modularnosti, važe za sve tipove problema, dok druge ocene, kao što je primenljivost, jesu specifične za tip problema. Na kraju, biramo specifikacionu tehniku koja najbolje podržava kriterijum koji je najvažniji za dati problem.

Tabela 4.4 Važnost specifikacionih kriterijuma tokom životnog ciklusa reaktivnog sistema (Ardis i dr. 1996) (Z - zahtevi, P - projektno rešenje, I - implementacija, T - testiranje, O - održavanje, D - drugo) © 1996 IEEE

Z	P	I	T	O	D	Kriterijum
+		+				Primenljivost
		+		+		Mogućnost implementacije
+	+		+			Mogućnost testiranja/simulacije
+			+	+		Mogućnost proveravanja
				+		Mogućnost održavanja
	+		+			Modularnost
+	+					Nivo apstrakcije/izražajnosti
+			+			Ispravnost
+	+	+	+	+		Mogućnost verifikacije
		+		+		Bezbednost izvršavanja
		+	+	+		Zrelost alata
+						Proizvoljnost
					+	Krivulja učenja
					+	Zrelost tehnike
	+					Modelovanje podataka
+	+	+		+		Disciplina

Kako nijedan pristup nije univerzalno primenljiv na sve sisteme, biće neophodno da kombinujemo više pristupa za kompletno definisanje svih zahteva. Neki metodi su bolji za snimanje kontrolnog toka i sinhronizacije, dok su drugi metodi bolji kod snimanja

transformacija podataka. Neki problemi se lakše opisuju u funkciji događaja i akcija, dok se drugi problemi bolje opisuju u funkciji kontrolnih stanja koja odražavaju faze ili ponašanje. Prema tome, korisno je upotrebiti jedan metod za zahteve u pogledu podataka, a drugi za opisivanje procesa ili vremenski zavisnih aktivnosti. Možda će biti potrebno da izrazimo promene u ponašanju kao i globalne invarijante. Modeli koji su za učesnike u razvoju adekvatni, možda su teški za tim za testiranje. Prema tome, izbor specifikacione tehnike (ili tehnika) je povezan sa karakteristikama datog projekta i prioritetima razvojnog tima i naručioca.

4.12 PRIMER IZ INFORMACIONOG SISTEMA

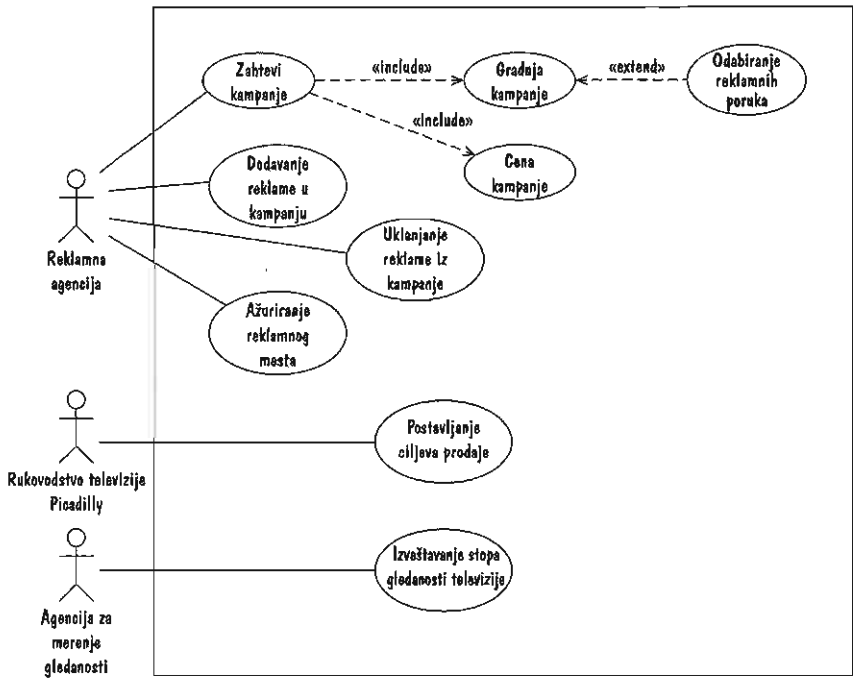
Podsetimo se našeg primera televizije Picadilly koja prodaje oglasno vreme za oblast za koju televizija Picadilly ima dozvolu. Možemo da koristimo više specifikacionih notacija za modelovanje zahteva u vezi sa kupovinom i prodajom oglasnog vremena. Dati problem predstavlja informacioni sistem, pa ćemo zato koristiti samo notacije koje su orijentisane prema podacima.

Prvo, možemo da nacrtamo dijagram slučajeva korišćenja za predstavljanje ključnih upotreba sistema, prikazujući očekivane korisnike i glavne funkcije koje svaki korisnik može da inicira. Delimični dijagram može da izgleda kao na slici 4.29. Napomenimo da ovaj dijagram na visokom nivou predstavlja snimak suštinske funkcionalnosti sistema, ali ne prikazuje ništa o načinima na koje svaki od tih slučajeva korišćenja može da se završi uspehom ili neuspehom. Na primer, zahtev kampanje ne bi uspeo ako je svo reklamno vreme već prodato. Dijagram takođe ne prikazuje ništa o tipu informacija koje mogu da predstavljaju ulaz sistema, koje sistem obrađuje, ili koje daje na izlazu. Potrebno nam je još informacija o svakoj od upotreba, da bismo bolje razumeli problem.

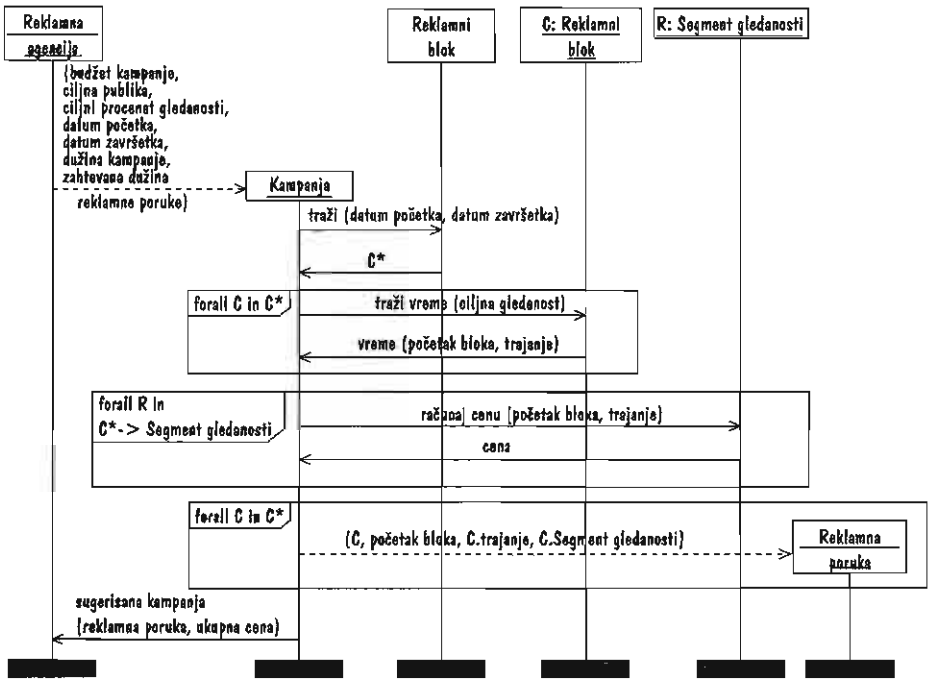
Sledeći korak može biti crtanje tragova događaja, kao što je onaj prikazan na slici 4.30, koji prikazuje tipične scenarije u okviru slučajeva korišćenja. Na primer, kampanja mora da zadovolji sledeće zahteve:

- pretraživanje svih relevantnih reklamnih blokova da bi se videlo ima li neprodatog vremena i koja je verovatnoća da će reklame u neprodatom vremenskom bloku gledati ciljna grupa kojoj je kampanja namenjena;
- izračunavanje cene kampanje na osnovu cena pronađenih raspoloživih reklamnih mesta;
- rezervacija raspoloživih reklamnih poruka za datu kampanju.

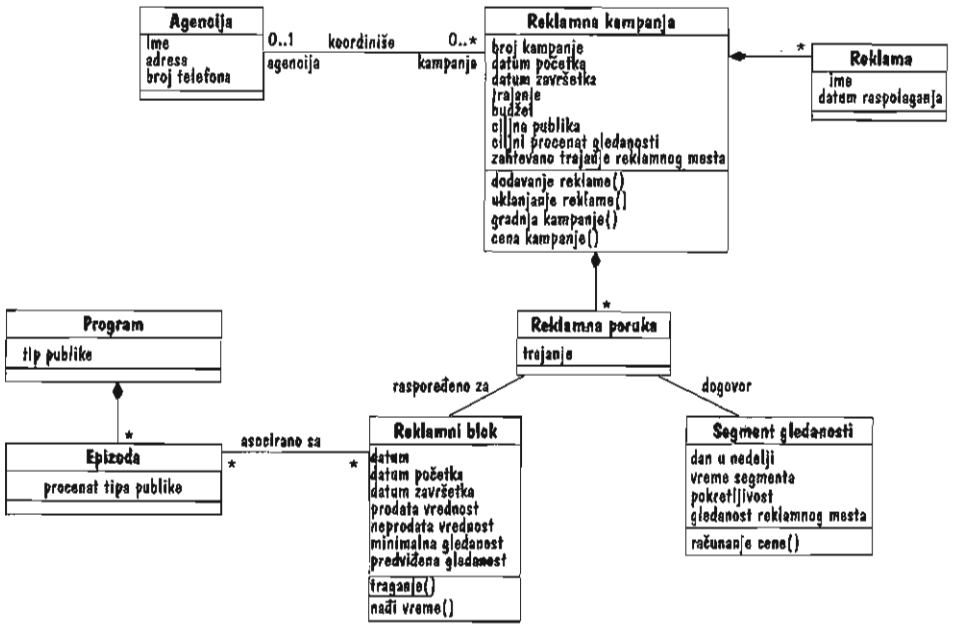
Slika 4.30 koristi dijagram redosleda poruka (*Message Sequence Chart*, MSC) u UML stilu, u kojem entiteti sa podvučenim imenima predstavljaju instance klasa, dok entiteti čija imena nisu podvučena predstavljaju klase. Prema tome, traganje za raspoloživim reklamnim mestima se radi prvo traženjem skupa `C*` relevantnih `Reklamnih pauza od klase`, a onda traženjem slobodnog vremena podesnog za kampanju u svakoj instanci `Reklamne pauze`. Pravougaonici okružuju sekvence poruka koje se ponavljaju za više instanci. Na primer, rezervisanje vremena u više instanci `Reklamne poruke` ili pravljenje više `Reklamnih mesta`. Rezultati u vidu kampanje reklamnih mesta i cene kampanje vraćaju se u `Reklamnu agenciju` koja je podnela zahtev. Slični tragovi mogu da se nacrtaju za druge moguće odgovore na zahtev za kampanju i za druge slučajeve. Prilikom crtanja tih slučajeva, započinjemo da uočavamo ključne entitete i odnose, koje možemo da snimimo u UML dijagramu klasa, kao na slici 4.31.



SLIKA 4.29 Slučaj korišćenja sistema oglašavanja televizije Picadilly (adaptirano iz Robertson i Robertson 1994)



SLIKA 4.30 Dijagram redosleda poruka za uspešanu reklamnu kampanju koja ispunjava zahteve (adaptirano iz Robertson i Robertson 1994)



SLIKA 4.31 Delimični UML dijagram klase reklamnog sistema televizije Picadilly (adaptirano iz Robertson i Robertson 1994)

Kompletna specifikacija u slučaju televizije Picadilly prilično je obimna i ds mnogo detalja. Ovde smo citirali primere koji pokazju koliko su različite notacije podesne za predstavljanje različitih aspekata zahteva u vezi sa datim problemom. Važno je da odaberemo kombinaciju tehnika koja daje kompletnu sliku problema, gde će se te tehnike koristiti za projektovanje, implementiranje i testiranje sistema.

4.13 PRIMER IZ PRAKSE

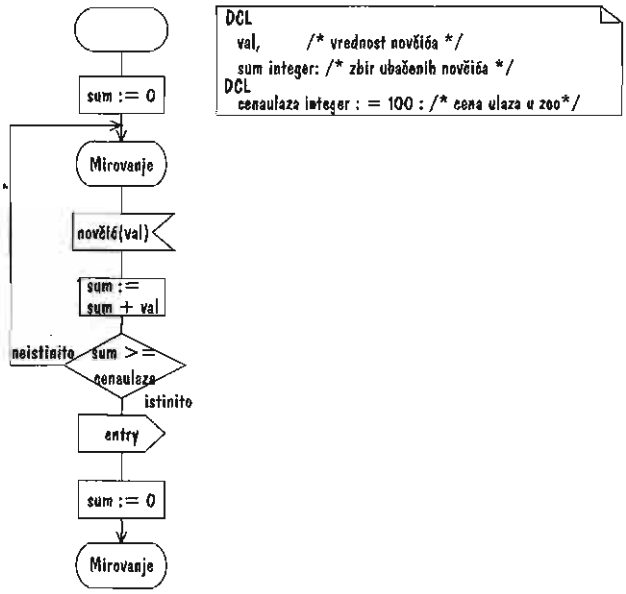
Setimo se eksplozije rakete Ariane-5 koja je bila prouzrokovana preuzimanjem dela koda iz Ariane-4. Nuseibeh (1997) je analizirao problem sa tačke gledišta višekratne upotrebljivosti zahteva. Drugim rečima, mnogi softverski inženjeri osećaju da mogu imati veliku dobit od višekratnog korišćenja specifikacije zahteva (i odgovarajućih projektnih rešenja, koda i testnih slučajeva) iz sistema koji su već razvijeni. Specifikacije koje su kandidati za višekratnu upotrebu identifikuju se pronalaženjem zahteva koji su funkcionalno ili po ponašanju isti ili slični, da bi se oni potom modifikovali gde i koliko je to neophodno. U slučaju Ariane-4, inercijalni referentni sistem (SRI) imao je mnoge funkcije koje su bile potrebne i za raketu Ariane-5.

Međutim, Nuseibeh napominje da i ako je potrebna funkcionalnost rakete Ariane-4 bila slična, Ariana-5 je imala svoje aspekete koji su bili znatno različiti. Naročito, funkcionisanje SRI-ja koje se nastavljalo nakon poletanja kod Ariane-4, nije bilo potrebno nakon poletanja Ariane-5. Da je validacija zahteva bila izvršena kako treba, analitičari bi otkrili da funkcije koje su aktivne nakon poletanja nisu bile povezane ni sa jednim od zahteva rakete Ariane-5 u definiciji i specifikaciji zahteva. Prema tome, validacija zahteva mogla je da ima presudnu ulogu u sprečavanju eksplozije rakete.

Druga preventivna mera mogla je da bude simulacija zahteva. Simulacija bi pokazala da SRI nastavlja da funkcioniše nakon poletanja. Zatim, projektno rešenje rakete Ariane-5 je moglo biti izmenjeno radi preuzimanja modifikovane verzije SRI koda. Razmotrimo ponovo spisak kriterijuma koje je predložio Ardis sa svojim kolegama, za izbor specifikacionog jezika. Taj spisak ima dva elementa koji su naročito važni za specifikaciju sistema poput sistema rakete Ariane-5, a to su mogućnost testiranja/simulacije i bezbednost izvršavanja. U Ardisovoj studiji, tim je ispitivao sedam specifikacionih jezika: Modeshart, VFSM, Esterel, Lotos,Z, SDL i C. Ispitivana je pogodnost korišćenja za svaki od navedenih kriterijuma. Jedino je SDL bio ocenjen kao „jak" u pogledu mogućnosti testiranja/simulacije i bezbednosti izvršavanja. SDL model sastoji se od više istovremenih procesa koji međusobno komuniciraju, kao što je, na primer, proces ubacivanja novčića u prorez, dat na slici 4.32.

Da bi se izvršila validacija SDL modela, sistemski zahtevi mogu da se napišu kao temporalno-logičke invarijante:

```
CLAIM;
    Rampa = zaključana
    IMPLIES (Rampa = zaključana)
        UNLESS (sum >= cenaulaza);
ENDCLAIM;
```



SLIKA 4.32 SDL proces ubacivanja novčića u prorez za slučaj obrtne rampe

SDL je zreo formalni metod koji uključuje objektno orijentisane koncepte i moćne odlike modelovanja. Na primer, procesi mogu da se dinamički aktiviraju, mogu im se dodeliti identifikatori i mogu da skladište trajne podatke. Događaji mogu da nose parametarske podatke i mogu da se usmeravaju u specifične procese referisanjem na identifikatore procesa, dok tajmeri mogu u realnom vremenu da modeluju kašnjenja i vremenska ograničenja. Na raspolaganju su komercijalni alati za podršku projektovanju,

otklanjanju grešaka i održavanju SDL specifikacija. Prema tome, jedna od mogućih tehnika prevencije mogla bi da bude upotreba specifikacionog metoda, kao što je SDL, uz prateće pomoćne alate.

U kasnijim poglavljima videćemo da se preventivne mere mogu preduzimati i tokom projektovanja, implementacije ili testiranja. Međutim, mere koje su se mogle preduzeti tokom analize zahteva, dovele bi do najvećeg razumevanja razlika između Ariane-4 i Ariane-5, i otkrile glavni uzrok greške.

Šta ovo poglavlje znači za vas

U ovom poglavlju videli smo najbolji način rada za razvoj kvalitetnih softverskih zahteva. Videli smo da učesnici u razvoju softvera ne treba izolovano da sprovode aktivnosti vezane za zahteve. Napori prilikom definisanja i specifikacije zahtevaju tesnu saradnju sa korisnicima, naručiocem, timom za testiranje, projektantima i drugim članovima tima. Međutim, tu su i znanja koja su važna i kojima treba samostalno da ovladate:

- Bitno je da dokumenti sa definicijom i specifikacijom zahteva opisuju problem, ostavljajući izbor rešenja projektantima. Najbolji način da se spreči da zalutate u prostor rešenja problema jeste opisivanje zahteva i specifikacija u funkciji pojava iz okruženja.
- Postoji više izvora i sredstava za izvođenje zahteva. Treba imati u vidu da postoje funkcionalni i kvalitativni zahtevi. Funkcionalni zahtevi objašnjavaju šta sistem treba da radi, dok kvalitativni zahtevi ograničavaju rešenja zavisno od bezbednosti, pouzdanosti, budžeta, rokova itd.
- Postoji više različitih tehnika za izradu definicija i specifikacija. Neke su opisne, kao što su dijagrami odnosa između entiteta i logika, dok ostali opisuju ponašanje, kao što su tragovi događaja, dijagrami toka podataka i funkcije. Za neke tehnike se koristi grafička notacija, dok su druge tehnike zasnovane na matematici. Kod svake tehnike je naglasak na sagledavanju problema s drugog aspekta i svaka sugeriše različite kriterijume za razlaganje problema na potprobleme. Često je poželjno da se koriste kombinacije tehnika za određivanje različitih aspekata sistema.
- Specifikacione tehnike takođe se razlikuju zavisno od njihovih pomoćnih alata, zrelosti, razumljivosti, lakoće korišćenja i formalnog matematičkog pristupa. Svaka tehnika treba da se procenjuje za konkretan projekat, jer ne postoji najbolja univerzalna tehnika.
- Odgovori na pitanja o zahtevima mogu da se dobiju pomoću modela ili prototipa. U oba slučaja, cilj je usredsređivanje na potproblem koji predstavlja srž pitanja, umesto da obavezno modelujemo ili pravimo prototip celokupnog problema. Ako pravimo prototip, potrebno je da unapred odlučimo da li će rezultujući softver biti zadržan ili odbačen.
- Potrebno je da se izvrši validacija zahteva da bi se osiguralo da oni tačno odražavaju očekivanja naručioca. Potrebno je takođe proveriti da li su zahtevi potpuni, ispravni, dosledni, izvodljivi itd., i to oslanjajući se ponekad na tehnike ili alate koji su pridruženi odabranim specifikacionim metodima. Na kraju, potrebno je da verifikujemo da li specifikacija ispunjava zahteve.

Šta ovo poglavlje znači za razvojni tim

Razvojni tim mora da zajednički radi na saznavanju, razumevanju i dokumentovanju zahteva. Često se različiti članovi tima usredsređuju na različita gledišta zahteva. Na primer, stručnjak za mreže može da radi na mrežnim zahtevima, stručnjak za korisnički interfejs na prikazima i izveštajima, stručnjak za bazu podataka na prihvatu podataka i skladištenju itd. Raznorodni zahtevi će biti integrisani u sveobuhvatnu celinu, pa stoga zahtevi moraju da se napišu na način koji omogućava njihovo povezivanje i kontrolisanje. Na primer, izmena jednog zahteva može da utiče na druge zahteve koji su s njim u vezi, a metodi i alati moraju da podrže izmene koje osiguravaju rano i brzo otkrivanje grešaka.

U isto vreme, deo tima koji se bavi zahtevima, mora tesno da sarađuje:

- sa naručiocem i korisnicima, tako da naš tim gradi proizvod koji zadovoljava njihove potrebe;
- sa projektantima, tako da oni konstruišu projektno rešenje koje ispunjava specifikaciju zahteva;
- sa timom za testiranje, tako da njihovi test skriptovi adekvatno procene da li implementacija zadovoljava zahteve;
- sa piscima dokumentacije, tako da oni mogu da pišu korisnička uputstva na osnovu specifikacije.

Tim mora takođe da obrati pažnju na merenja koja odražavaju kvalitet zahteva. Oni koji se bave merenjem mogu da predlažu aktivnosti tima, kao što je izrada prototipa nekih zahteva, kada indikatori pokažu da zahtevi nisu dovoljno razumljivi.

Na kraju, moramo da radimo kao tim na pregledu dokumenata sa definicijama i specifikacijama, kao i da ažuriramo te dokumente shodno promeni zahteva tokom razvoja i održavanja.

Šta ovo poglavlje znači za istraživače

Postoje mnoge oblasti istraživanja u vezi sa zahtevima. Istraživači mogu:

- da istražuju načine za smanjivanje stepena neizvesnosti i rizika u zahtevima;
- da razvijaju specifikacione tehnike i alate koji omogućavaju lakše dokazivanje pretpostavki i tvrdnji, kao i da demonstriraju doslednost, potpunost i determinizam;
- da razvijaju alate za omogućavanje praćenja kroz različite međuproizvode i finalne proizvode softverskog razvoja. Alati posebno mogu da ocene uticaj predložene izmene na proizvode, procese i resurse;
- da ocenjuju različite načine za pregledanje zahteva: alate, kontrolne spiskove, inspekcije, prolaženje kroz dokumente itd. Važno je da znamo koje tehnike su najbolje za koju situaciju;
- da stvaraju nove tehnike za simulaciju ponašanja zahteva;
- da nam pomažu da razumemo koji tipovi zahteva su najbolji za višekratno korišćenje u projektima koji slede, kao i kako da napišemo zahteve na način koji poboljšava njihovo kasnije višekratno korišćenje.

4.14 SEMINARSKI RAD

Klijenti u FCO-u (*Financial Consolidation Organization*) su pripremili sledeći skup zahteva na engleskom jeziku za sistem *Aranžer pozajmica*. Poput većine zahteva, ovi moraju biti ispitani na više načina da bi se odredilo da li su ispravni, potpuni i dosledni. Upotrebom zahteva datih ovde i pomoćnog materijala o *Aranžeru pozajmica* iz ranijih poglavlja, treba oceniti i poboljšati ovaj skup zahteva. Treba upotrebiti više tehnika predstavljenih u ovom poglavlju, uključujući merenje zahteva i Ardisove liste. Ako je neophodno, izraziti zahteve pomoću jezika za pisanje zahteva ili tehnike modelovanja, da bi se osiguralo da su takođe izražene i statičke i dinamičke osobine sistema.

Preduslovi i pretpostavke

- Sistem *Aranžer pozajmica* pretpostavlja da već postoje zajmodavci, zajmoprimci, raspoložive pozajmice i investitori koji su zainteresovani za kupovinu paketa zajmova.
- Sistem *Aranžer pozajmica* sadrži repozitorijum informacija o pozajmicama od raznih zajmodavaca. Taj repozitorijum može da bude prazan.
- U pravilnim intervalima, svaki zajmodavac obezbeđuje izveštaje navodeći realizovane pozajmice. Pozajmice koje je FCO već kupio biće naznačene u tim izveštajima.
- Svaka pozajmica u repozitorijumu *Aranžera pozajmica* predstavlja investiciju koja treba da se stavi u paket i proda sa drugim pozajmicama.
- Sistem *Aranžer pozajmica* može da se koristi za najviše četiri simultane analize pozajmica.

Opis funkcionalnosti na visokom nivou

1. Sistem *Aranžer pozajmica* će primati mesečne izveštaje od svih zajmodavaca o novim pozajmicama koje je izdao dati zajmodavac. Pozajmice koje je nedavno kupio FCO za svoj investicioni portfelj, biće posebno naznačene u izveštaju. Sistem *Aranžer pozajmica* koristiće informaciju iz izveštaja da ažurira svoj repozitorijum raspoloživih pozajmica.
2. Sistem *Aranžer pozajmica*primaće mesečne izveštaje od svakog zajmodavca, gde će se pružati informacije o ažuriranjima statusa pozajmica koje je izdao taj zajmodavac. Ažurirana informacija će uključivati: aktuelnu kamatnu stopu za hipotekarni kredit sa promenljivom kamatom, kao i status zajmoprimca u vezi sa pozajmicom (redovan, kasni ne izvršava obaveze). Za pozajmice u FCO-ovom portfelju *Aranžer pozajmica* će ažurirati podatke u repozitorijumu. Pozajmice koje nisu u FCO-ovom portfelju biće takođe ispitane u cilju određivanja da li administrativna zabrana zajmoprimca treba da se ažurira. FCO pruža svakom zajmodavcu format izveštaja, tako da svi izveštaji imaju zajednički format.
3. Analitičar pozajmica može da izmeni individualne zapise podataka, kao što je opisano u pravilima za rad sa podacima.
4. Validacija svih novih podataka mora da se obavi pre nego što se oni dodaju u repozitorijum (prema pravilima opisanim u ograničenjima podataka).
5. Analitičar pozajmica može da koristi *Aranžer pozajmica* za identifikovanje paketa pozajmica za prodaju pojedinim investitorima.

Funkcionalni zahtevi

1. Analitičar pozajmica treba da može da pregleda sve informacije u repozitorijumu za određenu instituciju zajmodavca, za određenu pozajmicu ili za određenog zajmoprimca.
2. Analitičar pozajmica može da napravi, gleda, edituje ili obriše pozajmicu iz portfelja ili paketa pozajmica.
3. Pozajmica se automatski dodaje u portfelj, gde *Aranžer pozajmica* čita izveštaje koje su obezbedili zajmodavci. Izveštaj može da se pročita iz *Aranžer pozajmica*a samo nakon što je specifikovan pridruženi zajmodavac.
4. Analitičar pozajmica može da definiše novog zajmodavca.
5. Analitičar pozajmica može da obriše zajmodavca samo ako u portfelju nema pozajmica pridruženih tom zajmodavcu.
6. Analitičar pozajmica može da promeni adresu i telefonski broj zajmodavca, ali ne i ime zajmodavca i identifikacioni broj.
7. Analitičar pozajmica ne može da izmeni informacije o zajmoprimcu.
8. Analitičar pozajmica može od sistema da zatraži sortiranje, pretraživanje ili organizaciju informacija o pozajmicama, prema određenom kriterijumu: iznosu, kamatnoj stopi, datumu dospeća, zajmoprimcu, zajmodavcu, tipu pozajmice ili nekom drugom kriterijumu koji je naveden da treba da se uključi u određene pakete pozajmica. Organizacioni kriterijum treba da uključi opsege, tako da će se informacija uključiti samo ako je u okviru dve zadate granice (na primer, 1. januar 2005. i 1. januar 2008.). Organizacioni kriterijum može takođe da se zasniva na izuzimanju, kao recimo sve pozajmice koje nisu označene ili sve pozajmice osim onih između 1. januara 2005. i 1. januara 2008.
9. Analitičar pozajmica treba da može da zahteva izveštaje u nekom od tri formata: u datoteci, na ekranu ili u vidu štampanog izveštaja.
10. Analitičar zahteva trebalo bi da može da zahteva sledeće informacije u izveštaju: bilo koji atribut pozajmice, zajmodavca ili zajmoprimca, kao i sumarne statistike atributa (srednju vrednost, standardnu devijaciju, dijagram raspodele i histogram). Informacija u izveštaju može da bude ograničena na podskup ukupnih informacija, kao što je opisano organizacionim kriterijumima analitičara pozajmica.
11. Analitičaru pozajmica treba omogućiti upotrebu *Aranžera pozajmica*a za pravljenje paketa pozajmica koje zadovoljavaju propisane karakteristike investicionog zahteva. Analitičar pozajmica može da identifikuje te pakete pozajmice na više načina:
    - ručnim identifikovanjem podskupa pozajmica koje moraju biti uključene u pakete pozajmice, bilo imenovanjem pojedinih pozajmica ili njihovim opisivanjem pomoću atributa ili opsega;
    - zadajući *Aranžer pozajmica*u investicioni kriterijum, omogućavajući *Aranžer pozajmica*u da izvršava zahteve za optimizaciju paketa pozajmica, radi odabiranja najboljeg skupa pozajmica koje zadovoljavaju date kriterijume;
    - kombinovanjem prethodna dva, gde se prvo odabira podskup pozajmica (ručno ili automatski), a zatim optimizovanjem odabranog podskupa prema investicionom kriterijumu.
12. Pravljenje paketa pozajmica sastoji se od dva koraka: Prvo, analitičar pozajmica pomoću *Aranžer pozajmica*a pravi paket pozajmica prema kriterijumu kao što je prethodno opisano.

Zatim paket pozajmica može da se prihvati, odbaci ili modifikuje. Modifikovanje paketa pozajmica znači da analitičar može da prihvati neke, ali ne sve pozajmice koje mu je predložio *Aranžer pozajmica* za stavljanje pozajmica u paket, a može da doda određenu pozajmicu u paket, pre nego što prihvati kompletan paket.

13. Analitičar pozajmice mora da bude u mogućnosti da obeleži pozajmice koje će biti ubačene u paket. Nakon što se pozajmica markira, nije više na raspolaganju za uključivanje u druge pakete pozajmica. Ako analitičar pozajmica markira pozajmicu i odluči da je ne uključi u paket pozajmica, markiranje mora biti uklonjeno, a pozajmica postaje raspoloživa za uključivanje u druge pakete pozajmica.
14. Kada je kandidat za paket pozajmica prihvaćen, pozajmice iz paketa više se ne razmatraju kao kandidati za druge pakete pozajmica.
15. Sve transakcije koje su u toku moraju se razrešiti pre nego što analitičar pozajmica napusti sistem *Aranžera pozajmica.*
16. Analitičar pozajmica može da pristupi repozitorijumu investicionih zahteva. Taj repozitorijum može da bude prazan. U slučaju investicionog zahteva, analitičar koristi ograničenja zahteva (u pogledu rizika, profita i dospeća) u cilju definisanja parametara paketa pozajmica. Zatim, sistem *Aranžer pozajmica* identifikuje pozajmice koje treba da budu satvljene u paket radi zadovoljavanja ograničenja zahteva.

Ograničenja podataka

1. Jedan zajmoprimac može da ima više od jedne pozajmice.
2. Svaki zajmodavac mora da poseduje jedinstvenu identifikaciju.
3. Svaki zajmoprimac mora da ima jedinstvenu identifikaciju.
4. Svaka pozajmica mora da ima najmanje jednog zajmoprimca.
5. Svaka pozajmica mora da ima iznos pozajmice od najmanje 1 000 dolara, ali ne više od 500 000 dolara.
6. Postoje dve vrste pozajmica zasnovane na iznosu pozajmice: regularne i džambo. Regularna pozajmica je ona gde je iznos manji ili jednak 275 000 dolara. Džambo pozajmica je ona gde iznos prelazi 275 000 dolara.
7. Smatra se da zajmoprimac uredno otplaćuje pozajmicu ako su sve pozajmice tog zajmoprimca uredno otplaćivane. Smatra se da zajmoprimac ne izmiruje obaveze ako bilo koja pozajmica datog zajmoprimca ima neizmirene obaveze. Za zajmoprimca se kaže da pozajmice otplaćuje sa zakašnjenjem ako se bilo koja pozajmica zajmoprimca otplaćuje sa zakašnjenjem.
8. Pozajmica ili zajmoprimac mogu da promene stanje iz redovnog u zakašnjenje, iz redovnog u neizmirivanje, iz zakašnjenja u redovno ili iz zakašnjenja u neizmirenje. Jednom kada pozajmica ili zajmoprimac pređu u stanje neizmirivanja obaveza, ne mogu da se izmene u druga stanja.
9. Pozajmica može da pređe iz promenljive stope (ARM) u fiksnu stopu (FM) ili iz FM u ARM.
10. Profit koji zahteva investitor je broj od 0 do 500, pri čemu 0 znači da za datu pozajmicu iz paketa nema profita. Profit koji je različit od nule predstavlja stopu vraćanja investicije za datu pozajmicu iz paketa. Ako je profit x, onda investitor očekuje da primi originalnu investiciju plus x procenata od originalne investicije,

nakon što se pozajmice otplate. Prema tome, ako pozajmica iz paketa košta 1 000 dolara, a investitor očekuje stopu vraćanja od 40%, onda se investitor nada da će imati 1 400 dolara nakon što se otplate sve pozajmice.

11. Pozajmica ne može da se pojavi u dva paketa.

Ograničenja projektnog rešenja i interfejsa

1. *Aranžer pozajmica* treba da radi na Unix sistemu.
2. Analitičaru pozajmica treba omogućiti istovremeni uvid u informacije o više pozajmica, zajmoprimaca i više institucija zajmodavaca.
3. Analitičaru pozajmica treba omogućiti da prolazi napred i nazad kroz informacije prikazane na ekranu. Kada je informacija suviše obimna da bi se smestila na jedan ekran, korisnik mora da bude obavešten da postoji dodatna informacija koja može da se prikaže.
4. Kada sistem prikazuje rezultate pretraživanja, aktuelni organizacioni kriterijumi moraju uvek da budu prikazani zajedno sa informacijom.
5. Jedan zapis ili red izlaza ne smeju nikada da se prekinu u sredini polja.
6. Korisnik mora biti obavešten u slučaju da je zahtev za pretraživanje neodgovarajući ili nelegalan.
7. Kada se pojavi greška, sistem treba da vrati korisnika na prethodni ekran.

Zahtevi u pogledu kvaliteta

1. Sistem mogu da koriste najviše četiri analitičara pozajmica istovremeno.
2. Ako je prikazana informacija ažurirana, informacija se osvežava u roku od pet sekundi od dodavanja, ažuriranja ili brisanja informacija.
3. Sistem mora da odgovori na zahteve analitičara pozajmica za dobijanje informacija u roku manjem od pet sekundi od podnošenja zahteva.
4. Sistem mora da bude na raspolaganju analitičaru pozajmica tokom 97% poslovnog dana.

4.15 KLJUČNE REFERENCE

Knjiga Michaela Jacksona *Software Requirements and Specifications* (1995) obezbeduje generalni savet o tome kako da se prevaziđu uobičajeni problemi u razumevanju i formulisanju zahteva. Njegova ideja može da se primeni na bilo koju tehniku definisanja zahteva. Knjiga *Exploring Requirements* (1989), koju su napisali Donald Gause i Gerald Weinber, bavi se ljudskom stranom procesa definisanja zahteva, tj. problemima i tehnikama rada sa naručiocima i korisnicima, kao i smišljanjem novih proizvoda.

Sveobuhvatni obrazac definisanja zahteva, koji su razvili James i Suzanne Robertson, nalazi se na veb lokaciji Atlantic System Guilda, http://www.systemsguild.com. Uz taj obrazac ide i opis procesnog modela Volere, koji predstavlja kompletan proces saznavanja i proveravanja skupa zahteva. Upotreba obrasca opisana je u njihovoj knjizi *Mastering the Requirements Process* (1999).

Knjiga *Object-Oriented Analysis* (1991), koju su napisali Peter Coad i Edward Yourdon, predstavlja klasično štivo o objektno orijentisanoj analizi zahteva. Najtemeljitije reference o UML-u (*Unified Modeling Language*) jesu knjige koje su napisali James Rumbaugh, Ivan Jacobson i Grady Booch, posebno *Unified Modeling Language Refe-*

rence Manual. Tu su i dokumenti čiji je izdavač Object Management Group, i koji mogu da se preuzmu sa veb lokacije organizacije http://www.omg.org. Knjiga *Analysis Patterns* (1996), čiji je autor Martin Fowler, pruža smernice kako UML treba da se koristi za modelovanje uobičajenih poslovnih problema.

Počev od 1993, udruženje IEEE Computer Society počelo je da sponzoriše dve konferencije koje su direktno u vezi sa zahtevima i koje se naizmenično održavaju svake druge godine: International Conference on Requirements Engineering i International Symposium on Requirements' Engineering. Te dve konferencije sjedinile su se 2002. i formirale International Requirements Engineering Conference, koja se održava svake godine. Informacije o predstojećim konfernecijama i o zbornicima sa prošlih, mogu da se pronađu na veb stranici http://www.computer.org, čiji je vlasnik Computer Society.

Requirements Engineering Journal usredsređuje se isključivo na nove rezultate u saznavanju, predstavljanju i validaciji zahteva, uglavnom s obzirom na softverske sisteme. Časopis IEEE Software imao je specijalna izdanja o inženjerstvu zahteva u martu 1994, martu 1996, martu/aprilu 1998, maju/junu 2000, januaru/februaru 2003. i martu/aprilu 2004. Druge IEEE publikacije često imaju specijalna izdanja o pojedinim tipovima analize zahteva i metodima specifikacije. Na primer, septembra 1990. izdanja časopisa *IEEE Computer*, *IEEE Software* i *IEEE Transactions on Software Engineering* usredsredila su se na formalne metode, a to je bio slučaj i maja 1997. i januara 1998. sa izdanjima časopisa *IEEE Transactions on Software Engineering*, kao i aprila 1996. sa izdanjem časopisa *IEEE Computer*.

Postoji više standarda u vezi sa softverskim zahtevima. Ministarstvo odbrane SAD proizvelo je standard MilStd-498, Data Item for Software Requirements Speifications (SRS). Organizacija IEEE proizvela je standard IEEE Std 830-1998, koji predstavlja skup preporučenih praksi i standarda za formulisanje i strukturisanje specifikacija zahteva.

Postoji više alata koji podržavaju formulisanje zahteva i mogućnost praćenja. DOORS/ERS (Telelogic), Analyst Pro (Goda Software) i RequisitePro (IBM Rational) jesu popularni alati za upravljanje zahtevima, praćenje zahteva sve do artefakta, praćenje izmena, kao i ocenjivanje uticaja izmena. Mnoge notacije za modelovanje poseduju pomoćne alate koji podržavaju barem pravljenje i editovanje modela, obično podržavaju neki oblik proveravanja dobre formiranosti, kao i generisanje izveštaja, a u najboljem slučaju nude i automatsku validaciju i verifikaciju. Nezavisno istraživanje alata za rad sa zahtevima nalazi se na lokaciji www.systemsguild.com.

Postoji i radna grupa IFIP Working Group 2.9 koja se bavi inženjerstvom softverskih zahteva. Neke od prezentacija sa njihovih godišnjih sastanaka raspoložive su na njihovoj veb lokaciji http://www.cis.gsu.edu/~wrobinso/ifip2_9.

4.16 VEŽBE

1. Učesnici u razvoju rade zajedno sa naručiocima i korisnicima na definisanju zahteva i određivanju šta će da radi predloženi sistem. Ako, nakon što je sagrađen, sistem radi prema specifikaciji, ali ošteti nekoga fizički ili finansijski, ko je za to odgovoran?
2. Između mnogih nefunkcionalnih zahteva, koji mogu da se uključe u specifikaciju, jesu i oni vezani za bezbednost i pouzdanost. Kako možemo da osiguramo da se ti zahtevi mogu testirati na način kako su to definisali Robinsonovi? Naročito, kako možemo da demonstriramo pouzdanost sistema od koga se zahteva da nikada ne doživi otkaz?

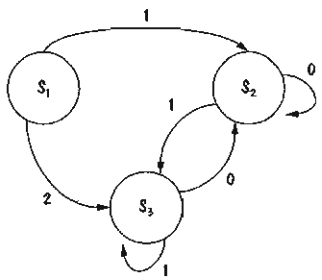
3. Tokom prvih sastanaka sa naručiocem, on navodi sledeće „zahteve" za sistem koji želi da vi sagradite:

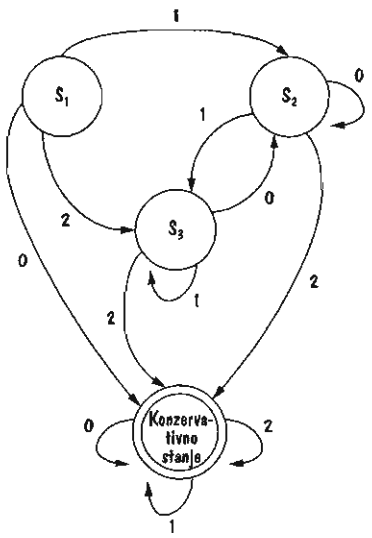
   (a) klijentski demon mora da bude nevidljiv za korisnika;
   (b) sistem treba da obezbedi automatsku verifikaciju oštećenih veza ili zastarelih podataka;
   (c) interne konvencije za imenovanje treba da osiguraju da zapisi budu jedinstveni;
   (d) komunikacija između baze podataka i servera treba da bude šifrovana;
   (e) između naslovnih grupa (tipova zapisa u bazi podataka) mogu da postoje odnosi;
   (f) datoteke treba da budu organizovane u grupe zavisnosti datoteka;
   (g) sistem mora da ima interfejs ka Oracle bazi podataka;
   (h) sistem mora da opslužuje 50 000 korisnika istovremeno.

   Treba klasifikovati sve što je navedeno kao funkcionalni zahtev, zahtev u pogledu kvaliteta, projektno ograničenje ili ograničenje procesa. Šta od toga može biti prevremena projektna odluka? Treba ponovo izraziti svaku od tih odluka kao zahtev koji je projektna odluka imala nameru da zadovolji.

4. Napisati tabelu odluka koja određuje pravila za igru „dame".
5. Ako tabela odluka ima dve identične kolone, onda je specifikacija zahteva redundantna. Kada možemo da kažemo da li je specifikacija kontradiktorna? Koje druge karakteristike tabele odlučivanja nas upozoravaju na probleme u zahtevima?
6. Napisati Parnasovu tabelu koja opisuje izlaz algoritma za pronalaženje korena kvadratne jednačine pomoću kvardatne formule.
7. Napisati specifikaciju konačnog automata za ilustrovanje zahteva za automatsku bankarsku mašinu (ABM).
8. Specifikacija konačnog automata je **kompletna** ako i samo ako postoji specifikovan prelaz za svaku moguću kombinaciju stanja i ulaznih simbola. Nekompletnu specifikaciju možemo da izmenimo da postane kompletna samo dodavanjem novih stanja, koja se nazivaju konzervativna stanja; nakon prelaza u konzervativno stanje, sistem ostaje u tom stanju, bez obzira na ulaz. Na primer, ako su 0, 1 i 2 jedini mogući ulazi, sistem prikazan na slici 4.33 može da bude kompletiran dodavanjem konzervativnog stanja, kao što je prikazano na slici 4.34. Na istu način, kompletirati specifikaciju automata iz vežbe 7.
9. **Osobina bezbednosti** je invarijantna osobina koja određuje da neko loše ponašanje ne može nikada da se dogodi. Na primer, osobina bezbednosti problema obrtne rampe je da broj ulaza u zoološki vrt nije nikada veći nego broj plaćenih ulaznica. **Životna osobina** jeste osobina koja određuje da pojedino ponašanje može eventualno da se dogodi. Na primer, životna osobina problema obrtne rampe je da kada se plati ulaznica, obrtna rampa postaje otključana. Slično tome, životna osobina bibliotečkog sistema je da svaki zahtev za pozajmicu, koji podnese član bez neizmirenih kazni, ima uspešan ishod. Te tri osobine, kada se logički izraze, izgledaju ovako:

SLIKA 4.33 Originalni sistem iz vežbe 7.





SLIKA 4.34 Kompletni sistem sa konzervativnim stanjima iz vežbe 7.

- □ (broj_novčića ≥ broj_ulazaka)
- □ (ubacivanje_novčića ⇒ rampa=otključana)
- □ ((pozajmljivanje(Član, Pub) ∧ Član.kazne = 0) ⇒
  ◊∃ Pozajmica. [Pozajmica.pozajmljivač=Član ∧ Pozajmica.Publikacija = Pub])

Navesti osobine bezbednosti i životne osobine specifikacije automatske bankarske mašine iz vežbe 6. Izraziti te osobine pomoću temporalne logike.

10. Dokazati da osobine bezbednosti i životne osobine iz vežbe 8 važe za model automata prema specifikaciji za automatsku bankarsku mašinu iz vežbe 7. Koje pretpostavke treba da napravite o okruženju ABM-a (tj. da mašina ima dovoljno gotovine), da bi dokazi bili uspešni?

11. Nekada deo sistema može da se sagradi na brzinu u cilju demonstriranja naručiocu da je neka funkcionalnost izvodljiva. Taj prototip sistema je obično nedovršen. Realni sistem se konstruiše tek nakon što naručilac i razvojni tim ocene prototip. Da li dokument sa sistemskim zahtevima treba da se napiše pre razvoja prototipa ili nakon toga? Zašto?

12. Napisati skup UML modela (dijagram slučajeva korišćenja, MSC dijagrame, dijagram klase) za onlajn telefonski direktorijum koji treba da zameni telefonski imenik koji obezbeđuje vaša telefonska kompanija. Direktorijum treba za zadato ime da vrati telefonski broj. Treba takođe da prikazuje pozivne brojeve za različite delove zemlje i da generiše telefonske brojeve za vašu oblast.

13. Nacrtati dijagrame toka podataka za ilustrovanje funkcija i toka podataka za sistem onlajn telefonskog direktorijuma navedenog u prethodnom problemu.

14. Koje su prednosti razdvajanja funkcionalnog toka od toka podataka?

15. Koje posebne vrste problema se predstavljaju prilikom određivanja zahteva sistema u realnom vremenu?

16. Predstaviti razlike između specifikacije objektno orijentisanih zahteva i funkcionalne dekompozicije.

17. Napisati Z specifikaciju za sistem raspoređivanja prezentacija. Sistem vodi evidenciju o tome koji prezentatori održavaju prezentacije koga dana. Nijedan prezentator ne treba da bude raspoređen da održava više od jedne prezentacije. Za jedan određeni datum, ne treba da se organizuje više od četiri prezentacije. Treba da postoje sledeće operacije: `Dodavanje` i `Uklanjanje` prezentacija iz rasporeda; `Zamena` datuma dve prezentacije; `Listanje prezentacija` raspoređenih za određeni datum; `Listanje prezentatora`, tj. datuma kada je određeni prezentator određen da govori; `Podsetnik`, tj. slanje poruke koja podseća prezentatora na dan prezentacije. Možete da definišete i dodatne operatore koji mogu da pomognu kod uprošćavanja specifikacije.
18. Dovršiti delimičnu SDL specifikaciju podataka za problem biblioteke sa slike 4.20. Naročito, napisati aksiome za negeneratorske operacije `poništavanje rezervacije`, `jesteNaPozajmici` i `jesteNaRazervaciji`. Modifikovati aksiome za operaciju `poništavanje rezervacije`, tako da ta operacija pretpostavlja da višestruki zahtevi `rezervacije` određene `stavke` mogu da se dogode između dva zahteva `poništavanja rezervacije` te stavke.
19. Na koje vrste problema treba da obratite pažnju kada pregledate zahteve? Treba napraviti listu provera tih problema. Da li lista za proveru može da bude univerzalno primenljiva ili je bolje koristiti listu za proveru koja je specifična za oblast primene?
20. Da li je ikako moguće da dokument sa definicijom zahteva bude isti kao i specifikacija zahteva? Šta su prednosti i mane postojanja dva dokumenta?
21. Pfleeger i Hatton (1997) ispitali su kvalitet sistema koji je bio specifikovan pomoću formalnih metoda. Oni su pronašli da je sistem bio neobično dobro strukturisan i lak za testiranje. Oni su bili mišljenja da je visok kvalitet bio posledica temeljitosti specifikacije a ne obavezno i njene formalnosti. Kako biste projektovali studiju za određivanje da li formalnosti ili temeljitost dovode do visokog kvaliteta?
22. Ponekad naručilac ima zahtev za koji znate da se ne može implementirati. Da li u svakom slučaju treba da se složite oko stavljanja zahteva u definicione i specifikacione dokumente, misleći da možete doći do nekog novog načina za zadovoljavanje zahteva, ili misleći da možete da naknadno tražite odbacivanje tog zahteva? Diskutujte koje su etičke posledice obećanja koja se ne mogu ispuniti.
23. Pronađite skup zahteva prirodnog jezika, na vašem poslu ili na veb lokaciji ove knjige. Pregledajte zahteve u cilju određivanja da li postoje neki problemi. Na primer, da li su zahtevi dosledni? Višeznačni? Suprotstavljeni? Da li sadrže odluke o projektnim rešenjima ili implementaciji? Koje tehnike predstavljanja mogu da pomognu kod otkrivanja i eliminisanja tih problema? Ako problemi ostanu u zahtevima, koji je njihov verovatni uticaj nakon projektovanja i implementacije sistema?

5 Dizajniranje sistema



U ovom poglavlju obradivaćemo:
- konceptualni i tehnički dizajn;
- dizajnerske stilove, tehnike i alatke;
- karakteristike dobrog dizajna;
- proveravanje dizajna;
- dokumentovanje dizajna.

U prethodnom poglavlju naučili smo kako da u saradnji sa svojim klijentima utvrdimo šta bi oni želeli da taj sistem radi. Kada se obavi analiza zahteva, kao rezultat se dobijaju dva dokumenta: jedan za klijenta gde su obuhvaćene njegove potrebe, a drugi za dizajnere u kojem se problem objašnjava tehničkom terminologijom. Sledeći korak u razvoju je prevođenje tih želja u rešenje: dizajn koji će zadovoljiti potrebe klijenta. U ovom poglavlju opisaćemo šta treba da se uradi i na koji način.

5.1 ŠTA JE TO DIZAJN?

Naši klijenti obično žele novi sistem bilo zato što sistem ne postoji ili zato što postojeći sistem ima neka svojstva koja nisu poželjna. U svakom slučaju, dokumenti zahteva opisuju problem koji bi sistem trebalo da reši. **Dizajn** je kreativni proces pretvaranja problema u rešenje; opis rešenja se takođe zove **dizajn**. Da bismo shvatili razliku između zahteva i dizajna, pogledajmo ponovo primer u kom Čak i Betsi Hauel žele novu kuću. Među njihovim zahtevima imamo stavke kao što su:

- prostor za igru troje dece i poseban prostor u kojem će spavati;
- spavaća soba za Čaka i Betsi;
- prostorija za kuvanje;
- grejanje za zimu i hlađenje za leto;
- unutrašnja vodovodna i električna instalacija.

Arhitekta uzima te zahteve u obzir i dizajnira kuću za Hauele. Arhitektonski dizajn sadrži konkretno rešenje, na primer: kuću sa četiri spavaće sobe na spratu, sa sobom za goste u prizemlju, kuhinjom, sobom za igranje i pomoćnom prostorijom. U suštini, arhitekta može da napravi nekoliko veoma različitih projekata od kojih svaki rešava problem. Na primer, jedan deci daje maksimalan prostor za igru, a drugi ima minimalan prostor za igru da bi svi članovi porodice imali velike spavaće sobe. Osim toga, mogu se razlikovati stilovi predloženih kuća: na primer, arhitekta može da ponudi planove za dvospratnu starinsku kuću, za vikendicu ili za modernu gradsku kuću. Zaista, različiti arhitekti će napraviti različite planove za rešenje istog problema. Svi predloženi projekti rešavaju problem i možda ne postoji „najbolji dizajn"; izbor porodice zavisiće od njihovih sklonosti prema različitim karakteristikama kuće.

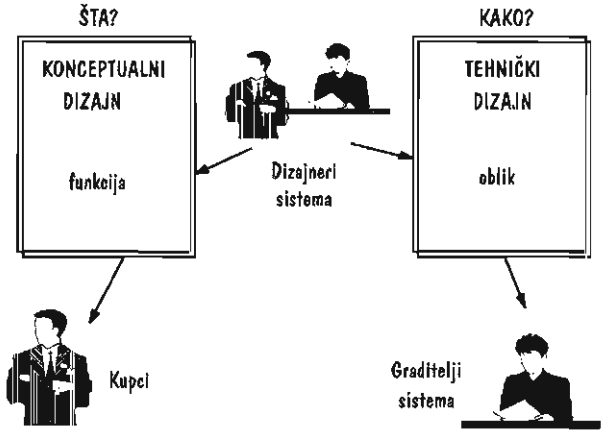
Softverski dizajn može se posmatrati na isti način. Specifikacija zahteva definiše problem. Zatim dolazi rešenje problema kao nešto što zadovoljava sve zahteve i specifikacije. U mnogim slučajevima je broj rešenja neograničen. Isto kao što će porodica Hauel konkretnu kuću koju želi da gradi izabrati među mnogim ponuđenim varijantama, i klijent će odlučiti da primeni jedno od više mogućih rešenja.

Priroda rešenja može da se menja i u toku opisa ili implementacije. Na primer, kada arhitekta pokaže porodici Hauel jedan skup nacrta, oni mogu da promene specifikacije ako im se nešto dopada ili ne dopada. Takve odluke nisu ni neobične ni nerazumne. Modifikacije ne moraju biti rezultat hira, mogu da nastanu zbog promenjenog doživljaja ili potrebe. Na primer, ako porodica odluči da ima još jednu bebu, možda će promeniti specifikaciju kuće i zatražiti da se doda još jedna soba. Ili ako Hauelovi utvrde da nešto nisu dobro razumeli ili su zaboravili da pomenu neki važan zahtev, logičnije je da promene specifikaciju nego da se pomire sa tim da žive u kući koja im iz nekih razloga ne odgovara. Na isti način, opis sistema može da se menja tokom ciklusa razvoja. Zaista, klijent često zajedno sa projektantima menja zahteve i posle okončanja polazne analize zahteva.

Konceptualni i tehnički dizajn

Da bi se zahtevi pretočili u sistem koji funkcioniše, dizajneri moraju da zadovolje i klijente i graditelje sistema u našem razvojnom timu. Klijenti shvataju šta sistem treba da radi. Isto tako, graditelji sistema moraju da znaju kako taj sistem treba da radi. Zbog toga je dizajn u stvari iterativni proces iz dva dela. Prvo proizvodimo **konceptualni dizajn** ili **dizajn sistema** koji opisuje klijentu tačno šta će sistem raditi. Pošto klijent odobri konceptualni dizajn, mi ga prevodimo u mnogo detaljniji dokument, **tehnički dizajn** koji omogućava graditeljima sistema da utvrde koji će hardver i softver biti potrebni za rešenje klijentovog problema. Proces je iterativan zato što dizajneri naizmenično rade na analizi zahteva, predlaganju mogućih rešenja, testiranju izvodljivosti svih aspekata rešenja, prikazivanju mogućnosti klijentima i dokumentovanju dizajna za programere. Ponekad se dizajn opisuje u jednom dokumentu, ali često postoje dva kao što je prikazano na slici 5.1. Oba dizajnerska dokumenta opisuju isti sistem, ali na različite načine zbog toga što su dokumenti namenjeni različitim korisnicima. Tako imamo konceptualni dizajn skoncentrisan na funkcije sistema i tehnički dizajn koji opisuje oblik sistema. Ovaj skup sličan je specifikaciji kuće gde kupci mogu da gledaju planove i prikaze, ali su graditeljima potrebne informacije o električnim i vodovodnim instalacijama, krovnim gredama i nosećim elementima. Hauelovi ne moraju da postanu građevinski eksperti da bi tačno

shvatili šta će da dobiju. Isto tako, graditelji ne moraju da znaju sve o klijentovom privatnom životu da bi im bile jasne njegove želje i potrebe.



SLIKA 5.1 Konceptualni i tehnički dizajn

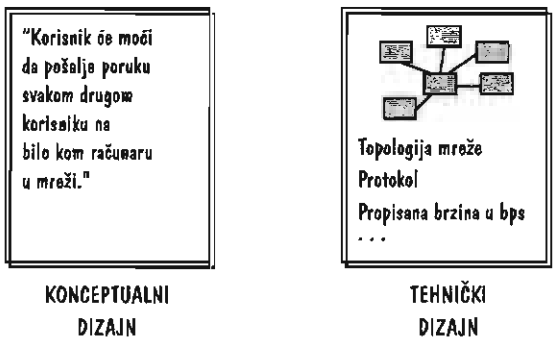
Kao što smo videli u prvom poglavlju, sistem se definiše svojim granicama, entitetima, atributima i odnosima. Konceptualni dizajn opisuje svaki od tih aspekata sistema u vidu odgovora na pitanja kao što su:

- Odakle će podaci stizati?
- Šta će se u sistemu događati sa podacima?
- Kako će sistem izgledati korisnicima?
- Šta će korisnici moći da biraju?
- Kakav je vremenski raspored događaja?
- Kako će izgledati izveštaji i ekrani?

Sistem se u konceptualnom dizajnu opisuje jezikom koji klijent razume, a ne računarskim žargonom i tehničkim terminima. Na primer, klijentu možemo reći da će meni na ekranu omogućiti korisnicima pristup do funkcija sistema. U konceptualnom dizajnu mogu se čak i nabrojati prihvatljivi odgovori korisnika i akcije do kojih oni dovode. Međutim, klijentu ne objašnjavamo kako se podaci čuvaju niti koja će vrsta sistema za upravljanje bazama podataka manipulisati podacima. Slično tome, kao što je prikazano na slici 5.2, klijentu možemo u konceptualnom dizajnu reći da se poruke usmeravaju sa jednog mesta na drugo. Međutim, ne navodi se mrežni protokol i topologija gde se opisuje kako sistem funkcioniše, a ne šta on radi. U konceptualni dizajn spada ono „šta" sistem radi, a u tehnički dizajn spada opis „kako"on to radi.

Ponekad su klijenti profesionalci i mogu da razumeju objedinjene i „šta" i „kako"; to se dešava kada su klijenti projektanti softvera, a mi gradimo alatku koju će oni koristiti za razvoj ili održavanje. U takvim slučajevima se tehnički i konceptualni dizajn može objediniti u jedan sveobuhvatan dokument dizajna. Inače je korisno da ti dokumenti budu odvojeni, ali povezani. Kao i u slučaju definicije i specifikacije zahteva važno je povezati konceptualni i tehnički dizajn tako da se promene do kojih dođe u jednom reflektuju u drugom dokumentu.

SLIKA 5.2 Razlike između dizajnerskih dokumenata



Dobar konceptualni dizajn bi trebalo da ima sledeće karakteristike:

- da bude pisan jezikom klijenta;
- da ne sadrži tehnički žargon;
- da opisuje funkcije sistema;
- da ne zavisi od implementacije;
- da bude povezan sa dokumentima specifikacije zahteva.

Drugim rečima, konceptualni dizajn omogućava klijentu da shvati šta će sistem raditi tako što objašnjava vidljive spoljašnje karakteristike sistema.

Nasuprot tome, tehnički dizajn opisuje konfiguraciju hardvera, softverske potrebe, komunikacione interfejse, ulaze u sistem i izlaze iz njega, mrežnu arhitekturu i sve ostalo čime se zahtevi prevode u rešenje klijentovog problema. Znači, opis u tehničkom dizajnu je tehnička slika specifikacije sistema. On obično najmanje sadrži:

- opis glavnih hardverskih komponenti i njihove funkcije;
- hijerarhiju i funkcije softverskih komponenti;
- strukture i tokove podataka.

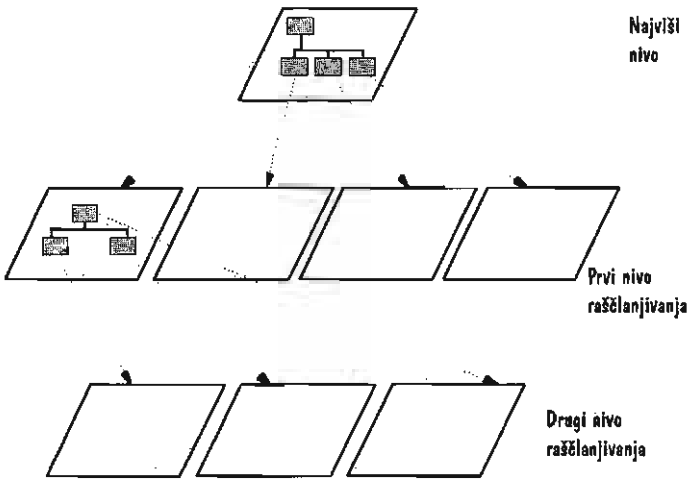
5.2 RAŠČLANJIVANJE I MODULARNOST

Projektovati sistem znači utvrditi skup komponenti i interfejsa među komponentama kojima se zadovoljava konkretan skup zahteva (DeMarco 1982). Kao što postoje mnogi načini da se zahtevi iskažu i dokumentuju, postoje mnogi načini da se napravi dobar dizajn. Ponekad se izbor zasniva na sklonostima dizajnera, a ponekad metod diktiraju zahtevana struktura sistema ili struktura podataka. Međutim, u svakom metodu postoji neka vrsta raščlanjivanja: počinje se od okvirnog opisa ključnih elemenata sistema a zatim se detaljno prikazuje kako će se elementi i funkcije sistema međusobno uklapati.

Wasserman (1995) predlaže da se dizajn realizuje na jedan od sledećih pet načina:

1. **Modularno raščlanjivanje:** Ova konstrukcija se zasniva na tome da se komponentama dodele funkcije. Dizajner počinje sa opštim opisom funkcija koje treba implementirati i gradi detaljna objašnjenja o organizaciji svake komponente i njenim relacijama sa ostalim komponentama.
2. **Raščlanjivanje na osnovu podataka:** Ovaj dizajn se zasniva na spoljašnjim strukturama podataka. Opšti opis prikazuje globalnu strukturu podataka, a detaljni opisi navode koji će elementi podataka biti obuhvaćeni i kakve su njihove međusobne veze.
3. **Raščlanjivanje na osnovu događaja:** Ovaj dizajn se zasniva na događajima koje sistem mora da obradi i koristi informacije o tome kako događaji menjaju stanje sistema. Opšti opis sadrži spisak različitih stanja, a detaljan opis navodi kako dolazi do transformacija stanja.
4. **Dizajn spolja ka unutra:** Ovaj pristup „crne kutije" zasniva se na onome što korisnik unosi u sistem. Znači, opšti opis navodi šta sve korisnik može da unese, a detaljni opisi opisuju šta će sistem uraditi sa svakim od unetih elemenata (kao i dobijene rezultate).
5. **Objektno orijentisani dizajn:** Ovaj dizajn definiše klase objekata i njihove međusobne veze. Na najopštijem nivou opisuju se tipovi objekata. Na detaljnijim nivoima se opisuju atributi objekata i akcije, a dizajn objašnjava međusobne veze objekata.

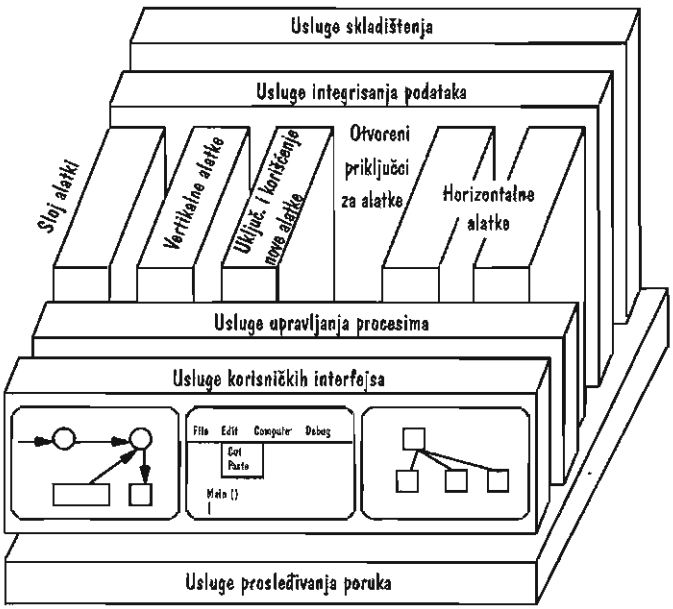
Znači, dizajn se može izvesti na osnovu sistemskih podataka, događaja, korisničkog unosa, opštih opisa funkcija ili kombinovanjem i pravljenjem hijerarhije informacija sa sve više detalja. Ovaj proces prikazan je na slici 5.3.



SLIKA 5.3 Nivoi raščlanjivanja

Da bismo pokazali kako funkcioniše raščlanjivanje, pretpostavimo da smo izabrali pristup zasnovan na podacima. Raščlanjivanje počinje od podataka i definicije tipova podataka do kojih se došlo tokom analize zahteva. Setićete se da je analiza uključivala analizu procesa koji se vrše na svakom tipu podataka, kao što su stvaranje, upiti, menjanje i brisanje. Sistem možemo da projektujemo počevši od objekata ili od tipova podataka tako što ćemo da organizujemo podatke i način na koji se njima manipuliše. Konkretno, mi projektujemo objekte i tipove objekata sa dozvoljenim transformacijama, a ne rukujemo podacima direktno. Rezultat ove analize podataka takođe je jedna hijerahija komponenti; svaki sledeći niži nivo sadrži sve više detalja o podacima. Ovo raščlanjivanje razlikuje se od opisa zahteva po tome što obuhvata informacije o tome *kako* su elementi podataka predstavljeni i povezani, a ne samo *šta* će sistem raditi.

Bez obzira na pristup dizajnu, svaka vrsta raščlanjivanja razdvaja dizajn na sastavne delove koje nazivamo **modulima** ili **komponentama**. Za sistem kažemo da je **modularan** kada svaku aktivnost sistema vrši samo jedna komponenta sa dobro definisanim ulazima i izlazima. U svakom slučaju, komponenta dizajna je entitet sa dobro definisanim ulazom, izlazom ili karakteristikama. Kažemo da je komponenta **dobro definisana** ako su svi njeni ulazi bitni za njenu funkciju, a sve izlaze proizvodi neka od njenih akcija. To znači da ako nedostaje jedan ulaz, komponenta nije u stanju da obavi svoju funkciju u celosti. Osim toga, izraz „dobro definisana" znači da ne postoje nepotrebni ulazi; svi ulazi se koriste za generisanje izlaza. Na kraju, komponenta je dobro definisana samo kada je svaki izlaz rezultat rada komponente i kada nijedan ulaz ne postaje izlaz, a da nije na neki način transformisan u komponenti.



SLIKA 5.4 Referentni model NIST/ECMA za integraciju okruženja (Chen i Norman 1992) (c) 1996 IEEE.

Raščlanjivanje je korisno na više načina. Isto kao što arhitekta može Hauelovima da pokaže opšte planove bez detalja o vodovodnim i električnim instalacijama, tako i arhitekta softvera može da upotrebi opšti dizajn i objasni glavne karakteristike ne ulazeći u detalje. Na primer, na slici 5.4 prikazan je referentni model NIST/ECMA za integraciju okruženja, predlog arhitekture kojom se različite softverske alatke uklapaju u razvojno okruženje. Model se naziva „toster" pošto njegove komponente liče na toster za hleb. Referentni model nam prikazuje da će okruženje pružati nekoliko vrsta usluga: skladištenje, integrisanje podataka, upravljanje procesima i korisnički interfejs. Model takođe pokazuje da je u dizajnu predviđeno uključivanje postojećih alatki bez ikakve modifikacije. Jasno je da ovaj opšti prikaz nije dovoljan programerima da bi ga implementirali bez dodatnog raščlanjivanja, ali je dovoljan da se dizajn okruženja prikaže klijentima i prodavcima alatki.

5.3 ARHITEKTONSKI STILOVI I STRATEGIJE

Videli smo da arhitekta počinje proces projektovanja kuće opštim pregledom kako će kuća da izgleda i kako će da funkcioniše. Arhitekta obično radi odozgo nadole, pravi crteže sa sve više i više detalja o kući. Shaw i Garlan (1996) predlažu da softverska arhitektura bude prvi korak i u dizajnu softvera. Oni razlikuju tri nivoa dizajna: arhitekturu, dizajn koda i izvršni dizajn.

1. **Arhitektura** povezuje sposobnosti sistema navedene u specifikaciji zahteva sa sistemskim komponentama koje će ih implementirati. Komponente su obično moduli, a arhitektura takođe opisuje njihove međusobne veze. Osim toga, arhitektura definiše operatore kojima se od podsistema prave sistemi.
2. **Dizajn koda** obuhvata algoritme i strukture podataka, a komponente su elementi programskog jezika, kao što su brojevi, znakovi, pokazivači i niti kontrole. Tu su takođe elementarni operatori uključujući osnovne jezičke primitive za aritmetiku i manipulaciju podacima, kao i mehanizmi za kompoziciju kao što su matrice, datoteke i procedure.
3. **Izvršni dizajn** još detaljnije izlaže dizajn koda. On opisuje dodelu memorije, formate podataka, šablone bitova i tako dalje.

Korisno je raditi odozgo nadole kada se prvo dizajnira arhitektura, zatim vrši dizajn koda i na kraju izvršni dizajn. Međutim, Krasner, Curtis i Iscoe (1987) su analizirali navike projektanata na 19 projekata; njihov izveštaj tvrdi, a to potvrđuju i drugi dokazi, da dizajneri u stvari naizmenično rade na tim nivoima kako sve više ulaze u rešenje i njegove posledice. Simon (1981) i Rittel i Webber (1984) napominju da je razvoj sistema loše definisan zadatak; moramo pokušati da nađemo rešenje da bismo shvatili sve nijanse problema. Na primer, uzmimo da grupa dizajnera odluči da bi sistemom trebalo upravljati pomoću tabela. Međutim, dok prave prototip nekih funkcija, dolaze do zaključka da dizajn kojim se upravlja pomoću tabela neće zadovoljiti zahtevano vreme odgovora. Zato ponovo dizajniraju sistem kako bi bio brži uz upotrebu matrica umesto tabela. Slično tome, dok dizajneri istražuju druge aspekte sistema, mogu u komunikaciji sa programerima ili ljudima koji testiraju sistem da promene dizajn kako bi unapredili implementaciju, mogućnosti testiranja ili održavanje. Ove iteracije znače da dizajneri moraju postepeno da rade na arhitekturi, na dizajnu koda i na izvršnom dizajnu onoliko koliko im to

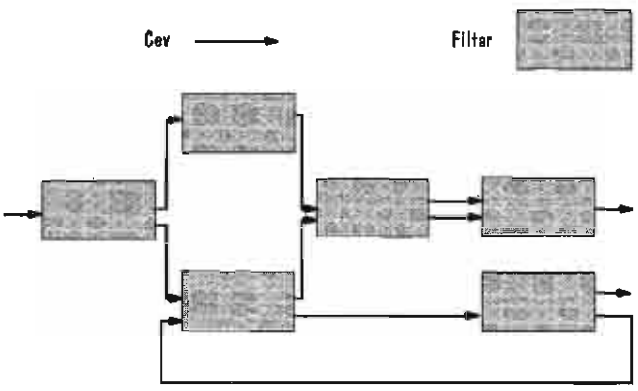
dozvoljavaju njihovo razumevanje i kreativnost. Ipak, važno je da arhitektura obezbedi kohezivnu „opštu sliku" za usmeravanje daljeg dizajna i razvoja; a naizmeničan rad na komponentama dizajna ne sme tu kohezivnost da naruši.

Kao što zgrade odražavaju određeni arhitektonski stil, tako postoje i stilovi arhitekture softvera. Stil obuhvata komponente, spojnice i ograničenja u kombinovanju komponenti. Shaw i Garlan (1996) primećuju da postoji 7 najčešće korišćenih stilova: cevi i filtri, objektno orijentisani dizajn, implicitno pozivanje, slojevi, skladišta, interpreteri i kontrola procesa. Oni tvrde da će razumevanje karakteristika pojedinih stilova pomoći da se lakše donese odluka o izboru onog koji je najpogodniji.

Cevi i filtri

Jedna komponenta iz sistema sa cevima i filtrima prikazanog na slici 5.5 kao ulaz i izlaz koristi tokove podataka koje nazivamo **cevima**. Često transformacija podataka iz ulaznih u izlazne počinje pre nego što komponenta koju nazivamo **filtar** završi sa čitanjem ulaznog toka. U ovoj vrsti sistema filtri su nezavisni i nisu svesni postojanja niti funkcija ostalih filtara u sistemu. Osim toga, ispravnost izlaza iz sistema ne zavisi od redosleda kojim se filtri primenjuju. Sistem sa cevima i filtrima se koristi kad god se prevodi program; ti filtri su redom: leksička analiza, raščlanjivanje, semantička analiza i generisanje koda. Sistemi sa cevima i filtrima imaju nekoliko značajnih svojstava:

- Dizajneri mogu da shvate uticaj celokupnog sistema na ulaze i izlaze kao kompoziciju filtara.
- Pošto se filtri mogu povezivati bez ograničenja, njihovo ponovno korišćenje u drugim sistemima je jednostavno.
- Razvoj sistema je jednostavan pošto se relativno lako mogu dodavati novi filtri i uklanjati stari.
- Pošto su filtri nezavisni, dizajneri mogu da simuliraju ponašanje sistema i analiziraju njegova svojstva kao što je propusna moć.
- Moguće je konkurentno izvršavanje filtara.



SLIKA 5.5 Cevi i filtri

Međutim, cevi i filtri imaju nekoliko nedostataka. Prvo, podstiču paketnu obradu i nisu primereni za interaktivne aplikacije. Drugo, kada su dva toka podataka povezana, sistem mora da održi njihovu međusobnu povezanost. Treće, pošto su filtri nezavisni, neki od njih možda ponavljaju pripremne funkcije drugih filtara što utiče na performansu i dovodi do koda višeg stepena složenosti. Na primer, svaki filtar koji obrađuje datumski objekat mora da proveri njegovu ispravnost: mesec od 1 do 12, dan od 1 do 31 i ispravan dan za dati mesec.

Objektno orijentisani dizajn

U poglavlju 4 videli smo da zahtevi mogu da se organizuju prema objektima i njihovim apstraktnim tipovima. Dizajn takođe može da gradi svoje komponente oko apstraktnih tipova podataka. To jest, svaka komponenta je primerak apstraktnog tipa podataka. Na taj način dizajn zasnovan na objektima mora da ima dve značajne karakteristike: objekat mora da očuva integritet predstavljanja podatka, a predstavljanje podataka mora biti sakriveno od ostalih objekata (enkapsuliranje smo opisali u poglavlju 4). Poslednja karakteristika olakšava promenu implementacije bez poremećaja u ostatku sistema. Osim toga, kombinovanje podataka i rutina koje njima manipulišu podstiče dizajnere na raščlanjivanje osnovnog problema na skup agenata koji su u međusobnoj interakciji.

Međutim, da bi objekti mogli da utiču jedni na druge svaki od njih mora da zna identitet onog drugog. Ova situacija se razlikuje od sistema sa cevima i filtrima gde su filtri potpuno međusobno nezavisni. Ova zavisnost znači da promena u identitetu jednog objekta zahteva da se promene sve ostale komponente koje pozivaju promenjeni objekat. Objektno orijentisani dizajn detaljnije ćemo obraditi u poglavlju 6.

Implicitno pozivanje

Model dizajna za implicitno pozivanje pokreću događaji, a zasnovan je na pojmu difuznog emitovanja. Umesto da direktno poziva proceduru, komponenta objavljuje da je nastupio jedan događaj ili više njih. Zatim, ostale komponente mogu tim događajima da pridružuju procedure (to se naziva *registrovanjem* procedure), a sistem poziva sve takve registrovane procedure. Razmena podataka u ovoj vrsti sistema mora da se obavlja pomoću zajedničkih podataka u skladištu. Ova vrsta dizajna se često koristi u mrežama sa komutiranjem paketa ili u sistemima zasnovanim na učesnicima, za obezbeđivanje konzistentnosti u bazama podataka i u korisničkim interfejsima za razdvajanje prikaza podataka od aplikacija koje upravljaju podacima.

Na primer, Reiss (1990) izveštava o okruženju koje se zove Field, u kojem se alatke kao što su editori registruju za događaje koji nastupaju tokom rada programa za pronalaženje grešaka. Da biste razumeli postupak, zamislite kako program za pronalaženje grešaka obrađuje kôd, red po red i utvrdi da nedostaje jedna promenljiva. On objavljuje događaj „nedostaje promenljiva", a sistem automatski poziva editor teksta registrovan za taj događaj. Na ovaj način editor automatski omogućava korisniku da uredi red u kojem promenljiva nedostaje. Dajemo još nekoliko primera događaja koje objavljuje program za pronalaženje grešaka.

```
DEBUG VALUE <system><file><line><var><value>
DEBUG ENTER <system><file><func><line><value>
DEBUG EXIT <system><file><func><line><value>
EVENT ADD <system><id><event_type><file><line><text>
```

```
EVENT REMOVE <system><id><event_type><file><line><text>
STOP-ERROR <signal><file><line>
DEBUG AT <system><file><func><line>
DEBUG FOCUS <system><file><func><line>
DEBUG CLEAR <system>
DEBUG RESET <system>
WHERE <system><level><file><func><line><addr><args>
WHERE DUMP <system><level><name><value>
WHERE BEGIN <system>
WHERE END <system><level>
DEBUG SYSTEM <system>
DEBUG NOSYSTEM <system>
UPDATE <system><fileline>
```

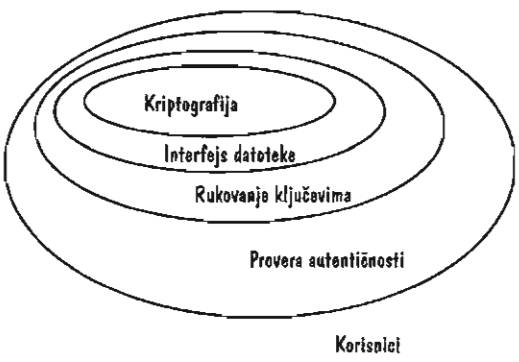
Program za pronalaženje grešaka ne zna koje su se alatke registrovale za različite događaje i nema nikakvu kontrolu nad postupkom koji ostale alatke sprovode kao odgovor na nastali događaj. U ovom stilu dizajna, kada jedna komponenta ili sistem objavi događaj, ona uglavnom ne zna na koje komponente će taj događaj uticati. Zbog toga, sistemi sa implicitnim pozivanjem obično sadrže i neko eksplicitno pozivanje.

Ovaj stil dizajna posebno je koristan za ponovno korišćenje komponenti dizajna iz drugih sistema. Pošto svaka komponenta može da se registruje za neki događaj, u sistem može da se doda već postojeća komponenta iz nekog drugog sistema i ona može sebe da registruje nezavisno od ostalih komponenti. Slično tome, kako se sistem razvija i zahteva nadogradnje, lako je ukloniti staru komponentu i dodati novu. S druge strane, najveća mana ovog dizajnerskog stila je to što nije sigurno da će neka komponenta odgovoriti na događaj. Ta zavisnost od konteksta i redosleda događaja veoma otežava testiranje sistema i proveru njegove ispravnosti.

Slojevi

Sistemi sa slojevima organizovani su kao što je prikazano na primeru sa slike 5.6. Slojevi su hijerarhijski raspoređeni; svaki sloj pruža usluge sledećem sloju koji ga okružuje i igra ulogu klijenta sloju koji obuhvata. U nekim sistemima svi slojevi imaju pristup nekim ili svim ostalim slojevima; u drugim sistemima svaki sloj ima pristup samo susednim slojevima. Dizajn sadrži protokole koji regulišu način međusobnog dejstva za svaki par slojeva.

SLIKA 5.6 Bezbednosna arhitektura sa slojevima



Da biste shvatili kako funkcioniše ova vrsta sistema, razmotrite sliku 5.6 na kojoj je prikazan sistem za bezbednost datoteka. Unutrašnji sloj, kriptografija, obuhvata funkcije za kriptovanje i dekriptovanje ključa koji se koristi u osnovnoj šemi kriptovanja za sistem. Drugi sloj, interfejs datoteke, kriptuje i dekriptuje datoteku. Treći sloj je rukovanje ključevima; on komponenti omogućava da 'potpiše' datoteku, proverava potpis i izračunava heš kôd za dobijanje pristupa datoteci. Na kraju, četvrti sloj proverava autentičnost. Ovaj sloj upravlja datotekom lozinki koja se čuva u kriptovanom obliku i od korisnika zahteva identifikaciju (korisničko ime) i lozinku.

Korisnici mogu da pristupe sistemu u različitim slojevima dizajna, zavisno od potreba izraženih u zahtevima. Na primer, ako korisnici ne moraju da znaju ništa o šemi kriptovanja, oni mogu da komuniciraju sa sistemom samo u poslednjem spoljašnjem sloju da bi promenili lozinku ili dali podatke za identifikaciju. S druge strane, ako je sistem projektovan za korišćenje u različitim zemljama (gde su zakonski dozvoljeni različiti algoritmi kriptovanja ili gde postoje ograničenja za izvoz određenih algoritama), tada neke klase korisnika mogu da imaju pristup „najunutrašnjijem" sloju i mogu da menjaju algoritam kriptovanja.

Slojevite arhitekture u velikoj meri koriste pojam apstrakcije. To jest, svaki sloj se može smatrati sledećim stepenom apstrakcije, a dizajneri mogu da koriste slojeve za razlaganje problema u niz sve apstraktnijih koraka. Štaviše, zbog ograničenja po kojima slojevi međusobno deluju, relativno je lako dodavati ili menjati sloj kada se za tim pojavi potreba; takve promene obično pogađaju samo dva susedna sloja. Iz istog razloga je ponovno korišćenje sloja relativno jednostavno, a izmene su potrebne jedino u susednim slojevima.

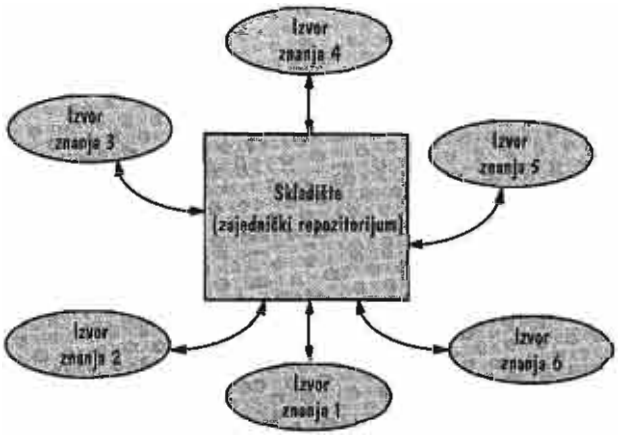
S druge strane, sistem u slojevima nije uvek lako strukturisati. Kada ispitujemo skup zahteva, slojevi apstrakcije nisu uvek očigledni. Čak i kada smo u stanju da napravimo slojeviti dizajn, dodatno koordinisanje među slojevima može da utiče na performanse sistema.

Skladišta

U skladištu postoje dve vrste komponenti. Centralno skladište podataka i kolekcija komponenti koje se nad njim izvršavaju da bi se informacije sačuvale, pronašle i ažurirale. Pri projektovanju skladišta značajno je odlučiti na koji način će te dve vrste komponenti međusobno da deluju. U **tradicionalnoj bazi podataka** transakcije u obliku ulaznog toka pokreću izvršavanje procesa. U takozvanom ***blekbord* sistemu gde su svi elementi repozitorijuma deljeni**, pokretanjem procesa upravlja centralno skladište.

Na slici 5.7 prikazan je tipičan sistem blekbord. On obično se sastoji od tri elementa: samog zajedničkog repozitorijuma (blekborda), izvora saznanja i kontrole. Sistem blekbord sadrži hijerarhiju statusnih podataka za rešavanje problema zavisnih od aplikacije koja se dizajnira. Izvori znanja su zasebne celine podataka, takođe zavisne od aplikacije, a međusobno deluju isključivo preko zajedničkog repozitorijuma. Izvršavanje zavisi od stanja zajedničkog repozitorijuma. Kako se ono menja, izvori znanja reaguju slično kao u stilu implicitnog pozivanja. Dizajn zajedničkog repozitorijuma se koristi u mnogim sistemima, pogotovo kod obrade signala i kod prepoznavanja oblika.

Mnogi drugi sistemi su takođe organizovani kao skladišta: biblioteke komponenti koje se mogu višekratno koristiti, velike baze podataka i mašine za pretraživanje. Prednost ovog tipa arhitekture je njegova otvorenost; predstavljanje podataka se često daje na uvid različitim isporučiocima tako da mogu dizajnirati alatke za pristupanje skladištu. S druge strane ta ista osobina predstavlja i manu, zato što deljeni podaci moraju biti u obliku koji je prihvatljiv za sve izvore podataka, čak i kada se ti izvori drastično razlikuju.



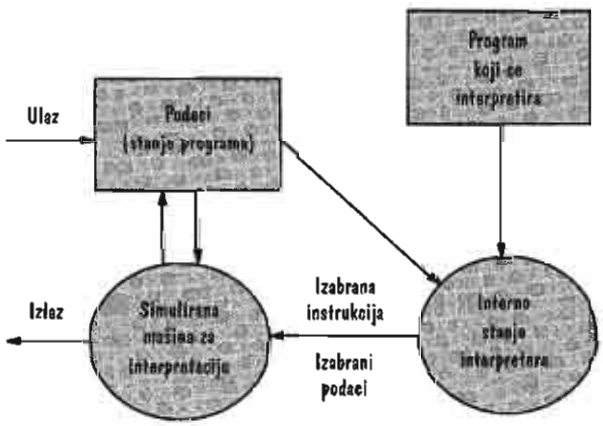
SLIKA 5.7 Tipičan sistem blekbord

Interpreteri

Interpreter uzima niz znakova koji nazivamo *pseudo kôd* i konvertuje ga u stvarni kôd koji se zatim izvršava. Na taj način, interpreter je mašina koja „interpretira" pseudo kôd tako da on postaje izvršiv. Interpreteri se mogu koristiti ne samo za konvertovanje programskih jezika u mašinski kôd već takođe i za konvertovanje bilo koje vrste kodiranja u eksplicitniji oblik ili u oblik koji se lakše koristi. U tu svrhu, interpreter se sastoji od četiri dela kao što je prikazano na slici 5.8:

1. Memorija koja sadrži pseudo kôd koji treba da se interpretira.
2. Mašina za interpretiranje koja konvertuje pseudo kôd i simulira program koji on predstavlja.
3. Trenutno stanje mašine za interpretiranje.
4. Trenutno stanje simuliranog programa.

SLIKA 5.8 Primer interpretera (iz knjige Shaw i Garlan, 1996)

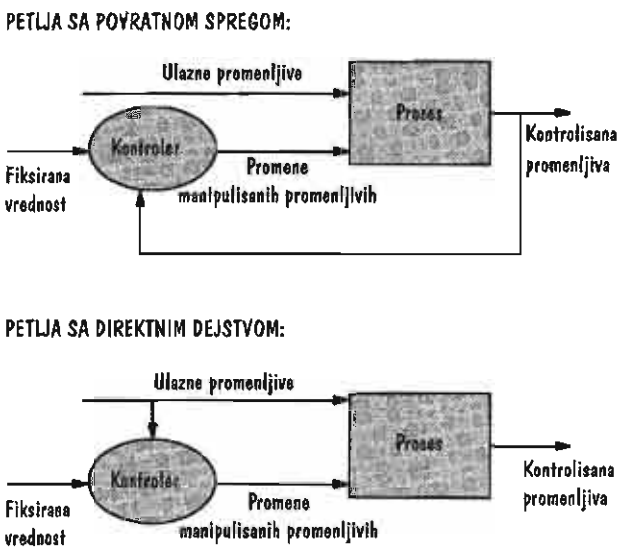


Mada je ovaj stil specifičan za određenu vrstu problema, uključujemo ga zato što se znatno razlikuje od ostalih stilova.

Kontrola procesa

Shaw i Garlan (1996) ističu da se sistemi za kontrolu procesa veoma razlikuju od dizajna zasnovanih na funkcijama ili na objektima koje karakterišu vrste komponenti u dizajnu. Sisteme za kontrolu procesa ne karakterišu samo vrste komponenti već takođe i odnosi koji među njima vladaju. Svrha sistema za kontrolu procesa je da se određene izlazne karakteristike procesa održe na određenim referentnim vrednostima ili blizu tih vrednosti koje se nazivaju *granične tačke*. Na primer, sistem može da kontroliše temperaturu šipki u nuklearnoj centrali; on nadgleda proces fisije, temperaturu i protok vode i obezbeđuje da uvek bude dovoljno vode kako bi šipke bile ispod kritične fiksirane temperature. Slično tome, vaš kućni sistem za grejanje i hlađenje koristi termostat za nadgledanje temperature vazduha, a zatim kontroliše peć i klima uređaj da bi temperatura vazduha bila unutar prihvatljivog raspona.

Najčešće korišćeni softverski kontrolni sistem je onaj sa zatvorenom petljom, koji se javlja u dva moguća oblika: sistem sa povratnom spregom i sistem sa direktnim dejstvom; oba su prikazana na slici 5.9. Sistem sa povratnom spregom meri kontrolnu promenljivu, kao što je temperatura, i podešava proces tako da kontrolisana promenljiva bude blizu fiksirane vrednosti ili tačno na njoj. Sistem sa direktnim dejstvom pokušava da predvidi buduće efekte na kontrolisanu promenljivu tako što meri ostale promenljive u procesu koje mogu da služe kao indikatori. Ovakav sistem je primereniji od sistema sa povratnom spregom ako se kontrolisana promenljiva neprihvatljivo sporo menja, kao i kada treba više vremena za promenu temperature vode ili vazduha.



SLIKA 5.9 Sistem sa povratnom spregom i sistem sa direktnim dejstvom (prilagođeno iz knjige Shaw i Garlan ,1996).

Pri projektovanju sistema za kontrolu procesa potrebno je obraditi mnoga pitanja: koje promenljive treba pratiti, kakve senzore upotrebiti, kako ih kalibrisati i kako vremenski rasporediti merenja i kontrole. Osim toga, Shaw i Garlan (1996) preporučuju da se u arhitekturi razdvoje tri dela kontrolne petlje:

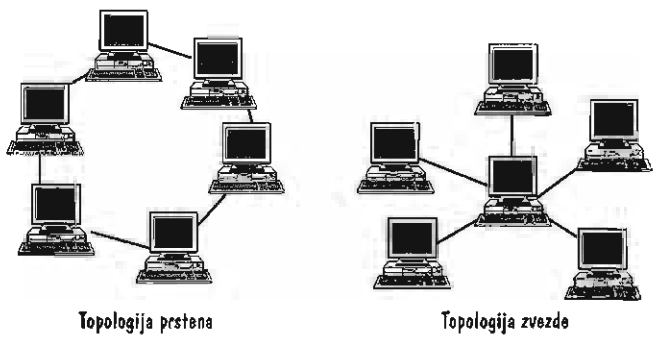
- Elementi za računanje koji odvajaju proces od politike kontrole. Definicija procesa trebalo bi da sadrži mehanizme za menjanje procesnih promenljivih. Kontrolni algoritam bi trebalo da objasni kako se donose odluke o trenutku i načinu vršenja tih promena.
- Elementi podataka, uključujući procesne promenljive (ulazne, kontrolisane i manipulisane), zadate vrednosti i senzore koji će se koristiti.
- Šema kontrolne petlje; otvorena ili zatvorena petlja, povratna sprega ili direktno dejstvo.

Prednost ovog arhitektonskog stila je u tome što je funkcionisanje odvojeno od reagovanja na spoljašnje poremećaje.

Ostali stilovi

Postoji nekoliko drugih arhitektonskih stilova koji se često koriste u određenim vrstama aplikacija. Arhitekture distribuiranih sistema bave se načinima međusobnog dejstva različitih grupa u sistemu. One se obično opisuju u smislu topologije njihove konfiguracije. Na primer, kao što je prikazano na slici 5.10, mogu se organizovati u prsten, gde se poruke šalju po prstenu iz jednog sistema u drugi, ili kao zvezda u kojoj centralni sistem prima sve poruke i usmerava ih na odgovarajuća odredišta.

Jedan popularan vid arhitekture distribuiranih sistema je sistem **klijent-server** u kome **klijentski** sistem zahteva akciju ili uslugu, a **serverski** sistem odgovara na zahtev (Dixon 1996). Najčešće, server ne zna unapred broj klijenata koji će mu pristupati kada sistem bude u funkciji; nisu mu poznati ni identiteti klijenata. Međutim, klijent zna za identitet servera i šalje serveru poruke koristeći pozive procedura. Prednost sistema klijent-server je u tome što korisnici dobijaju informaciju koja im je potrebna samo onda kada im ona stvarno treba. Štaviše, dizajn definiše i prezentacione detalje, tako da različiti klijenti mogu da pregledaju iste podatke na različite načine. Nedostatak ovih sistema je to što oni obično zahtevaju složenije upravljanje bezbednošću, upravljanje sistemom i razvoj aplikacija pa zato zahtevaju više resursa za implementaciju i podršku. U podsetniku 5.1 opisuje se sistem koji je implementiran kao klijent-server.



SLIKA 5.10 Topologije prstena i zvezde za distribuirane sisteme

PODSETNIK 5.1 KLIJENT-SERVER SISTEM SVETSKOG KUPA

Godine 1994. utakmice Svetskog kupa u fudbalu održavale su se u SAD. U toku samo jednog meseca 24 tima su odigrala 52 utakmice koje su privukle ogroman broj posetilaca i televizijsku publiku. Utakmice su se igrale u 9 različitih gradova koji se nalaze u 4 vremenske zone. Čim bi jedan tim dobio utakmicu, on je često morao za sledeću utakmicu da prede u drugi grad. Tokom ovog procesa rezultati svih utakmica su se evidentirali i saopštavali štampi i navijačima. U isto vreme, da bi se sprečilo da dođe do nasilja među navijačima, organizatori su podelili i pratili oko 20.000 propusnica.

Sistem je zahtevao i centralnu kontrolu i distribuciju funkcija. Na primer, sistem je pristupao centralnim informacijama o svim igračima. Nakon ključne utakmice, sistem je bio u stanju da prikaže istorijske informacije (fotografije, video i tekst) o igračima u toj utakmici. Zato se činilo da je najpogodnija klijent-server arhitektura.

Izgrađen je sistem koji je imao centralnu bazu podataka koja se nalazila u Teksasu i služila je za upravljanje prodajom karata, za bezbednost, za usluge izveštavanja i linkove ka Internetu. Ovaj server je takođe izračunavao statistike igre, pružao istorijske informacije, fotografije za službu bezbednosti i video snimke. Klijenti su se izvršavali na 1.600 radnih stanica Sun od kojih su neke služile za podršku administrativnom osoblju i štampi (Dixon 1996).

U SAD je postojao pritisak da se razvijaju **arhitekture specifične za domen** koje bi koristile prednosti određenih domena primene, kao što su avionska industrija ili automatizovana proizvodnja. Ove arhitekture koriste prednosti jednoobraznosti u domenima primene tako da neke bazične pretpostavke i odnose nije potrebno ni obrazlagati. Dugoročni cilj bio je da se naprave rešenja koja se u mnogim slučajevima mogu direktno primeniti, ili se njihove komponente mogu ponovo iskoristiti.

Neke arhitekture su heterogene. To jest, kombinuju poželjne aspekte iz više stilova tako što koriste najprimereniji stil za svaki podsistem. Na primer, sistem sa cevima i filtrima može biti najprimereniji na prvom nivou raščlanjivanja, ali se svaki filtar projektuje i implementira pomoću sasvim različitih arhitektura. Shaw i Garlan (1996) daju još neke primere kombinovanja različitih arhitektura.

5.4 PITANJA U KREIRANJU DIZAJNA

Sheila Brady, direktor razvoja za operativni sistem Appleovog Systema 7, izvestila je na konferenciji o metodima 1995. da je dozvolila različitim projektantima da koriste tehnike koje žele pod uslovom da dokumentacija bude razumljiva ostalim projektantima. Njen pristup uzima u obzir činjenicu da se prilikom kreiranja dizajna postavljaju mnoga pitanja: šta je najbolje za aplikaciju, kako se u tome oseća projektant i šta je primereno za celokupnu arhitekturu. Prema tome, ne postoji stil ili metod koji bi bio najbolji za sve situacije, a zbog mogućnosti pojave novih stilova i tehnika moramo biti u stanju da ih poredimo sa onima koji se trenutno primenjuju. U ovom odeljku istražujemo nekoliko pitanja kojima projektanti moraju da se bave, ne samo prilikom biranja odgovarajućeg stila, već i prilikom pravljenja samih detalja vezanih za dizajn.

Modularnost i nivoi apstrakcije

Već smo napomenuli da je modularnost jedna od karakteristika dobrog dizajna. U modularnom dizajnu komponente imaju jasno definisan ulaz i izlaz i svaka komponenta ima jasno izraženu svrhu. Prema tome, lako je ispitati svaku komponentu odvojeno od ostalih i utvrditi da li ona implementira zahtevane zadatke. Štaviše, modularne komponente se organizuju u hijerarhiju koja je rezultat raščlanjivanja ili apstrahovanja, tako da se sistem može ispitati nivo po nivo. Iz tih razloga pokušavamo da projektujemo sistem tako da on bude što modularniji.

U raščlanjivanju dizajna, komponente na svakom nivou usavršavaju komponente iz prethodnog nivoa. Na svakom nižem nivou nalazimo sve više detalja o svakoj komponenti. Prema tome, najviši nivo smatramo najapstraktnijim, pa kažemo da su komponente uređene po **nivoima apstrakcije**. Nivoi apstrakcije nam pomažu da shvatimo problem kojim se sistem bavi i rešenje koje dizajn predlaže. Ispitivanjem nivoa odozgo nadole prvo se rešavaju apstraktniji problemi, a rešenje se gradi generisanjem detaljnijih opisa. Na izvestan način, apstraktniji viši nivoi kriju detalje funkcionalnih komponenti i podataka. U podsetniku 5.2 prikazano je kako nivoi apstrakcije unapređuju dizajn.

Modularnost na sličan način sakriva detalje. Parnas (1972) predlaže da komponente jedna od druge kriju unutrašnje detalje i obradu. Prednost ovakvog **sakrivanja informacija** je da svaka komponenta krije od ostalih dizajnersku odluku. Na taj način, ako postoji verovatnoća da se pojedinačne odluke vezane za dizajn promene, dizajn celine može da ostane netaknut dok se menjaju jedino komponente.

Kombinovanjem modularnih komponenti i više nivoa apstrakcije, dobijamo više različitih pogleda na sistem. Komponente najvišeg nivoa omogućavaju nam da vidimo rešenje kao celinu, a kriju detalje koji bi nam inače skretali pažnju. U funkcionalnom raščlanjivanju takav prikaz može da nam pokaže glavne funkcije koje će sistem izvršavati. U objektno orijentisanom dizajnu ovaj prikaz ilustruje apstraktne tipove, pa možemo da vidimo relacije različitih sistemskih objekata bez potrebe da razmatramo svaki pojedinačni primerak. Kada nam zatreba više detalja o nekom delu sistema, prelazimo na niže nivoe apstrakcije.

Otvarajući nam mogućnost da prelazimo na niže nivoe kada nam je potrebno da potražimo više detalja, modularnost obezbeđuje neophodnu fleksibilnost da bismo shvatili šta sistem treba da radi, pratili tok podataka i funkcija kroz ceo sistem, i da uočili složenije tačke. Nasuprot raširenom shvatanju, rastavljanjem problema na delove nećemo uvek od jednog složenog problema napraviti skup jednostavnijih problema. Međutim, modularnost omogućava da se izoluju oni delovi problema koje je najteže savladati; nivoi apstrakcije omogućavaju da se problem postepeno shvati na sve detaljnijim nivoima. Kada problem ovako izolujemo, neće nas zbunjivati niti nam skretati pažnju funkcije i podaci koji sa tim problemom nisu u vezi.

Dodatna prednost modularnosti je mogućnost da se različite komponente projektuju na različite načine. Na primer, korisnički interfejs može da se projektuje objektno i sa prototipovima, a dizajn bezbednosnih mehanizama može da sadrži dijagrame promena stanja.

PODSETNIK 5.2 UPOTREBA APSTRAKCIJE

Apstrakciju možemo da koristimo za proširivanje dizajna. Pretpostavimo da je jedna od funkcija sistema uređivanje elemenata u listi L. Početni opis dizajna je:

```
Preurediti L u nepadajućem redu
```

Sledeći nivo apstrakcije može da bude konkretan algoritam:

```
DO WHILE i je između 1 i (dužina od L) -1;
  Postaviti LOW na indeks najmanje vrednosti u L(i),..., L(dužina od L)
  Zameniti L(i) i L(LOW)
ENDDO
```

Algoritam nam daje mnogo više informacija. Obaveštava nas o proceduri koja će se koristiti za preuređivanje liste L. Međutim, može se ići i dalje u detalje. Treći i konačni algoritam opisuje tačno kako će se vršiti preuređivanje:

```
DO WHILE i je između 1 i (dužina od L) -1;
  Neka LOW bude trenutno vrednost i
  DO WHILE j je između i+1 i (dužina od L) -1;
    IF L(LOW) je veće od L(j)
      THEN neka LOW bude trenutno vrednost j
    ENDIF
  ENDDO
  Neka TEMP bude L(LOW)
  Neka L(LOW) bude L(i)
  Neka L(i) bude TEMP
ENDDO
```

Svaki nivo apstrakcije ima svoju svrhu. Ako nas zanima samo kako L izgleda pre i posle preuređivanja, dovoljna nam je prva apstrakcija. Pošto sadrži više detalja, drugi algoritam daje prikaz procedure kojom se vrši preuređivanje. Ako nas zanima samo brzina algoritma, drugi nivo apstrakcije odgovara. Međutim, ako pišemo kôd za operaciju preuređivanja, u trećem nivou apstrakcije vidimo tačno šta treba uraditi; potrebno je vrlo malo dodatnih informacija.

Pogledajte prethodna tri preuređivanja. Kada bismo vam pokazali samo treći nivo, možda ne biste odmah uočili da procedura opisuje preuređivanje. Na prvom nivou je priroda procedure očigledna. Treći nivo vam odvlači pažnju od same prirode procedure. Prema tome, informacija koja krije apstrakciju usredsređuje našu pažnju na svrhu komponente ili algoritma.

Apstrakcija je osnova objektno orijentisanog dizajna i takođe podržava skrivanje informacija. Na primer, umesto da direktno stavljamo u stek, mi definišemo objekt po imenu *stek* i metode koje se nad njim izvršavaju, *push* i *pop*. Za manipulisanje elementima u steku koristimo objekat i njegove metode, a ne sam stek kao strukturu podataka. Možemo takođe da definišemo upite koji će nam obezbediti informacije o steku – da li je pun ili je prazan i šta je na vrhu – bez promene njegovog stanja.

Saradnja na dizajnu

U većini projekata dizajn ne pravi samo jedan čovek, već na dizajnu sarađuje ceo tim. Često se različiti delovi projekta dodeljuju različitim ljudima. Tim mora da reši niz pitanja, počevši od odluke koji član tima najbolje odgovara za određeni aspekt sistema, preko dokumentovanja dizajna tako da svaki član shvata ostale delove, pa do koordinisanja komponenti dizajna tako da one dobro funkcionišu kao celina. Projektni tim mora da bude svestan razloga za podelu projekta (podsetnik 5.3) i da upotrebi snage tima na njihovom rešavanju.

Jedan od glavnih problema u stvaranji na dizajnu jeste savlađivanje razlika u ličnom iskustvu, shvatanjima i sklonostima. Drugi problem je u tome što se ljudi ponekad u grupi ponašaju drugačije nego kao pojedinci. Na primer, japanski projektanti softvera ponekad u grupi neće da iskažu vlastito mišljenje zato što se timski rad više ceni nego individualni. Harmonija je veoma važna i mlađi saradnici u Japanu priklanjaju se shvatanjima starijih kolega (Ishii 1990). Watson, Ho i Raman (1994) došli su do sličnih zaključaka kada su poredili ponašanje grupa na sastancima u SAD sa ponašanjem u Singapuru. Za američke grupe bila je važna paralelna komunikacija i anonimna razmena informacija, za razliku od singapurskih grupa koje su prednost davale harmoniji. U ovakvim slučajevima možda je poželjno da se projektuje pomoću alatke za grupni rad kako bi se sačuvala anonimnost. Zaista, Valacich i dr. (1992) izveštavaju da ovakva vrsta čuvanja anonimnosti može da unapredi sveukupne performanse grupe. Međutim, postoje i mane. U nekim grupama anonimnost može kod nekih pojedinaca da dovede do manjeg osećanja odgovornosti. Prema tome, važno je međusobno delovanje članova grupe posmatrati u kulturološkom i etičkom kontekstu.

PODSETNIK 5.3 UZROCI NEUSPEHA DIZAJNA

Guindon, Krasner i Curtis (1987) proučavali su ponašanje dizajnera na 19 projekata kako bi utvrdili razloge zbog kojih proces projektovanja dovodi do neuspeha. Utvrdili su 3 klase neuspeha: nedostatak znanja, kognitivna ograničenja i kombinacije ta dva razloga.

Glavne vrste neuspeha procesa bile su:

- nepostojanje specijalizovanih semantskih dizajna;
- nepostojanje metaseme za proces dizajniranja, što za posledicu ima loše alociranje resursa za različite projektne aktivnosti;
- pogrešno određivanje prioriteta, što dovodi do lošeg izbora alternativnih rešenja;
- teškoće da se rasprave sva konstatovana ili izvedena ograničenja za definisanje rešenja;
- teškoća da se mentalno simuliraju situacije sa mnogo koraka ili mnogo slučajeva za testiranje;
- teškoće u evidentiranju i vraćanju na potprobleme čije rešavanje je odloženo;
- teškoće u proširivanju ili objedinjavanju rešenja pojedinačnih elemenata u kompletno rešenje.

Kako industrija softvera teži da smanji troškove i poveća produktivnost tako što razvija softver za saradničke grupe raspršene po celom svetu, povećava se značaj razumevanja ponašanja u grupi. Yourdon (1994) prepoznaje četiri faze u ovoj ~~vrsti~~ distribuiranog razvoja.

1. U prvoj fazi projekat se izvršava na jednom mestu sa projektantima iz raznih zemalja.
2. U drugoj fazi analitičari na licu mesta utvrđuju sistemske zahteve. Zatim se zahtevi saopštavaju grupama dizajnera i programera na drugim lokacijama gde će oni nastaviti razvoj.
3. U trećoj fazi projektanti na različitim mestima grade generičke proizvode i komponente koje se koriste u celom svetu.
4. U četvrtoj fazi projektanti na različitim mestima grade proizvode u kojima se koriste njihova pojedinačna specijalistička znanja.

Primetićete da je ovaj model suprotan nalazima koje smo citirali u prethodnom odeljku, gde dizajneri naizmenično rade sa zahtevima, sa analitičarima, sa izvršiocima na testiranju i na kodiranju, kako bi unapredili svoja saznanja. Swartout i Balzer (1982) takođe ukazuju na međusobna dejstva specifikacije i dizajna. Kada se implementira Yourdonov model, problemi mogu da se jave u drugoj fazi u kojoj putevi komunikacije moraju ostati otvoreni da bi se omogućio iterativni proces dizajniranja.

Za unapređenje komunikacije mogu se koristiti povratne informacije u obliku beleški, prototipova, crteža itd. Međutim, eksplicitno predstavljanje zahteva i dizajna mora biti nedvosmisleno i mora obuhvatiti sve pretpostavke o planiranom radu sistema. Polanyi (1966) napominje da se namere ne mogu potpuno izraziti ni na jednom jeziku; neke nijanse nisu očigledne. Prema tome, komunikacija u grupi može da bude neuspešna ako primalac tumači informaciju u smislu svojih shvatanja i konteksta. Na primer, u ličnom kontaktu veliku količinu informacija prenosimo gestikulacijom i izrazom lica; ova vrsta informacija se gubi prilikom elektronske saradnje (Krauss i Fussell 1991).

Ta teškoća se usložnjava kada se komunikacija obavlja na više jezika. Na primer, na italijanskom jeziku postoji više od pet stotina reči za testeninu, a Beduini imaju više od 160 reči za „kamilu". Izuzetno je teško prevesti nijanse tih različitih reči. Winograd i Flores (1986) čak tvrde da je potpuni prevod iz jednog prirodnog jezika u drugi nemoguć zato što se semantika prirodnog jezika ne može formalno i potpuno definisati. Prema tome, glavni izazov u proizvodnji dobrog softverskog dizajna je postizanje zajedničkog razumevanja među grupama ljudi koji možda posmatraju sistem i njegovo okruženje na veoma različite načine. Taj izazov ne potiče samo od „složenosti tehničkog problema, već [takođe] nastaje zbog socijalne interakcije dok korisnici i projektanti sistema uče da formulišu, razvijaju i izražavaju svoje ideje i vizije" (Greenbaum i Kyng 1991).

Da bismo se uhvatili u koštac sa ovim pitanjima, mora nam biti jasno da je dizajn softvera i saradnički i iterativni proces. Kada gradimo softverski sistem, to nije samo izgradnja proizvoda; mi gradimo takođe i zajedničko saznanje o kupcima, korisnicima, domenu primene, okruženju i još o mnogo čemu. Naši napori moraju biti usredsređeni na što bolje upoznavanje sa svim ovim aspektima.

Dizajn korisničkog interfejsa

Korisnički interfejsi mogu da budu nezgodni za projektovanje zato što razni ljudi imaju različite stilove percepcije, razumevanja i rada. Na primer, jedan korisnik može da upotrebljava paket za obradu teksta koristeći tastaturu dok se drugi uglavnom oslanja na miša. Slično tome, korisnici obavljaju akcije različitim redosledom; skloniji su različitim komandama, brojčanicima i prozorima; više ili manje koriste ekrane za pomoć i priručnike.

Marcus (1993) opisuje mnoga pitanja u vezi sa dizajnom interfejsa. On ističe da bi interfejs trebalo da ima u vidu nekoliko ključnih elemenata:

- **metafore**: osnovni izrazi, slike i koncepti koji mogu da se prepoznaju i nauče;
- **mentalni model**: organizacija i predstavljanje podataka, funkcija, zadataka i uloga;
- **navigaciona pravila za model**: način kretanja po podacima, funkcijama, aktivnostima i ulogama;
- **izgled**: karakteristike izgleda sistema koje korisniku prenose informacije;
- **utisak**: tehnike interakcije koje pružaju korisniku privlačan doživljaj.

Cilj ovih elemenata i celog korisničkog interfejsa jeste da se „pomogne korisnicima da brzo pristupaju sadržaju složenih sistema bez gubitka razumevanja dok se kreću kroz informacije" (Marcus 1993).

Korisnički interfejs može da obuhvati različite tehnologije: agente, hipertekst, zvuk, trodimenzionalne prikaze, video i virtualnu stvarnost. Te tehnologije mogu zatim da se implementiraju pomoću različitih hardverskih konfiguracija uključujući tastature, grafičke monitore, elektronske olovke i naočari za virtualnu stvarnost. Ali, da bi se dizajnirali prijatni i efikasni interfejsi, moramo da razmotrimo dva ključna pitanja: kulturu i „ukuse".

Pitanja kulture. U četvrtom poglavlju smo videli da su prototipovi korisni da bi korisnici i kupci lakše odlučili kakav interfejs da traže. Na isti način prototipovi mogu da se koriste u fazi dizajna da bi se testiralo šta ko voli i odredilo koje vrste interfejsa su izvodljive i koje zadovoljavaju zahteve za performansom i pouzdanošću. Ali, prototipovi za utvrđivanje preferencija moraju uzeti u obzir i kulturološke razlike i dinamiku grupa u populaciji očekivanih korisnika. Pošto se naš softver koristi na celom svetu, moramo da imamo na umu verovanja, vrednosti, norme, tradicije, običaje i mitove onih ljudi koji će koristiti naš sistem. Neki dizajneri interfejsa korisnicima nude izbor jezika u svojim menijima, ikonama ili numeričkim formatima (kao što je valuta ili razmera). Međutim, Nakakoji (1994) ističe da jezički prevod nije dovoljan za prelazak u drugu kulturu. Štaviše, mnoge grupe korisnika se sastoje od ljudi iz različitih kultura, pa ne možemo pretpostaviti da ćemo prihvatanje i pravilno korišćenje sistema obezbediti tako što ćemo napraviti po jedan sistem za svaku od glavnih kultura. Prema tome, kao što smo u poglavlju 2 videli da različiti radnici imaju različite stilove i preferencije, kulture korisnika dodaju još jedan sloj složenosti u zadatak dizajniranja dobrih i korisnih interfejsa.

Da bi naši sistemi bili prilagođeni različitim kulturama, možemo da projektujemo interfejse u dva koraka. Prvo treba da eliminišemo specifične kulturne reference ili predubeđenja da bi interfejsi bili što više „međunarodni". Jones i dr. (1991) nude smernice za smanjivanje kulturoloških predubeđenja u mnogim aspektima naših sistema, uključujući priručnike, poruke, oznake, legende, ikone, grafikone i zvukove. Na primer, neki

tekstovi postaju značajno duži kada se prevedu iz engleskog jezika u neki drugi i zato ne bi trebalo praviti pomoćne prozore sa fiksnim dimenzijama. Slično tome, tekstualni opisi i brojevi trebalo bi da se čuvaju zasebno da bi se lakše menjali.

U drugom koraku se uzima dizajn bez specifičnih kulturnih obeležja i prilagođava se kulturi u kojoj će se softver koristiti. Na primer, u različitim kulturama se ikonama i bojama pridaje posebno značenje. U Engleskoj je grimizna boja kraljevska, a u Japanu označava dostojanstvo i plemenitost. Ali, u staroj Grčkoj je grimizno bilo simbol smrti i zla (Fukuda 1994)! Zato grimizna boja neće biti dobar izbor za međunarodni dizajn. Slično tome, moramo da uzmemo u obzir razlike u očekivanom toku objekata na ekranu: u nekim kulturama se čita zdesna nalevo, a u drugim sleva nadesno (Russo i Boor 1993). Na isti način, funkcionalnost sistema se ponekad neočekivano poremeti zbog kulture korisnika (Nakakoji 1994). Na primer, alatka za izgradnju konsenzusa neće imati mnogo svrhe u Japanu gde glavne odluke donosi najstariji član grupe. Iz tog razloga, moramo veoma pažljivo testirati naš dizajn na pretpostavljenim korisnicima da bismo bili sigurni da će se sistem koristiti onako kako je to u dizajnu predviđeno.

Važno je imati na umu da kultura ne zavisi samo od nacije već i od regiona, pola, zanimanja, starosti ili korporacije. Potencijalni korisnici jednog sistema mogu da ne pripadaju istoj kulturi kao projektant, pa je opasno ako projektanti pretpostavljaju da znaju šta korisnici žele. Slično tome, rukovodioci imaju drugačije norme, predstave i način rada od projektanata koje nadgledaju; najbolje je da se interfejs testira u kulturi u kojoj će se koristiti (Bodker i Pedersen 1991). Konkretno, čest običaj da se projektanti ili rukovodioci „igraju" sa različitim interfejsima da bi im to pomoglo u donošenju odluka vezanih za dizajn možda i nije baš najbolji.

Preferencije korisnika. Neki aspekti dizajna zavise od preferencija korisnika, bilo samostalnih bilo kao članova radne grupe. Na primer, Marcus (1993) je predložio tri interfejsa od kojih je svaki skrojen za drugačiju publiku. Interfejs 1 bio je namenjen evropskim odraslim muškim intelektualcima koji koriste engleski jezik. Za interfejs 2 se smatralo da je dobro prilagođen belim američkim ženama, a interfejs 3 potrošačima koji koriste engleski, ali više vole međunarodni dizajn. Marcus je u dizajnu za evropske odrasle muške intelektualce koji koriste engleski jezik pretpostavio da će im se dopasti „pristojan tekst, odmerena gustina informacija i klasični pristup u biranju fontova (na primer, serifni slogovi simetrično raspoređeni po osama, slično elegantnim bronzanim evropskim tablama na zgradama)". Pretpostavio je da će se belim amerikankama dopasti „detaljnija prezentacija, zaobljeni oblici i odsustvo brutalnijih izraza kao što su Kill, Trash [i] Abort koji se dopadaju muškim softverskim inženjerima". Na kraju, pretpostavio je da će se korisnicima koji vole kulturološku raznovrsnost dopasti sažetija terminologija, zgusnutije informacije i jednostavna jasna tipografija (Marcus 1993).

Teasley i drugi (1994) testirali su Marcusova tri korisnička interfejsa da bi utvrdili da li zaista odgovaraju preferencijama kako je Marcus predviđao. Na 54 američka građana i 35 van Amerike (43 muškarca i 46 žena) utvrdili su da se svim korisnicima najviše dopao interfejs 1, a najmanje interfejs 2. Međunarodni korisnici su birali interfejs 1 ili 2, a žene su uglavnom birale interfejs 1. Štaviše, kod pogađanja najverovatnije publike za svaki interfejs, 40 od ukupno 89 ispitanika je smatralo da je interfejs 2 namenjen Evropljanima, a 41 je smatrao da je interfejs 1 za žene.

Prema tome, izgleda da se nijedan univerzalan interfejs ne može primeniti ni na jednu kulturu i da može biti teško definisati smernice za dizajn koje bi nam garantovale

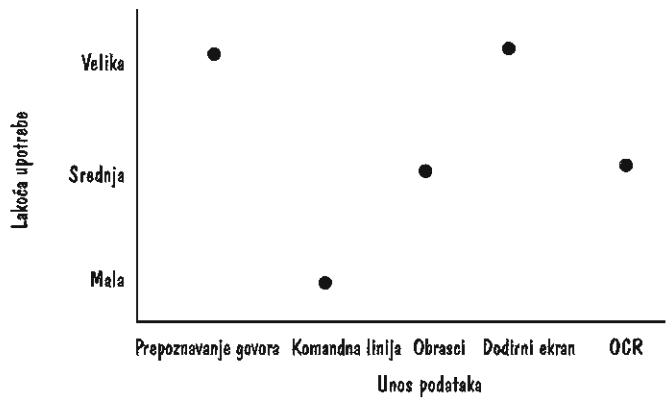
da će se interfejsi naših sistema dopasti korisnicima. Rezultati ovih i još nekih ispitivanja naglašavaju značaj ispitivanja prototipa na konkretnoj ciljnoj publici za sistem koji projektujemo.

Smernice za određivanje karakteristika korisničkog interfejsa. Biranje karakteristika za dizajn korisničkog interfejsa zahteva mnogo međusobnih ustupaka. Lane (u knjizi Shaw i Garlan 1996) predlaže da razmotrimo pitanja dizajna u smislu prostora. To jest, svaki nedostatak reflektuje bar dve dimenzije izbora: na primer, moramo pronaći balans između performansi i funkcionalnosti ili između lakoće korišćenja i bezbednosti. Svoj izbor možemo da posmatramo na grafikonu kao na slici 5.11, gde odlučujemo koja tehnika za unos podataka može da obezbedi željenu lakoću upotrebe. U tom primeru upisivanje komandi na komandnu liniju daleko je teže nego popunjavanje obrasca ili jednostavno korišćenje dodirnog ekrana.

U tabeli 5.1 navedene su neke dimenzije koje Lane preporučuje za odlučivanje o pitanjima dizajna. Svaku stavku u ovoj tabeli trebalo bi detaljno ispitati da bi se odredilo kakva će biti struktura korisničkog interfejsa i kako će on funkcionisati. Na primer, ako kao osnovno ponašanje interfejsa izaberemo biranje iz menija, onda korisnik može više puta da bira iz alternativnih grupa koje sistem može da prikaže. Druga mogućnost je da korisnik popuni obrazac tako što upiše tekst u kojem se nalaze vrednosti određenih promenljivih. Slično tome, interfejs komandnog jezika može da koristi veštački simbolički jezik, a može i prirodni. Na kraju, neki sistemi dozvoljavaju korisniku direktno manipulisanje podacima.

Konkurentnost

U mnogim sistemima se akcije moraju izvršavati konkurentno, a ne sekvencijalno. Na primer, sistem za praćenje kvaliteta vode može da sadrži na desetine senzora koji izveštavaju o karakteristikama kao što su temperatura, brzina, kiselost i hemijski sastav. Senzori mogu da izveštavaju samo kada se vrednost promeni, a sistem mora da izračuna kvalitet vode na osnovu trenutnih podataka iz senzora. Sistem mora da dozvoli konkurentno prihvatanje podataka iz senzora.



SLIKA 5.11 Tipičan prostor dizajna

Tabela 5.1 Pitanja koja treba razmotriti prilikom analiziranja karakteristika (Lane, u knjizi Shaw i Garlan 1996)

Funkcionalne dimenzije	Strukturne dimenzije
Obrada spoljašnjih događaja • nema spoljašnjih događaja; • obrada događaja dok se čeka ulaz; • spoljašnji događaji unapred najavljuju korisničke komande.	Nivo apstrakcije interfejsa aplikacije • monolitni program; • apstraktni uređaji; • komplet alatki; • rukovalac interakcijom sa fiksiranim tipovima podataka; • rukovalac interakcijom sa promenljivim tipovima podataka; • prošireni rukovalac interakcijom.
Prilagodljivost korisniku • velika; • srednja; • mala.	Promenljivost apstraktnog uređaja • idealni uređaj; • parametrizovani uređaj; • uređaj sa promenljivim operacijama; • ad hoc uređaji.
Prilagodljivost korisničkog interfejsa različitim uređajima • nikakva; • lokalne promene ponašanja; • globalne promene ponašanja; • promena semantike aplikacije.	Notacija za definisanje korisničkog interfejsa • implicitno u zajedničkom kodu korisničkog interfejsa; • implicitno u aplikacionom kodu; • spoljašnja deklarativna notacija; • spoljašnja proceduralna notacija; • unutrašnja deklarativna notacija; • unutrašnja proceduralna notacija.
Organizacija računarskog sistema • jednoprocesna; • višeprocesna; • distribuirana obrada.	Osnova za komunikaciju • događaji; • čisto stanje; • stanje sa savetima; • stanje + događaji.
Klasa osnovnog interfejsa • biranje iz menija; • popunjavanje obrasca; • komandni jezik; • prirodni jezik; • direktno manipulisanje.	Kontrola mehanizama niti • nikakva; • standardni procesi; • laki procesi; • nenajavljeni procesi; • rukovaoci događajima; • rutine za obradu prekida.
Prenosivost aplikacije na različite stilove korisničkog interfejsa • velika; • srednja; • mala.	

Sekvencijalni sistemi obično koriste jedan tok izvršavanja za kontrolu događaja, dok konkurentni sistemi moraju da imaju složeniji dizajn. Jedan od najvećih problema kod konkurentnih sistema je potreba da se obezbedi konzistentnost podataka koji su zajednički za komponente koje se istovremeno izvršavaju. Na primer, pretpostavimo da imamo stek *X* i da dve komponente, *komponenta_1* i *komponenta_2* koriste elemente iz *X*. Svaka komponenta može da proveri da li je *X* prazno, pa ako nije da uzme sledeći element iz steka. Međutim, razmotrimo sledeći redosled događaja:

1. *Komponenta_1* pita da li je *X* prazan.

2. *Komponenta_1* se informiše da *X* nije prazan.
3. *Komponenta_2* pita da li je *X* prazan.
4. *Komponenta_2* se informiše da *X* nije prazan.
5. *Komponenta_1* uzima prvi element iz steka, ali on je ujedno i poslednji.
6. *Komponenta_2* pokušava da uzme sledeći element iz steka, ali *X* je prazan i sistem ulazi u nedozvoljeno stanje.

Jedan od načina da se kontroliše konkurentnost jeste da se propiše količina vremena za izvršavanje bilo koje aktivnosti. Na taj način, pažljivim planiranjem vremena može se obezbediti da jedna akcija ne ometa drugu. Međutim, planiranje vremena nije uvek pod sistemskom kontrolom, pogotovo kod sistema za rad u realnom vremenu koji reaguju na spoljašnje događaje, kao što je promena senzorskih podataka.

Za rešavanje ove vrste problema koristimo tehnike za sinhronizovanje konkurentnih procesa. **Sinhronizovanje** je metod kojim se omogućava uporedno izvršavanje dve aktivnosti tako da jedna ne ometa drugu.

Međusobno isključivanje je popularan način za sinhronizovanje procesa; njime se obezbeđuje da kad jedan proces pristupi podatku nijedan drugi proces ne sme da utiče na taj podatak. U našem gornjem primeru sa stekom, princip međusobnog isključivanja može se primeniti u sledećoj modifikaciji:

1. *Komponenta_1* pita da li je *X* prazan.
2. *Komponenta_1* se informiše da *X* nije prazan.
3. *Komponenta_2* pita da li je *X* prazan.
4. *Komponenta_2* se informiše da *X* nije prazan.
5. *Komponenta_1* uzima prvi element iz steka koji je ujedno i poslednji, i tada zaključava *X*.
6. *Komponenta_2* pokušava da uzme sledeći element iz steka, ali dobija obaveštenje da je *X* zaključan.
7. Neka druga komponenta doda jedan element na *X*.
8. *Komponenta_2* se obaveštava da je *X* otključan (obaveštenje stiže od nekog spoljašnjeg procesa ili tako što *komponenta_2* ponovi pokušaj).
9. *Komponenta_2* uzima prvi element iz steka.

U opštem slučaju, ako dve operacije mogu da utiču na stanje zajedničkog objekta, trebalo bi da se one izvršavaju po šemi međusobnog isključivanja. Slično tome, ako jedna operacija ispituje vrednost stanja jednog objekta, tada bi taj objekt trebalo da se zaključa da se to stanje ne bi promenilo od trenutka ispitivanja do trenutka kada se izvrši akcija izabrana na osnovu vrednosti dobijene u fazi ispitivanja.

Prioritet procesa ili komponente takođe može da se koristi u rešavanju konflikata konkurentnog rada. Ko ima veći prioritet može da „pobedi" u borbi između dva procesa ili komponente i da drugoj zaključa pristup dok se ne dovrši aktivnost sa većim prioritetom.

Vremenski raspored, sinhronizacija i prioriteti procesa daju prednost ispravnosti nad determinizmom zato što mehanizmi zaključavanja i vremenskog raspoređivanja zavise od redosleda kojim zahtevi stižu. Idealno bi bilo da možemo da projektujemo sisteme tako da budu ispravni, bez obzira na vremenski raspored zahteva. Srećom, postoje dva načina da se to postigne: preko monitora i zaštitnika.

Monitori. Monitor je apstraktni objekt ili komponenta koja kontroliše međusobno isključivanje određenih procesa. Da biste videli kako monitor funkcioniše, pretpostavite da je *M* monitor datog procesa *P*. Komponenta *A* želi da izvrši *P*, zato mora da prođe kroz *M* kako bi dobila pristup do *P*. Dok *A* izvršava proces *P*, komponenta *B* želi takođe da izvrši *P*. *M* suspenduje *B* sve dok *A* ne završi rad sa *P*, a zatim *M* aktivira *B* i odobrava mu da pristupi procesu *P*. Da bi se izbegli problemi koji su svojstveni vremenskom rasporedu kakav je upravo opisan, uz monitor obično postoji i komponenta za proveru stanja koja proverava da li su ispunjeni svi uslovi za pozivanje zahtevanog procesa. U slučaju steka, komponenta za proveru stanja obezbeđuje da stek ne bude prazan i komponenti ne dozvoljava pristup ako je stek prazan. Često je korisno da se prijavi izuzetak ako zahtevani uslovi nisu ispunjeni.

Zaštitnici. Zaštitnik je zadatak koji se uvek izvršava: njegova jedina svrha je kontrola pristupa enkapsuliranom resursu (tj. strukturi podataka ili procesu). Kao i kod monitora, zaštitnik takođe sadrži komponentu za proveru stanja koja mu pomaže u donošenju odluka za kontrolu pristupa. Ako je *G* zaštitnik resursa *R*, zadatak *T* mora da izda zahtev prema *G* da bi koristio *R*; međusobno dejstvo *T* i *G* naziva se randevu. *G* trenutno suspenduje *T* sve dok proveravač stanja ne prijavi da su uslovi zadovoljeni; tada *G* reaguje na zahtev. Prema tome, *G* neprestano reaguje na punovažne zahteve zadataka. Iz aspekta *T*, ponašanje *R* je sakriveno, a *G* je nedeterminističko. Ispravnost dizajna može se proveriti nezavisno od vremenskog rasporeda, a tek kada budemo implementirali dizajn, moraćemo da odlučimo kako da postupimo sa neodređenošću.

Šabloni dizajna i višekratno korišćenje

Često dizajniramo i gradimo sisteme koji su u nekom pogledu slični sistemima koje smo ranije napravili. Na primer, kada gradimo niz kontrolnih sistema za postrojenja koja prečišćavaju otpadne vode, svaki sistem predstavlja unapređenje ili zamenu prethodnog. Ili, na primer, pravimo niz aplikacija sa sličnim funkcijama koje treba da se izvršavaju u različitim okruženjima; komercijalne proizvode za široku prodaju kao što su sistemi za upravljanje bazama podataka ili kalendarski sistemi koji se obično projektuju na taj način. Mi želimo da iskoristimo zajedničke elemente sistema da ne bismo svaki od njih razvijali „od nule".

Popularan način za pronalaženje zajedničkih elemenata jeste da se nađu šablonski dizajni. Zatim kada pravimo sledeći sličan sistem, možemo ponovo da iskoristimo šablone kao i kôd, testove i dokumente koji su za njih vezani.

> **Šablonski dizajni** imenuju, apstrahuju i identifikuju ključne aspekte zajedničke strukture dizajna zbog kojih bi šablon bio koristan za višekratno korišćenje. Šablon dizajna identifikuje upotrebljene klase i primerke, njihove uloge i saradnju, kao i distribuciju odgovornosti (Gamma i dr. 1995).

Kao što ćemo videti u sledećim poglavljima, u idealnom slučaju želimo dizajn ponovo da upotrebimo bez ikakve izmene. Ali, često moramo da promenimo dizajn kako bi on zadovoljio zahteve i ograničenja koji su u sadašnjem sistemu drugačiji nego što su bili u prethodnim sistemima. Prema tome, važno je praviti šablonske dizajne tako da ne budu previše čvrsto vezani za specifičnosti određenog sistema. To jest, želimo da što više omogućimo višekratno korišćenje, a istovremeno da zadovoljimo zahteve trenutnog sistema.

5.5 KARAKTERISTIKE DOBROG DIZAJNA

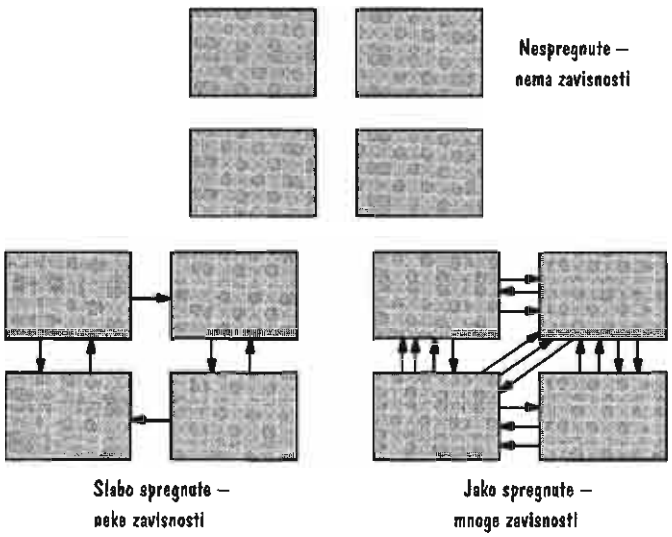
U poglavlju 1 pregledali smo nekoliko modela za kvalitet softvera, da bismo videli koji su atributi poželjni u proizvodima koje pravimo. Dok projektujemo sistem, želimo da obezbedimo neke od ovih atributa kvaliteta. Dizajn najvišeg kvaliteta bi trebalo da ima karakteristike koje čine kvalitetan proizvod: lako razumevanje, laka implementacija, lako testiranje, lako modifikovanje i pravilno prevođenje iz specifikacije zahteva. Posebno je važno lako modifikovanje zato što promene u zahtevu ili promene potrebne da bi se ispravile greške ponekad zahtevaju promenu dizajna. U ovom odeljku detaljnije ćemo razmotriti atribute u kojima se ogleda kvalitet dizajna.

Nezavisnost komponenti

Apstrakcija i sakrivanje informacija omogućavaju da se ispitaju načini na koje su komponente međusobno povezane u sveukupni dizajn. U podsetniku 5.4 opisano je kako bi u projektovanju trebalo razmatrati pitanja kontrole komponenti. U većini projekata se trudimo da komponente budu što nezavisnije jedna od druge. Komponentu koja nije na složen način vezana sa drugima nije samo lakše shvatiti već se nezavisne komponente takođe mnogo lakše menjaju. Slično tome, kada se analizom koda utvrdu da greška u sistemu potiče iz dizajna, razlog će se lakše izolovati i ispraviti ako su komponente nezavisne.

Za prepoznavanje i merenje stepena nezavisnosti komponenti u dizajnu koriste se dva koncepta: međusobna spregnutost i unutrašnja kohezija (Yourdon i Constantine 1978).

Međusobna spregnutost. Kažemo da su dve komponente **jako spregnute** ako između njih postoji velika zavisnost. **Slabo spregnute** komponente imaju neku međuzavisnost, ali je njihova međusobna spregnutost slaba. **Nespregnute** komponente nemaju nikakve međusobne veze; one su potpuno nezavisne, kao što je prikazano na slici 5.12.

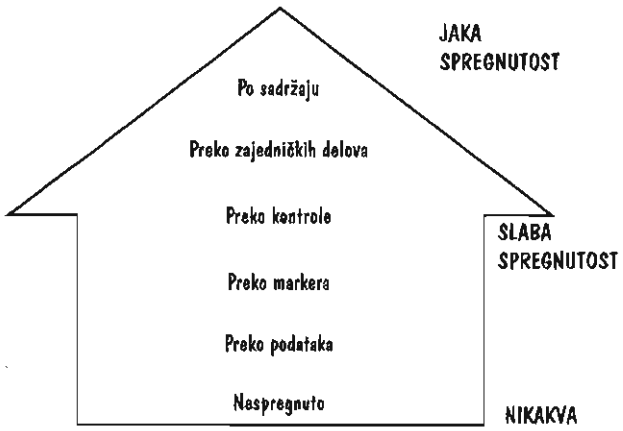


SLIKA 5.12 Međusobna spregnutost komponenti

Komponente mogu da zavise jedna od druge na mnogo načina, pa međusobna spregnutost zavisi od nekoliko stvari:

- Reference iz jedne komponente u drugu. Na primer, komponenta A može da poziva komponentu B, pa komponenta A zavisi od komponente B za izvršenje te funkcije ili procesa.
- Količina podataka koji se prenose iz jedne komponente u drugu. Na primer, komponenta A može komponenti B da predaje parametar, sadržaj matrice ili blok podataka.
- Stepen kontrole jedne komponente nad drugom. Na primer, komponenta A može komponenti B da preda kontrolni indikator. Vrednost indikatora obaveštava komponentu B o statusu nekog resursa ili podsistema, o tome koji se proces da poziva, ili da li se neki proces uopšte poziva.
- Stepen složenosti interfejsa između komponenti. Na primer, komponenta A predaje komponenti B parametar, nakon čega B može da se izvršava. Ali, komponente C i D razmenjuju vrednosti pre nego što D može da zaključi izvršenje, tako da je interfejs između A i B jednostavniji od interfejsa između C i D. Još je složenije korišćenje monitora ili zaštitnika.

Prema tome, međusobnu spregnutost možemo meriti na skali spregnutosti, počevši od potpune spregnutosti do potpune nespregnutosti, kao što je prikazano na slici 5.13.



SLIKA 5.13 Raspon spregnutosti

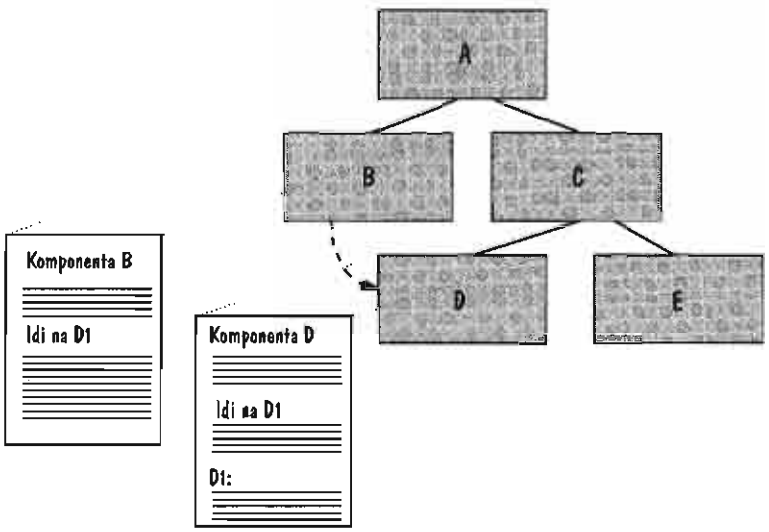
U stvarnosti, malo je verovatno da bi se mogao napraviti sistem od potpuno nespregnutih komponenti. Kao što sto i stolice, iako potpuno nezavisni, mogu u kombinaciji da čine trpezariju, naizgled nespregnute komponente može indirektno da povezuje kontekst. Zato naš cilj nije obavezno potpuna nespregnutost, već da sprega bude što slabija. Znači, želimo što manju zavisnost među komponentama, i to iz jednog dobrog razloga. Ako neka sistemska aktivnost utiče na jedan element, uvek želimo da znamo koja je komponenta sistema proizvela taj uticaj u datom trenutku. To saznanje nam dozvoljava da promenom jednog dela dizajna sistema izazovemo što manje poremećaja. Na primer, pretpostavimo da smo prilikom ispravljanja neke greške ili unošenjem promene na za-

htev kupca odlučili da se jedna funkcija ili tip podataka promeni. Idealno bi bilo kad bismo prosto mogli jednu komponentu da zamenimo drugom. Ako imamo modularan dizajn sa slabim vezivanjem među komponentama, možda bi mogao da se primeni scenario izvadi-stavi. Ako je vezivanje slabo, promenom će biti pogođeno samo nekoliko drugih komponenti koje će morati da se promene ili zamene. Ali, ako je vezivanje jako, tada promena može da utiče na veliki deo sistema. Prema tome, slabo vezivanje pomaže da broj komponenti koje treba prepraviti ostane minimalan.

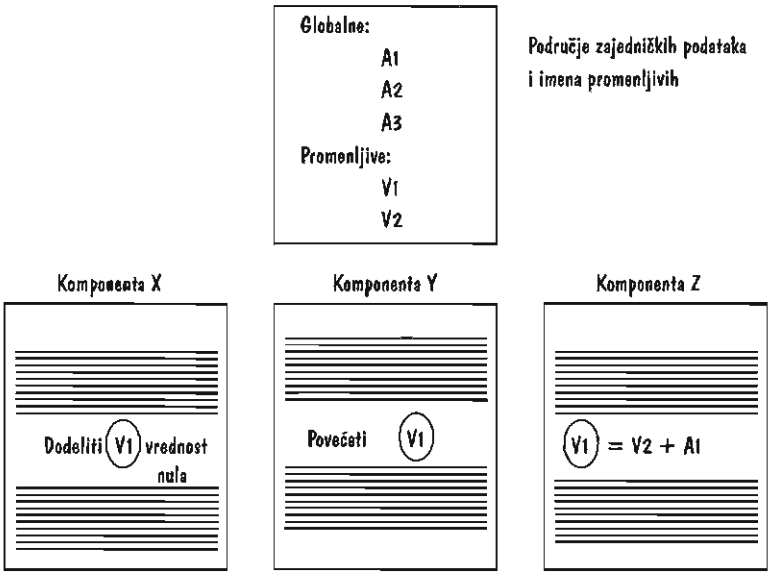
Neki tipovi međuzavisnosti su manje poželjni od drugih. Najnepoželjniji slučaj je kada jedna komponenta sasvim menja drugu komponentu. Tada promenjena komponenta potpuno zavisi od komponente koja je menja. Ovo nazivamo **spregnutost preko sadržaja**. Spregnutost preko sadržaja nastupa kada jedna komponenta menja interni podatak u drugoj komponenti, ili kada se jedna komponenta grana na sredinu druge komponente. Na slici 5.14 komponenta B se grana u D iako je D pod kontrolom komponente C.

Moguće je donekle smanjiti stepen međuzavisnosti ako dizajn organizujemo na takav način da podaci budu dostupni iz zajedničkog skladišta podataka. Međutim, sprega i dalje postoji zato što prilikom promene zajedničkih podataka moramo da ispitamo sve komponente koje tim podacima pristupaju da bi se procenio efekat promene. Ova vrsta zavisnosti naziva se **sprega preko zajedničkih delova**. Kod spregnutosti preko zajedničkih delova ponekad je teško odrediti koja komponenta je dodelila promenljivoj određenu vrednost. Na slici 5.15 prikazano je kako funkcioniše ovaj vid međuzavisnosti.

Kada jedna komponenta predaje parametre za kontrolisanje rada druge komponente, kažemo da između te dve komponente postoji **sprega preko kontrole**. Kontrolisana komponenta ne može da funkcioniše bez uputstva iz komponente koja vrši kontrolu. U dizajnu koji sadrži ovaj vid međuzavisnosti bilo bi bolje da svaka komponenta izvršava samo jednu funkciju ili jedan proces. Tako se smanjuje količina kontrolnih informacija koja se prenosi iz jedne komponente u drugu i kontrola se ograničava na fiksiran i prepoznatljiv skup parametara koji obrazuju dobro definisani interfejs.



SLIKA 5.14 Primer sprege preko sadržaja



SLIKA 5.15 Primer sprege preko zajedničkih delova

Kada se za prenos informacija iz jedne komponente u drugu koristi struktura podataka, pa se prenosi cela struktura podataka, kažemo da među komponentama postoji **sprega preko markera**; ako se predaju samo podaci, kažemo da postoji **sprega preko podataka**. Ako imamo **spregu preko markera**, u komponentama moraju da postoje identične vrednosti podataka, identičan format i organizacija. Prema tome, sprega preko podataka je jednostavnija i daje manju verovatnoću greške. Ako između komponenti mora da postoji veza, najpoželjnija je sprega preko podataka, jer se podaci najlakše prate kada treba uneti promene.

Komponente u objektno orijentisanom dizajnu obično su slabo spregnute, pošto svaka definicija objekta komponente sadrži definicije akcije koje ona izvršava i koje se vrše nad njom. Prema tome, kada se izabere objektno orijentisani pristup, automatski se dobija slaba spregnutost.

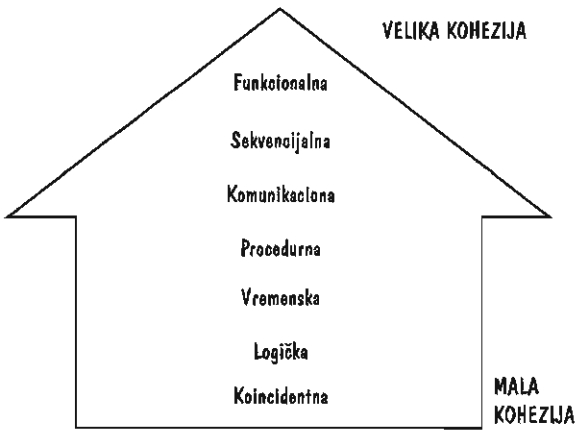
Unutrašnja kohezija modula. Za razliku od merenja međusobne zavisnosti komponenti, kohezija se odnosi na unutrašnji „lepak" kojim je komponenta izgrađena. Ako komponenta ima veliku koheziju, njeni unutrašnji delovi su jače vezani jedni za druge i za njenu osnovnu svrhu. Drugim rečima, komponenta je kohezivna ako svi njeni elementi služe izvršavanju istog zadatka i bitni su za to izvršavanje.

Prvobitne definicije ovih stepena bile su zasnovane na pojmovima funkcionalnog raščlanjivanja. Zajednički cilj dizajna bio je da svaka komponenta bude što kohezivnija tako da svi delovi obrade u komponenti budu usmereni na njenu jedinstvenu funkciju. Ovu definiciju možemo promeniti da bi bila primenjiva na sve vrste raščlanjavanja. To jest, možemo da pravimo nivoe kohezije za komponente bez obzira na način kojim je izvršeno raščlanjavanje. Na slici 5.16 prikazano je nekoliko vrsti kohezija.

Najlošiji stepen kohezije, **koincidentnu koheziju**, imamo u komponenti čiji delovi nisu međusobno spregnuti. U tom slučaju, nespregnute funkcije, procesi ili podaci nalaze se u

istoj komponenti iz praktičnih razloga, ili sasvim slučajno. Na primer, koincidentno kohezivna je komponenta u kojoj se proverava bezbednosna klasifikacija korisnika i takođe se štampa platni spisak za ovu nedelju.

SLIKA 5.16 Vrste kohezije



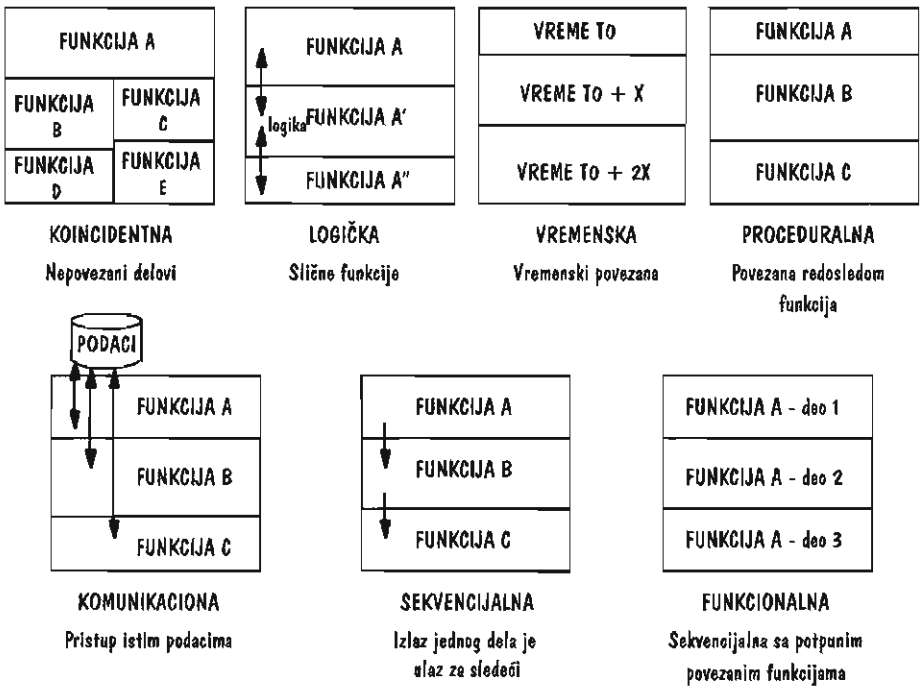
Sledeći jači stepen kohezije jeste **logička kohezija** (mada i dalje nepoželjna) gde se u istoj komponenti nalazi nekoliko logički povezanih funkcija ili podataka. Na primer, jedna komponenta može da učitava sve vrste ulaza (trake, diskove i telekomunikacione portove), bez obzira na to odakle ulaz stiže i kako će se koristiti; „ulaz" je spona koja povezuje ovu komponentu. Mada ima više smisla od koincidentne kohezije, elementi logičke kohezije nisu funkcionalno povezani. U našem primeru, pošto ulaz može da ima različite svrhe za različite komponente, na jednom mestu izvršavamo više nepovezanih funkcija.

Ponekad koristimo jednu komponentu za inicijalizovanje sistema ili skupa promenljivih. Takva komponenta izvršava nekoliko funkcija u nizu, ali su funkcije povezane jedino po vremenu izvršavanja, pa je to **vremenska kohezija**. Komponente sa vremenskom i logičkom kohezijom teško se menjaju. Pretpostavimo da morate promeniti dizajn sistemske funkcije *X*. Pošto logički ili vremenski kohezivne komponente izvršavaju više različitih funkcija, kada menjate funkciju *X*, morate da pretražite sve komponente kako biste pronašli delove koji utiču na funkciju *X*.

Često neke funkcije moraju da se izvršavaju u određenom redosledu. Na primer, podatke treba prvo uneti da bi mogli da se provere i zatim se njima manipuliše: tri funkcije u zadatom redosledu. Kada se funkcije grupišu u jednu komponentu samo da bi se obezbedio taj redosled, kažemo da je komponenta **proceduralno kohezivna**. Druga alternativa je da udružimo neke funkcije zato što se izvršavaju nad istim podacima ili proizvode iste podatke. Na primer, ponekad se istovremeno preuzimaju nepovezani podaci zato što se preuzimanje može izvršiti jednim pristupom disku ili traci. Komponente koje su sastavljene na ovaj način su **komunikaciono kohezivne**. Međutim, komunikaciona kohezija često remeti modularnost i funkcionalnu nezavisnost dizajna.

Ako se izlaz iz jednog dela neke komponente koristi kao ulaz u sledeći njen deo, kažemo da komponenta ima **sekvencijalnu koheziju**. Pošto i takva komponenta nije napravljena na osnovu funkcionalnih relacija, mogućno je da ta komponenta ne obuhvata

svu obradu jedne funkcije. Ideal kome težimo jeste **funkcionalna kohezija**, u kojoj je svaki element obrade bitan za izvršavanje samo jedne funkcije i u kojem se svi bitni elementi nalaze u jednoj komponenti. Funkcionalno kohezivna komponenta izvršava samo funkciju za koju je projektovana, ali takođe izvršava samo tu funkciju i ništa više. Na slici 5.17 dat je primer ovih različitih tipova kohezije.



SLIKA 5.17 Primeri kohezije

Pojam kohezije može se proširiti na objektno orijentisane i druge vrste dizajna u kojima se komponente zasnivaju na podacima ili na događajima ako imamo u vidu krajnji cilj: da spajamo samo one objekte i akcije koje imaju jednu zajedničku smislenu svrhu. Na primer, kažemo da je komponenta objektno orijentisanog dizajna kohezivna ako su svi njeni atributi, metodi ili akcije bitni za objekat. Objektno orijentisani sistemi često imaju veoma kohezivan dizajn pošto ih postupak izgradnje primorava da postave akcije uz objekat nad kojim se izvršavaju.

Prepoznavanje i obrada izuzetaka

Kada učimo da vozimo kola, kažu nam da treba da vozimo defanzivno. To znači da ne preuzimamo aktivnu ulogu samo kada reagujemo na incidente koji su se dogodili, već i da predviđamo kada bi situacija mogla da postane opasna i da preduzmemo odgovarajuće postupke da bi se problem izbegao. Na isti način, trebalo bi i da projektujemo defanzivno, da pokušamo da predvidimo situacije koje bi mogle da dovedu do problema u sistemu.

Defanzivno projektovanje nije jednostavno. Specifikacija zahteva govori o tome šta bi sistem trebalo da radi, ali obično nije očigledno šta sistem ne bi smeo da radi. Ekstre-

mni primer je skup bezbednosnih zahteva koji nam kažu da sistem treba da radi ono što se zahteva, ali ništa drugo (C. P. Pfleeger 1997a). Kako da definišemo „ali ništa drugo" i kako da to testiramo? Jedan od načina je da pronađemo poznate **izuzetke** - tj. situacije za koje znamo da su suprotne onome što sistem zaista treba da radi. Zatim, u projekat uključimo **obradu izuzetaka** tako da sistem na zadovoljavajući način reaguje na svaki izuzetak i ne degradira funkcionisanje sistema.

Tipični izuzeci su:

- nemogućnost pružanja usluge;
- pružanje pogrešne usluge ili podataka;
- oštećenje podataka.

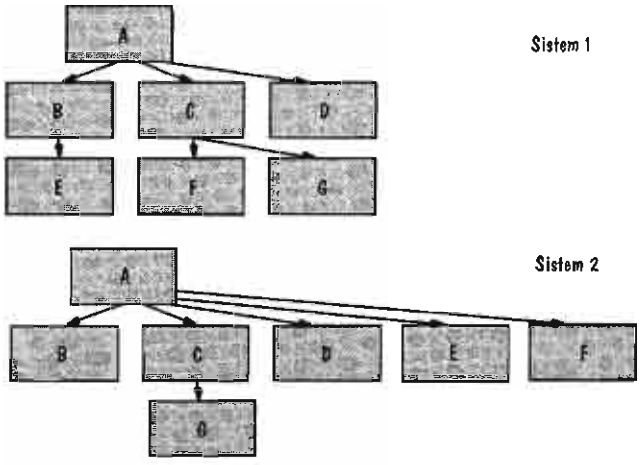
PODSETNIK 5.4 PITANJA KONTROLE

U većini različitih slučajeva, prilikom izrade dizajna treba doneti odluku koliko komponenti će biti pod kontrolom jedne određene komponente. U strukturnom dijagramu za sistem 1 na slici 5.18. strelica povezuje dve komponente samo ako prva komponenta može da pozove drugu. Za datu komponentu se skup komponenti prema kojima postoje strelice naziva njenim *kontrolnim dometom*. Sve komponente koje data komponenta poziva zajedno čine *domen efekta*. Nijedna komponenta ne bi smela da bude u domenu efekta ako se ne nalazi u kontrolnom dometu. Ako je domen efekta jedne komponente širi od njenog kontrolnog dometa, gotovo da je nemoguće garantovati da promena te komponente neće uništiti ceo dizajn.

Razmotrite sisteme 1 i 2 kao dva moguća dizajna jednog istog sistema. *Ulazni faktor* je broj komponenti koje kontrolišu jednu određenu komponentu, a *izlazni faktor* je broj komponenti koje ta komponenta kontroliše. Prema tome, u sistemu 1 komponenta A ima Iz-fak = 3, a u sistemu 2 je njen Iz-fak = 5. Slično tome, Ul-fak za komponentu C jednak je 1 u oba sistema.

Koji je dizajn bolji? U opštem slučaju, želimo da imamo što manji broj komponenti sa visokim Iz-faktorom. Komponenta koja kontroliše mnoge druge komponente obično je znak da kontrolna komponenta ima previše zadataka; kontrolna komponenta je verovatno višefunkcionalna. Prema tome, dizajn sistema 1 je bolji nego dizajn sistema 2, zato što njegove komponente imaju niski izlazni faktor. S druge strane, kako povećavamo broj nivoa u dizajnu, ponekad se javi potreba da se određena komponenta ponovo upotrebi. Na primer, često se javlja potreba da se u nekom stringu traži konkretan znak. Ako napravimo opštenamensku komponentu koja to obavlja i zatim je pozivamo iz mnogih drugih komponenti, dobijeni dizajn je efikasniji i lakše ga je testirati i menjati od nekog sistema u kome bismo stringove morali da tražimo pomoću gomile sličnih funkcija. Prema tome, za dizajn sa velikim brojem nivoa napravićemo skup *pomoćnih* komponenti, alatki ili gradivnih blokova koje druge komponente koriste za izvršavanje čestih zadataka. Tipična pomoćna komponenta će imati visoki ulazni faktor zato što je poziva mnogo drugih komponenti. Jedan od naših ciljeva u projektovanju sistema jeste da pravimo komponente koje imaju visoki ulazni faktor, a niski izlazni faktor.

SLIKA 5.18 Primer kontrole



Svaki izuzetak koji prepoznamo možemo da obradimo na jedan od tri načina:

1. *Pokušati ponovo:* Vraćamo sistem u prethodno stanje i ponovo pokušamo da izvršimo uslugu primenom druge strategije.
2. *Ispraviti:* Vraćamo sistem u prethodno stanje, ispravimo neki aspekt sistema i ponovo pokušamo da izvršimo uslugu primenom iste strategije.
3. *Izvestiti:* Vraćamo sistem u prethodno stanje, predamo izveštaj o problemu komponenti koja se bavi greškama i ne vršimo uslugu.

Prema tome, za svaku uslugu koju želimo da naš sistem vrši, moramo da otkrijemo načine na koje ona zapada u problem kao i načine kako da se ti problemi reše. Kao što ćemo videti kasnije u ovom poglavlju, u traženju izuzetaka možemo da koristimo analizu sa stablom grešaka i analizu režima otkazivanja.

Meyer (1992a) daje primer kako se obrada izuzetaka može ugraditi u dizajn. Pretpostavimo da šaljemo poruku preko mreže. Znamo da, u slučaju da postupak ne uspe želimo ponavljanje slanja i odustajanje nakon 100 neuspešnih pokušaja. Te informacije možemo da uključimo u dizajn:

```
pokušaj_prenosa (poruka: STRING) je
-- Pokušaj da se poruka prenese preko komunikacione linije
-- pomoću procedure nižeg nivoa nesiguran_prenos,
-- koji može da ne uspe, pa pokreće izuzetak.
-- Nakon 100 neuspešnih pokušaja, odustati (pokretanjem
-- izuzetka u pozivaocu).
local
neuspeh: INTEGER
do

nesiguran_prenos (poruka)
rešenje
neuspeh := neuspeh + 1;
```

```
if neuspeh < 100 then probaj_opet
end
end
```

U dizajnu možemo koristiti nekoliko tehnika za hvatanje izuzetaka dok se kôd izvršava:

- kontrolni zbirovi i kontrolne cifre: za dvostruku proveru ispravnosti podataka i izračunavanja;
- redundantne veze: predviđanje pokazivača unapred i unazad;
- merenje vremena.

Sprečavanje grešaka i otpornost na greške

U podsetniku 5.5 imamo upozorenje da bezbednost softvera ne bi trebalo uzimati zdravo za gotovo. Dizajn bi trebalo da predvidi greške i da ih obradi tako da što manje ometa rad i pruži maksimalnu bezbednost. Cilj je da kôd bude što čitiji od grešaka, a to se postiže tako što se u sam dizajn ugrađuje sprečavanje otkazivanja i obrada grešaka. Videli smo kako se koristi obrada izuzetaka za rešavanje neuobičajenih situacija koje možemo unapred da predvidimo. Takođe želimo da se obezbedimo od grešaka ugrađenih u svaku komponentu, kao i od grešaka koje potiču od drugih komponenti, drugih sistema i interfejsa.

Kako dolazi do otkazivanja? Kada čovek pogreši, ljudska greška dovodi do **greške** u nekom softverskom proizvodu. Na primer, možda ste pogrešno razumeli zahtev korisničkog interfejsa i napravili ste dizajn u kojem se ogleda to pogrešno shvatanje. Greška u dizajnu može da dovede do neispravnog koda, neispravnih uputstava u priručniku za korisnike ili do neispravnog skripta za testiranje. Na ovaj način, jedna greška može da proizvede jednu ili više grešaka, u jednom proizvodu ili u više njih.

Postoji razlika između grešaka i otkaza. **Otkaz** je situacija kada se sistem ne ponaša prema zahtevu. Otkaz može da se otkrije pre ili posle isporuke sistema, pošto može da nastupi u toku testiranja ili u toku rada. U nekom smislu, greške i otkaz predstavljaju unutrašnji i spoljašnji prikaz sistema. Drugim rečima, greške predstavljaju probleme koje vide projektanti, dok su otkazi problemi koje vide korisnici ili kupci.

Važno je shvatiti da ne dovodi svaka greška do otkaza, pošto se uslovi u kojima greška dovodi do otkaza sistema možda nikad neće dogoditi. Na primer, kôd koji sadrži grešku možda se nikad neće izvršavati ili se neće dovoljno izvršavati da bi se neki brojač uvećao preko dozvoljenih granica (kao u slučaju Ariane-4).

Jedna od karakteristika dobrog dizajna je način na koji se sprečavaju ili tolerišu greške. Umesto da čeka da sistem pogreši i da tada rešava problem, dizajner može da predvidi šta bi moglo da se dogodi i izgradi sistem koji će reagovati na prihvatljiv način.

Aktivno otkrivanje grešaka. Kada projektujemo sistem koji čeka da se greška javi u vreme izvršavanja, primenjujemo **pasivno otkrivanje grešaka**. Međutim, ako povremeno proveravamo simptome grešaka ili pokušavamo da predvidimo kada će doći do otkazivanja, primenjujemo **aktivno otkrivanje grešaka**. Na primer, možemo da primenimo politiku **međusobne sumnje**, gde svaka komponenta sistema polazi od toga da ostale komponente sadrže greške. Svaka komponenta proverava ispravnost dobijenih podataka i njihovu konzistentnost. Na ovaj način, program za platni spisak bi proverio da *časovi_rada* nisu negativni pre nego što izračuna iznos plate. Štaviše, grešku bi trebalo obraditi čim se otkrije, a ne da se čeka završetak obrade. Takva trenutna obrada greške ograničava štetu nastalu zbog greške, umesto da dozvoli da greška dovede do otkaza a zatim do destrukcije.

PODSETNIK 5.5 POTREBA ZA BEZBEDNIM DIZAJNOM

Koliko su bezbedni sistemi koje projektujemo? Izveštaje iz prakse je teško protumačiti. Kod nekih sistema je jasno da je korisno što su im neke funkcije implementirane softverski, a ne hardverski (odnosno što nisu prepuštene na rešavanje ljudima koji rade sa sistemom). Na primer, u automobilskoj industriji i proizvodnji aviona tvrde da je sprečen veliki broj nesreća time što se u kontrolne sisteme uvodi sve više i više softvera. Međutim, drugi podaci su zabrinjavajući. Na primer, od 1986. do 1997. američki FAD (telo za kontrolu hrane i lekova) primio je preko 450 izveštaja o softverskim greškama u medicinskoj opremi, od kojih je 24 dovelo do smrti ili povrede (Anthes 1997).

Ovaj broj možda predstavlja samo vrh ledenog brega. U stvari, pošto se hiljade takvih izveštaja mora predati od roku od 15 dana od incidenta, proizvođači možda još nisu znali pravi razlog za otkazivanje kada su podnosili izveštaj. Na primer, u jednom slučaju kada su prema izveštaju otkazale baterije, na kraju je otkriveno da je greška u softveru, u stvari, dovela do njihovog pražnjenja. A Leveson i Turner (1993) detaljno opisuju probleme koji su doveli do bar 3 smrti i nekoliko povreda, zbog loseg korisničkog interfejsa za mašinu za terapiju zracenjem. Značaj softverskog dizajna postaje odgledan u mnogim organizacijama koje ranije nisu bile svesne uloge softvera. Na primer, juna 1997. nova regulativa na saveznom nivou ovlastila je FDA da ispituje i dizajn softvera medicinskih uređaja.

Problemi sa dizajnom softvera javljaju se svuda i nisu ograničeni samo na medicinske uređaje, i mnogi razvijaoci su posebno predostrožni. Kada ispituje kontrolni softver nuklearnog reaktora, baltimorska kompanija za gas i struju sprovodi i reviziju dizajna. Takođe mnoge grupe u kompaniji Hewlett-Packard koriste priliku da tokom zvaničnih inspekcija i kontrola odstrane i greške u dizajnu, pre nego što počne kodiranje (Grady i Van Slack 1994).

Anthes (1997) izveštava o predlozima Alana Barbella, vođe projekta u Environmental Criminology Research, instituta koji kontroliše medicinske uređaje. Barbell kaže da dizajneri softvera moraju da obezbede direktan uvid u način korišćenja njihovih proizvoda, a ne da se oslone na ljude iz prodaje i marketinga. Dizajneri mogu da ugrade preventivne mere koje će obezbediti da se njihov proizvod ne koristi na pogrešan način.

Drugi pristup aktivnom otkrivanju grešaka je upotreba redundanse, gde se porede rezultati dva procesa i ispituje da li su jednaki. Na primer, knjigovodstveni program može da sabere sve redove, a zatim sve kolone kako bi obezbedio da totali budu identični. Slično tome, kontrolni zbir, kontrolna cifra ili bit parnosti u sklopu prenosa podataka može da upozori sistem na grešku u podacima. U nekim sistemima postoji čak više računara koji vrše ista izračunavanja; svemirski šatl funkcioniše na taj način, i u njemu se sledeći postupak određuje glasanjem sedam računara. Po teoriji, ako dva funkcionalno ekvivalentna sistema projektuju dva različita tima projektanata u različito vreme koristeći različite tehnike, veoma je mala verovatnoća da ti sistemi u isto vreme naprave istu grešku. Međutim, ovaj pristup koji se naziva **programiranje sa n verzija** pokazao se manje pouzdanim nego što se mislilo, zato što mnogi dizajneri uče da projektuju na sličan način i primenjuju slične šablone i tehnike (Knight i Leveson 1986).

Kada je potrebno aktivno otkrivanje grešaka, sistemski dizajn često predvida da još jedan računar radi paralelno sa prvim. Drugi sistem ispituje prvi, analizira sistemske podatke tražeći znakove koji bi mogli da ukažu na problem. Na primer, drugi sistem može

da pronađe proces koji nije bio na rasporedu u dužem vremenskom periodu. Ovaj simptom bi mogao da bude znak da se prvi sistem „negde zaglavio" u nekoj petlji unutar procesa ili da čeka na ulaz. Takođe sistem može da pronađe blok memorije koji je dodeljen i nije više u upotrebi, ali se ne nalazi na spisku raspoloživih blokova. Slično tome, drugi sistem može da otkrije komunikacionu liniju koja nije oslobođena na kraju prenosa. Ta tehnika se koristi u nekim sistemima za obradu podataka „bez greške" gde dva procesora paralelno izvršavaju isti posao i neprestano upoređuju rezultate.

Ako drugi od dva redundantna sistema ne može da otkrije neke nedostatke samim poređenjem povezanih podataka, on može da pokreće **dijagnostičke transakcije**. U ovoj tehnici drugi sistem generiše lažne, ali bezazlene transakcije u prvom sistemu, kako bi utvrdio da li prvi sistem pravilno radi. Na primer, drugi sistem može da pozove prvi i proveri da li prvi sistem odgovara na poziv.

Ispravljanje grešaka. Kada se greška otkrije, ona mora da se otkloni. **Ispravljanje grešaka** predstavlja kompenzaciju sistema za prisustvo greške. Obično, ispravljanje grešaka otklanja štetu koju je greška izazvala, a takođe i menja proizvod da bi se greška eliminisala u budućnosti. Prema tome, naš dizajn mora da sadrži strategiju za obradu grešaka kako se na njih nailazi. Često zaustavljamo sistem ako ta greška na neki način utiče na sistem (tj. kada dođe do otkazivanja) ili naprosto evidentiramo postojanje otkazivanja, zabeležimo status sistema u vreme njegovog nastanka, a kasnije ispravljamo nastalu štetu.

Stepen kritičnosti sistema određuje strategiju koju ćemo da izaberemo. Na primer, kada telefonski poziv dovede do loše veze, strategija sistema je ponekad da se prekine linija; od kupca se očekuje da ponovo zove, daje se prednost sveukupnom integritetu telekomunikacionog sistema pred pojedinačnim pozivom. Međutim, takva strategija bi bila nezamisliva u medicinskom uređaju ili u sistemu avijacije.

Održavanje sistema je još jedan faktor pri odlučivanju koja će se strategija koristiti za ispravljanje grešaka. Mnogo je lakše pronaći izvor problema ako se rad naglo prekine nego ako sistem nastavi da radi; nastavak može da proizvede druge efekte zbog kojih se početna greška teže otkriva.

Otpornost na greške. Ponekad je ispravljanje greške isuviše skupo, riskantno ili nezgodno. Zato dizajn svodi na minimum štetu nastalu zbog greške i nastavlja rad sa što manje smetnji za korisnike. **Otpornost na greške** je izolovanje štete do koje je greška dovela i u mnogim slučajevima je praktična ili čak i poželjna. Na primer, pretpostavimo da softver kontroliše nekoliko prenosnih traka u proizvodnoj liniji. Ako se otkrije greška na jednoj od traka, sistem bi mogao da uključi zvono i preusmeri materijale na drugu traku. Kada se oštećena traka popravi, ona može da se vrati u proizvodnju. Taj pristup je svakako poželjniji nego da se proizvodnja potpuno prekine dok se neispravna traka ne popravi. Slično tome, bankarski sistem može da se prebaci na rezervni procesor ili da pravi dvostruke kopije podataka i transakcija za slučaj da jedan procesor otkaže.

Neke strategije za otpornost na greške oslanjaju se na sposobnost predviđanja lokacije grešaka i vremenskog rasporeda otkazivanja. Da bi se u dizajnu sistema napravila alternativna rešenja, dizajneri moraju biti u stanju da pogode do kakvih grešaka bi moglo da dođe. Neke greške je lako predvideti, ali složeniji sistemi su teži za analizu. U isto

vreme, veća je verovatnoća da će do značajnih grešaka doći u složenijim sistemima. I sam kôd kojim se implementira otpornost na greške može da sadrži greške koje mogu da izazovu nepopravljivu štetu. Zbog toga neke strategije za otpornost na greške izoluju područja mogućih grešaka umesto da predviđaju same greške. U sledećem odeljku istražićemo nekoliko tehnika koje pomažu da dizajn bude otporan na greške, a da se time ne poveća rizik.

5.6 TEHNIKE ZA POBOLJŠANJE DIZAJNA

U fazi dizajniranja možemo da koristimo mnoge tehnike koje smo koristili za definisanje zahteva. Tabele stanja, dijagrami toka podataka, SADT dijagrami i ostale tehnike mogu da se prilagode i da prikažu „kako" dizajn funkcioniše, a ne „šta" su zahtevi. Kako gradimo dizajn, možemo koristiti karakteristike prethodnog odeljka i postaviti ciljeve za kvalitet dizajna. Dizajn koji dostigne ciljeve obično se lakše implementira, testira, ispravlja i održava i zato pokušavamo da karakteristike kvaliteta ugradimo u sistem u što ranijoj fazi. U ovom odeljku, pogledaćemo nekoliko načina za unapređenje dizajna pre implementacije.

Smanjenje složenosti

Dok pravimo naš dizajn, pokušavamo da koliko je god moguće uprostimo njegovu strukturu ne menjajući pri tom prirodu rešenja. To uprošćavanje nam pomaže da lakše shvatimo rešenje. Kako se atributi veoma kvalitetnog dizajna prevode i na preostali razvoj, trebalo bi da zahvaljujući njima bude lakše projektovati, izgraditi i održavati softver i hardver.

Na primer, pomoću dijagrama međusobnih dejstava u sistemu može se smanjiti složenost pa će on biti lakši za razumevanje. Možemo ponovo da nacrtamo grafikone i dijagrame, da smanjimo broj ukrštanja kako bi bili jednostavniji za razumevanje. Ponovnim crtanjem sa očuvanjem relacija, ali svođenjem ukrštanja na najmanju meru, pokazuje se **planarnost** grafikona, minimalni broj ukrštanja koja se moraju upotrebiti kada se grafikon crta na papiru.

Ponekad je moguće smanjiti složenost tabela odlučivanja koje opisuju aktivnosti sistema. Na primer, razmotrite ponovo tabelu za prijem na univerzitet o kojoj smo govorili u poglavlju 4. Prijem na univerzitet zasniva se na četiri faktora: visokim ocenama, ocenama visoko standardizovanih testova, izvanrednim preporukama i vanškolskim aktivnostima. Za svakog kandidata možemo prisustvo svake promenljive da obeležimo sa „1", a odsustvo te promenljive sa „0". Neka dodeljene promenljive budu predstavljene kao:

w = student ima prosek ocena = A ili B
x = student ima ukupan zbir ocena na standardnim testovima od 1200 ili više
y = student ima izvanredne preporuke
z = student ima vanškolske aktivnosti

Univerzitet može da preduzme niz aktivnosti na osnovu konkretne kombinacije promenljivih za svakog studenta:

A_1: pozvati studenta da se ranije upiše

A_2: primiti studenta u redovan semestar

A_3: pozvati studenta na specijalan program

A_4: pozvati studenta u redovan program

A_5: uslovni prijem

A_6: odbacivanje

U tabeli 5.2 prikazane su sve moguće kombinacije promenljivih i njihove posledice. Na primer, ako je w=1 i z=1, ali su x i y oba jednaka nuli, preduzimaju se akcije *A2* i *A4*: student se prima na redovno školovanje.

Za opisivanje relacije između akcija i promenljivih može da se koristi Bulova algebra. (Potpuni opis Bulove algebre i kako se ona može primeniti na tabele odlučivanja pročitajte u Pfleeger i Straight [1985].) Za svaku akciju predstavljenu kolonom dajemo izraz kao sumu proizvoda, za svaki red u kojem se pojavljuje *X* imaćemo jedan izraz. Na primer, u koloni za akciju *A1* imamo oznaku *X* u poslednja dva reda: kada su w, x i y=1, a z=0 kao i kada su sve promenljive jednake 1. Iz svakog ovakvog reda obrazujemo proizvod na sledeći način: kad god je promenljiva =1 upotrebite je u proizvodu; kad god je promenljiva =0 upotrebite u proizvodu njen komplement. Ako komplement promenljive predstavimo podvučenim istim slovom, tako što komplement x pišemo kao $\underline{x}$, akciju *A1* možemo da opišemo kao dva uslova koja dovode do *A1* : kao sumu proizvoda $w\,x\,y\,\underline{z}$ i $w\,x\,y\,z$:

$$w\,x\,y\,\underline{z} + w\,x\,y\,z = A1$$

Tabela 5.2 Tabela odlučivanja za prijem na univerzitet

Promenljive				Akcije					
w	*x*	*y*	*z*	*A1*	*A2*	*A3*	*A4*	*A5*	*A6*
0	0	0	0						X
0	0	0	1						X
0	0	1	0				X	X	
0	0	1	1				X	X	
0	1	0	0		X		X		
0	1	0	1		X		X		
0	1	1	0		X		X		
0	1	1	1		X		X		
0	0	0	0		X		X		
1	0	0	1		X		X		
1	0	1	0		X		X		
1	0	1	1		X		X		
1	1	0	0		X	X			
1	1	0	1		X	X			
1	1	1	0	X		X			
1	1	1	1	X		X			

Na sličan način, generišemo svih šest jednačina koje opisuju relacije između ulaznih promenljivih i rezultujućih aktivnosti:

$$A1 = w\,x\,y\,z + w\,x\,y\,z$$
$$A2 = \underline{w}\,x\,\underline{y}\,\underline{z} + \underline{w}\,x\,\underline{y}\,z + \underline{w}\,x\,y\,\underline{z} + \underline{w}\,x\,y\,z + w\,\underline{x}\,\underline{y}\,\underline{z} + w\,\underline{x}\,\underline{y}\,z + w\,\underline{x}\,y\,\underline{z} + w\,\underline{x}\,y\,z + w\,x\,\underline{y}\,\underline{z} + w\,x\,\underline{y}\,z$$
$$A3 = w\,x\,\underline{y}\,\underline{z} + w\,x\,\underline{y}\,z + w\,x\,y\,\underline{z} + w\,x\,y\,z$$
$$A4 = \underline{w}\,\underline{x}\,y\,\underline{z} + \underline{w}\,\underline{x}\,y\,z + \underline{w}\,x\,\underline{y}\,\underline{z} + \underline{w}\,x\,\underline{y}\,z + \underline{w}\,x\,y\,\underline{z} + \underline{w}\,x\,y\,z + w\,\underline{x}\,\underline{y}\,\underline{z} + w\,\underline{x}\,\underline{y}\,z + w\,\underline{x}\,y\,\underline{z} + w\,\underline{x}\,y\,z$$
$$A5 = \underline{w}\,\underline{x}\,y\,\underline{z} + \underline{w}\,\underline{x}\,y\,z$$
$$A6 = \underline{w}\,\underline{x}\,\underline{y}\,\underline{z} + \underline{w}\,\underline{x}\,\underline{y}\,z$$

Kao što vidite, neki od ovih izraza su prilično složeni. Međutim, za uprošćavanje ovih formula možemo da iskoristimo svojstvo da je $x + \underline{x} = 1$. Na primer, $A1$ možemo da redukujemo na sledeći način:

$$\begin{aligned} A1 &= w\,x\,y\,\underline{z} + w\,x\,y\,z \\ &= w\,x\,y(z + \underline{z}) \\ &= w\,x\,y(1) \\ &= w\,x\,y \end{aligned}$$

U vežbi 10 imaćete priliku da proverite da se prethodni izrazi mogu redukovati na svoje sledeće ekvivalente:

$$A1 = w\,x\,y$$
$$A2 = x\,y + \underline{w}\,x\,y + w\,\underline{x}$$
$$A3 = w\,x$$
$$A4 = x\,y + \underline{w}\,x + w\,\underline{x}\,\underline{y}$$
$$A5 = \underline{w}\,\underline{x}\,y$$
$$A6 = \underline{w}\,\underline{x}\,\underline{y}$$

Slične metode se često primenjuju na zahteve koji se postavljaju pred dizajniranje hardverske logike. Kao rezultat dobijaju se dizajni sa manje logičkih kola ili dijagrami ožičenja sa manje ukrštanja. U tim slučajevima, jednostavnost ne služi samo lakšem razumevanju već takođe smanjuje troškove hardvera, a testiranje je brže i jeftinije.

Dizajn po ugovoru

Meyer (1992a) predlaže jedan pristup dizajnu po imenu *dizajn po ugovoru* koji olakšava da dizajn zadovolji specifikaciju. On počinje od posmatranja softverskog sistema kao skupa komponenti koje komuniciraju i čije se međusobno dejstvo zasniva na precizno definisanoj specifikaciji o tome šta svaka komponenta treba da radi. Te specifikacije, koje on naziva **ugovorima** određuju kako komponenta treba da sarađuje sa ostalim komponentama i sistemima. Takva specifikacija ne može da garantuje ispravnost komponente, ali predstavlja dobru osnovu za testiranje i proveru valjanosti.

U svakodnevnom životu, ugovor se potpisuje između dva učesnika kada jedan naručuje od drugoga određenu uslugu ili proizvod. Svaki učesnik očekuje neku korist od

svake obaveze: dobavljač proizvodi uslugu ili proizvod u zadatom vremenskom periodu (obaveza) u zamenu za novac, a klijent prihvata uslugu ili proizvod (korist) za novac (druga obaveza). Ugovor ozvaničava obaveze i koristi.

Meyer primenjuje pojam ugovora na softver. Softverska komponenta, po imenu **klijent**, primenjuje strategiju da izvrši skup zadataka *t1*, *t2...tn*. Svaki netrivijalan podzadatak *ti* izvršava se redom kada klijent pozove drugu komponentu, **dobavljača**, koja će zadatak izvršiti. To znači da između dve komponente postoji ugovor za izvršavanje dela zadatka. Svaki ugovor obuhvata međusobne obaveze (**preduslove**), koristi (**postuslove**) i ograničenja vezana za konzistentnost (**invarijante**). Sva ta svojstva ugovora zajedno čine **tvrdnje**.

Na primer, pretpostavimo da klijentska komponenta sadrži tabelu u kojoj se svaki element prepoznaje po nizu znakova koji se koristi kao ključ. Zadatak komponente dobavljača je da doda element iz tabele u rečnik ograničene veličine. Ugovor između dve komponente možemo da opišemo na sledeći način:

1. Klijentska komponenta obezbeđuje da rečnik nije pun, a da je ključ neprazan.
2. Dobavljačka komponenta upisuje element u tabelu.
3. Klijentska komponenta pristupa ažuriranoj tabeli gde se element nalazi.
4. Ako je tabela puna ili je ključ prazan, nema nikakve aktivnosti.

Ovaj ugovor možemo da formalizujemo, a Meyer (1992a) daje primer na objektno orijentisanom jeziku Eiffel, u sklopu generičke klase *dictionary [element]*:

```
put (x: ELEMENT; key: STRING) is
-- Ubaciti x tako da može da se pronađe pomoću ključa.
require
count < = capacity;
not key.empty
do
... Neki algoritam umetanja ...
ensure
has (x);
item (key) = x;
count = old count + 1
end
```

Klauzula *require* uvodi preduslov; klauzula *ensure* uvodi postuslov.

Ugovori mogu da se grade za sve vrste dizajna. Na primer, pretpostavimo da koristimo apstrakciju podataka za projektovanje objektno orijentisanog sistema koji kontroliše protok vode u rezervoare i isticanje kroz brane. Mogli bismo da imamo klase objekata kao što su *brana*, *rezervoar* i *reka*. Možemo da postavljamo pitanja o objektima kao što su *je_prazan* i *je_pun* za rezervoar, a možemo i da izdamo komande kao što su *isprazni* ili *popuni*. Naš dizajn može da odredi preduslove i postuslove; na primer, možemo da napišemo

```
fill is
-- Popuniti branu vodom
require
in_valve.open;
out_valve.closed
deffered
-- tj. nije još implementirano
ensure
in_valve.closed;
out_valve.closed;
is_full
end
```

Tvrdnja koja opisuje svojstvo klase zove se **invarijanta klase**. Te karakteristike se uvek odnose na klasu, čak i prilikom promene i zato su bitne za testiranje ispravnosti promene. Na primer, možemo da navedemo precizno šta mislimo pod pojmom *je_pun* i zatim testiramo uslov.

```
invariant
is_full = (0.95 * capacity <= nivo) and (nivo <= 1.05 *
    capacity) ...
```

Klase i tvrdnje mogu da se uporede sa narednim implementacijama da bi se matematički dokazala njihova konzistentnost. Osim toga, tvrdnje čine osnovu za testiranje. Ljudi koji rade na testiranju mogu na jednoj po jednoj klasi da utvrđuju efekte svake tvrdnje. Štaviše, kada se dizajn na neki način promeni, svaka tvrdnja može da se proveri i da se utvrdi da li ju je promena oslabila ili ojačala. Na kraju ovog poglavlja videćemo kako se na softver Ariane mogao primeniti dizajn po ugovoru.

Dizajn sa prototipom

Ima slučajeva kada kupac zahteva podsistem ili funkciju za koju nismo sigurni da je možemo napraviti i da li je ona uopšte potrebna. Na primer, kupac može da zahteva sistem za upravljanje bazom podataka na nekoj određenoj platformi; mi nismo sigurni da ta platforma može da zadovolji zahtevane karakteristike istovremenog pristupa i zaključavanja polja. Kao što smo videli u poglavlju 4, možemo da razvijemo prototip baze podataka i u njega implementiramo samo one funkcije koje su potrebne da se odgovori na naše pitanje.

Prototip omogućava veliki deo istih prednosti u fazi dizajna kao i u fazi analize zahteva. Prototip izvodljivosti omogućava nam da već u fazi dizajna otkrijemo da li će rešenje koje predlažemo zaista rešiti postojeći problem. Tako nas prototip podstiče na međusobnu komunikaciju i na komunikaciju sa kupcima da bismo istražili područja neizvesnosti koja se pojavljuju dok razmišljamo o projektnom rešenju. Na ovaj način mnoga pitanja razrešavamo pre nego što počne kodiranje, a izbegava se nastanak dodatnih problema tokom testiranja.

U prototipu se često izostavljaju mnogi detalji funkcionalnosti i performansi realnog sistema, pa možemo da se usredsredimo na konkretan aspekt sistema i bolje ga shvatimo. Na primer, mali tim je možda naviknut na pravljenje prototipa za svaki od više aspekata jednog problema: jedan za korisnički interfejs, jedan za performanse, jedan za bezbednost itd. Konačni sistem je zatim sinteza rezultata dobijenih od pojedinačnih prototipova. Na taj način, svaki prototip je obično pun „rupa" koje kasnije treba popuniti.

Ako je prototip namenjen samo za prikazivanje izvodljivosti ili poželjnosti, možda njegovom dizajnu nećemo posvetiti istu pažnju koju bismo posvetili realnom sistemu. Zato, umesto da pokušamo da popunimo „rupe" koje su ostale u prototipu ili da iskoristimo kôd za neki od modela prototipa, često odbacujemo prototip i gradimo konačni sistem od nule. To jest, zna se da će se ovaj **prototip za bacanje** odbaciti; on se pravi samo da bi se utvrdila izvodljivost ili određene karakteristike većeg dizajna. Dok god je naš kupac svestan da je prototip model za istraživanje, a ne potpun proizvod, prototip može da bude koristan da bismo i mi i naši kupci shvatili šta sistem treba da radi. Pošto on podstiče saradnju i komunikaciju, izrada prototipa takođe unapređuje odnose između kupaca i projektanata. U stvari, Brooks (1975) preporučuje izgradnju sistema, njegovo odbacivanje i ponovnu izgradnju kako bi drugi sistem iskoristio ono što smo naučili prilikom otkrivanja grešaka u izgradnji prvog sistema.

Međutim, novija istraživanja softverskog inženjerstva podstiču upotrebu brze izgradnje prototipa i upotrebu delova prototipa u realnom sistemu. Velikom pažnjom u definisanju i razvijanju komponenti višeg nivoa brzi prototip može da odgovori na pitanja o dizajnu, a istovremeno da obezbedi delove za izgradnju konačnog sistema. U ovom svetlu, brzi prototip ujedno sadrži specifikaciju, dizajn, implementaciju i testiranje. Nedostatak je što ovaj proces mora da bude brz da bi imao ikakvu vrednost. Ako se prototip ne može napraviti brže od samog sistema, onda on ne ispunjava svoju svrhu.

Prema tome, kada odlučujemo da li izrada prototipa odgovara za naš projekat, potrebno je nekoliko procena. Boehm, Gray i Seewaldt (1984) proučavali su projekte kako bi utvrdili koji su projekti podesni za upotrebu prototipa. Utvrdili su da su proizvodi razvijeni pomoću prototipa funkcionisali isto tako dobro kao i oni koji su bili razvijeni tradicionalnim tehnikama dizajna. Osim toga, ako su korišćeni prototipovi, utrošeno je 45% manje napora i proizvedeno 40% manje redova koda. Brzina i efikasnost sistema razvijenih pomoću prototipa bili su skoro isti kao kod sistema razvijenih na tradicionalan način.

Analiza pomoću stabla greške

Već smo učili o značaju identifikacije grešaka, njihovog ispravljanja i otpornosti na greške. Postoji nekoliko tehnika koje se mogu koristiti prilikom dizajna kako bi se otkrile moguće greške i njihove verovatne lokacije. **Analiza pomoću stabla greške**, metod koji je prvobitno razvijen za program projektila U.S. Minuteman, razmatra dizajn i pomaže nam da ga raščlanimo i tražimo situacije koje bi mogle da dovedu do otkazivanja. U tom smislu naziv nije primeren; mi u stvari analiziramo otkaze, a ne greške i tražimo potencijalne uzroke tih otkaza. Mi pravimo stabla grešaka koja prikazuju logičke putanje od efekta prema uzroku.

Tabela 5.3 Podsetnik za pronalaženje eventualnih grešaka

Ključna reč	Objašnjenje
Nema	Podaci i kontrolni signal nisu poslati niti primljeni.
Previše	Prevelika količina podataka ili prebrzo stižu.
Nedovoljno	Previše mala količina podataka ili presporo stižu.
Delimično	Podaci ili kontrolni signal nisu potpuni.
Različito	Podaci ili kontrolni signal imaju drugu komponentu.
Prerano	Signal stiže isuviše rano u odnosu na sat.
Kasno	Signal stiže isuviše kasno u odnosu na sat.
Pre	Signal stiže isuviše rano u odnosu na očekivani redosled.
Nakon	Signal stiže isuviše kasno u odnosu na očekivani redosled.

Stabla se zatim mogu koristiti kao podrška ispravljanju grešaka ili otpornosti na greške zavisno od izabrane strategije dizajna.

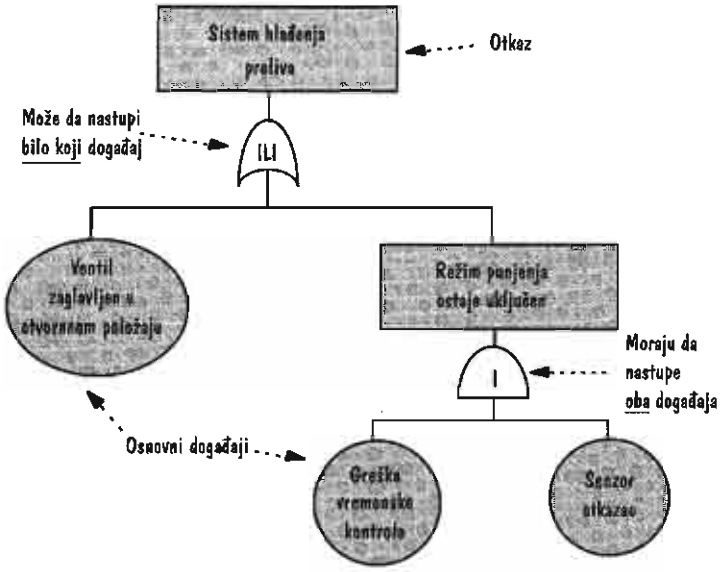
Analizu počinjemo identifikovanjem mogućih otkaza. Mada se to identifikovanje vrši tokom dizajna, mi uzimamo u obzir otkaze do kojih može da dođe zbog dizajna, u toku rada ili čak zbog održavanja. Skup ključnih reči koristimo da bismo lakše shvatili na koje sve načine sistem može da odstupa od predviđenog ponašanja. U tabeli 5.3 navedene su neke ključne reči koje mogu da se koriste; vi možete da napravite vlastitu kontrolnu listu za domen primene u kojem treba da radi vaš sistem.

Zatim pravimo dijagram čiji čvorovi predstavljaju otkaze, bilo pojedinačnih komponenti, određene funkcije sistema ili celog sistema. Linije dijagrama predstavljaju relacije između čvorova. Svaka linija je označena logičkim deskriptorom: *i* u slučaja da obe komponente moraju da otkažu, *ili* ako mora da otkaže jedna ili druga komponenta. Ponekad je linija označena sa *n_od_m* ako sistem sadrži *m* redundantnih komponenti, a do otkaza o kojem je reč dolazi ako otkaže *n* komponenti. Ključni uslov za izgradnju dijagrama je da svaki čvor predstavlja nezavisan događaj.

Kada nacrtamo dijagram, možemo da tražimo nekoliko vrsta slabih tačaka:

- singularne tačka otkaza, gde se bezbednost ili integritet sistema oslanja na jednu komponentu;
- neizvesnost, gde nema dovoljno ograničenja za vrednosti promenljivih ili uslova na koje bi se granalo;
- dvosmislenost;
- nedostaju komponente.

Na primer, razmotrimo jedan deo kontrolnog sistema elektrane prikazanog dijagramom na slici 5.19. Dijagram pokazuje da bi sistem hlađenja mogao da se prelije što bi bilo opasno otkazivanje. Do toga može doći ako se ventil koji reguliše dotok vode zaglavi u položaju „otvoren" ili ako sistem ostaje u režimu za punjenje (umesto da pređe u zatvoreni režim). Da bi do toga došlo, moraju istovremeno da nastupe dva događaja: merač vremena koji kontroliše dotok vode ne prekida a istovremeno otkaže senzor koji bi trebalo da otkrije da je sistem pun.



SLIKA 5.19 Deo kontrolnog sistema elektrane

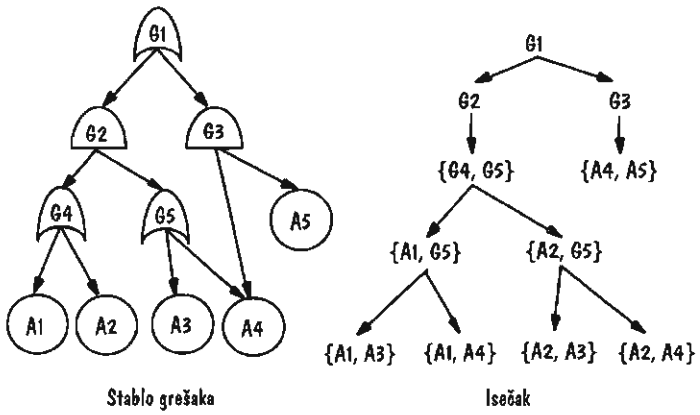
Iz ovog stabla grešaka možemo da izgradimo drugo stablo, koje je isečak datog stabla. Isečak stabla nam pomaže da pronađemo singularne tačke otkaza, pogotovo kada su stabla grešaka složena i teška za ručnu analizu. Pravila za formiranje isečka su sledeća:

1. Radeći odozgo nadole, dodeliti najviši čvor isečka prvom logičkom prolazu na vrhu stabla grešaka.
2. Ako naiđete na prolaz *ili*, podelite isečak na dva dela; ako naiđete na prolaz *i* dodajte kompozitni čvor sa podređenim čvorovima.
3. Nastavite sve dok svi čvorovi listovi ne budu osnovni događaji ili sastavni čvorovi osnovnih događaja.

Isečak stabla je skup čvornih listova iz stabla, koji nastaje kada se uklone duplikati. Na primer, razmotrite stablo grešaka na levoj strani slike 5.20. G1 je prvi logički prolaz, a njegov uslov *ili* dovodi do podele na G2 i G3 u stablu. Nadalje, G2 je sastavljen od G4 *i* G5, pa se to sastavljanje u stablu predstavlja kao {G4, G5}. Nastavljajući na ovaj način utvrđujemo da se skup sastoji od {A1, A3}, {A1, A4}, {A2, A3}, {A2, A4}, {A4, A5}. Možemo ga interpretirati tako da je jedinstvena tačka kvara {A*i*}, prouzrokovana jednim događajem, A*i*. Slično tome, otkazivanje {A*i*, A*j*} nastupa ako se dogodi i A*i* i A*j*, a {A*i*, A*j*, ..., A*n*} nastupa samo ako se dogode svi događaji koji ga čine. Prema tome, izveli smo analizu od efekta, pa do svih njegovih mogućih uzroka.

Do sada, koncepti koje smo opisali mogli su se primeniti na svaki sistem, hardverski ili softverski. Raščlanjivanje dizajna analizom stabla grešaka može za softver da bude preciznije zato što znamo da proizvoljan dizajn možemo predstaviti preko sekvence, odluke i iteracije (Böhm i Jacopini 1966). Prema tome, svaka sekvenca, odluka i iteracija može da se konvertuje u ekvivalentni isečak stabla, a konverzijom celokupnog stabla grešaka dobija se veoma velik skup isečaka. Zatim se dizajn razmatra tako što se pretpostavi

da je nastupio neki otkaz pa se pokušava pronaći skup događaja koji do njega dovode. Ako je taj skup prazan, pretpostavljamo da taj otkaz ne može da se desi.



SLIKA 5.20 Isečak stabla izveden iz stabla grešaka

Kada utvrdimo koje su tačke kvara u dizajnu, možemo da ga prepravimo i smanjimo njegovu osetljivost. Kada pronađemo grešku u dizajnu, imamo nekoliko mogućnosti:

- da je uklonimo;
- da dodamo komponente ili uslove kojima ćemo sprečiti da se izvrše ulazna stanja koja dovode do greške;
- da dodamo komponente koje će popraviti štetu u slučaju otkaza.

Mada je prva varijanta poželjna, ona nije uvek moguća.

Stabla grešaka su takođe korisna za izračunavanje verovatnoće da će nastupiti određeni otkaz. Ali, analiza pomoću stabla grešaka ima neke nedostatke. Prvo, izgradnja dijagrama može da zahteva dosta vremena. Drugo, mnogi sistemi sadrže veliki broj zavisnosti, pa je veoma teško otkriti nekonsistentnosti, a isto tako je teško usredsrediti se samo na najkritičnije delove dizajna, osim ako imamo veoma slabe sprege. Štaviše, zastrašujući su broj i raznolikost preduslova neophodnih za svaki otkaz i nije ih uvek lako uočiti, a ne postoje merenja koja bi nam pomogla da ih otkrijemo. Međutim, istraživači i dalje traže načine da se izgradnja stabala i procesi analize automatizuju. U SAD i Kanadi se analiza pomoću stabla grešaka koristi za kritične aplikacije u avio-industriji i u nuklearnim aplikacijama gde je rizik od otkaza tako veliki da se isplate intenzivni i znatni napori za izgradnju i analizu stabala grešaka.

5.7 PROCENA I PROVERA VALJANOSTI DIZAJNA

Pošto smo dizajnirali sistem, proveravamo ga na dva različita načina. Prvo proveravamo da li on zadovoljava sve zahteve koje je tražio kupac. Ovaj postupak nazivamo **provera valjanosti** dizajna. Zatim obraćamo pažnju na kvalitet dizajna. **Procena** znači proveravanje da li su uključene karakteristike dobrog dizajna kako smo ih ranije opisali. Postoje automatske alatke koje nam pomažu u oba posla. Međutim, te alatke ne mogu da obave

sav posao. Štaviše, ponekad imamo nekoliko projekata ili projektnih odluka od kojih treba izabrati jednu koja je najpoželjnija. U ovom odeljku razmotrićemo tehnike koje će nam pomoći u proceni i proveri valjanosti.

Matematička provera

Idealno bi bilo kad bismo mogli formalno da dokažemo da je dizajn ispravan u smislu da svaki proces u sistemu pravilno transformiše ulaz procesa u njegov očekivani izlaz. Neki istraživači su postavili određeni stepen matematičke strogosti za ovu vrstu provere tako što su podelili sistem na skup procesa. Svakom procesu se pridružuje skup ulaza, skup očekivanih izlaza i skup tvrdnji vezanih za proces. Zatim, za svaki takav proces demonstriramo da:

- ako je skup ulaza pravilno formulisan on se pravilno transformiše u skup očekivanih izlaza;
- proces se završava bez otkaza.

Ovaj postupak „dokazuje" da je dizajn ispravan. Međutim, ako bi se formalno koristio za dokazivanje da je svaka mala transformacija matematički ispravna, utrošilo bi se dosta vremena i sredstava, pa se ponekad primenjuje samo na najkritičnije delove sistema. Formalna notacija dizajna pomaže da postupak bude lakši, a istraživači i dalje grade alatke za njihovu podršku. Zbog toga se često koriste druge, manje formalne metode za proveru valjanosti dizajna.

Merenje kvaliteta dizajna

Neki istraživači razvijaju merenja kojima se procenjuju određeni ključni aspekti kvaliteta dizajna. Na primer, Chidamber i Kemerer (1994) su predložili opšti skup mera za objektno orijentisani dizajn, a Bieman i Ott (1993) su predložili mere kohezije za objektno orijentisane sisteme. Briand, Morasca i Basili (1994) predložili su opšte mere za dizajn visokog nivoa, uključujući koheziju i spregnutost, a Briand, Devanbu i Melo (1997) nastavljaju rad na tim idejama i predlažu načine za merenje spregnutosti.

Da biste videli kako ta merenja prikazuju informacije o dizajnu, razmotrite merenje spregnutosti za sledeću grupu. Briand i drugi (1997) primećuju da se spregnutost u dizajnu na na C++-u može zasnivati na tri različite karakteristike: relacijama (prijateljstvo, nasleđivanje ili nijedno), hodografu (da li promena teče prema klasi ili od nje) i vrstama (interakcija atributa i klase, interakcija metoda i klase ili interakcija metoda i metoda). Za svaku klasu u dizajnu oni su definisali metriku kojom se broje interakcije između neke klase i ostalih klasa ili metoda. Zatim, koristeći empirijske informacije o dizajnu i nastale sistemske greške i otkaze, analiziraju odnos između vrste sprege i vrste pronađenih grešaka. Na primer, prema njihovom izveštaju dobijeni kôd je imao više grešaka nego što je to uobičajeno ako je postojao veliki broj interaktivnih atributa između date klase i drugih klasa koje joj nisu preci, naslednici ili prijatelji. Slično, ako je mnogo eksternih metoda iz prijateljskih klasa zavisilo od metoda date klase, ta klasa je imala više grešaka. Na ovaj način se informacije o dizajnu mogu koristiti za predviđanje gde bi moglo doći do greške, a onda se tokom dizajniranja mogu preduzeti koraci i ugraditi mehanizmi za sprečavanje grešaka i stvaranje otpornosti na greške.

Card i Glass (1990) opisuju načine za merenje složenosti softverskog dizajna. Polazeći od toga da je cilj pouzdano predviđanje grešaka, oni pominju knjigu Carda, Churcha i

Agrestija (1986) u kojoj je utvrđeno da distribucija grešaka ne zavisi od vrste spregnutosti u dizajnu, a da su komponente većeg raspona kontrole obično skuplje i imaju veći procenat grešaka. Na osnovu subjektivno procenjene kohezije komponenti takođe su utvrdili da kohezivniji moduli imaju manji procenat grešaka i da su jeftiniji za proizvodnju.

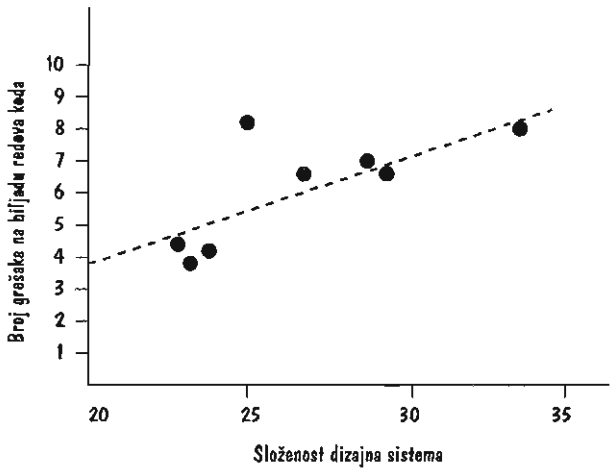
Card i Glass (1990) naglašavaju da složenost dizajna u suštini ima dva aspekta: složenost unutar svake komponente i složenost odnosa među komponentama. Oni definišu meru složenosti sistema koja obuhvata oba aspekta i opisuju kako su usavršili tu definiciju tokom testiranja sa podacima iz stvarnih projekata. Njihova konačna mera složenosti jeste zbir

$$C = S + D$$

gde je

$$S = (1/n)\sum f^2(i)$$
$$D = V(i)/[f(i) + 1]$$

S predstavlja strukturnu složenost (veze među komponentama), D predstavlja složenost podataka (unutar komponenti), $f(i)$ je izlazni faktor za komponentu i, $V(i)$ je broj ulaznih i izlaznih promenljivih komponente i, a n je broj komponenti. Odnos između C i broja grešaka na hiljadu redova koda u podacima sa kojima su eksperimentisali prikazan je na slici 5.21; statistika je pokazala da je složenost dizajna prikazana ovim merenjem bila odgovorna za 69% varijacije u procentu grešaka, a da je svaki porast složenosti za jednu jedinicu povećavao procenat grešaka za 0,4 greške na hiljadu redova koda.



SLIKA 5.21 Grafikon procenta grešaka prema složenosti sistemskog dizajna (Card i Glass 1990)

Važno je imati na umu da su ove mere izvedene iz konkretnih razvojnih programa i da se na vaše projekte može preneti ta tehnika, ali se ne mogu preneti i tačni brojevi i relacije. To jest, možete da koristite regresiju nad podacima iz ranijih projekata da biste napravili formule koje opisuju odnose među promenljivim, ali vaš obrazac će se možda razlikovati od onog koji je gore naveden.

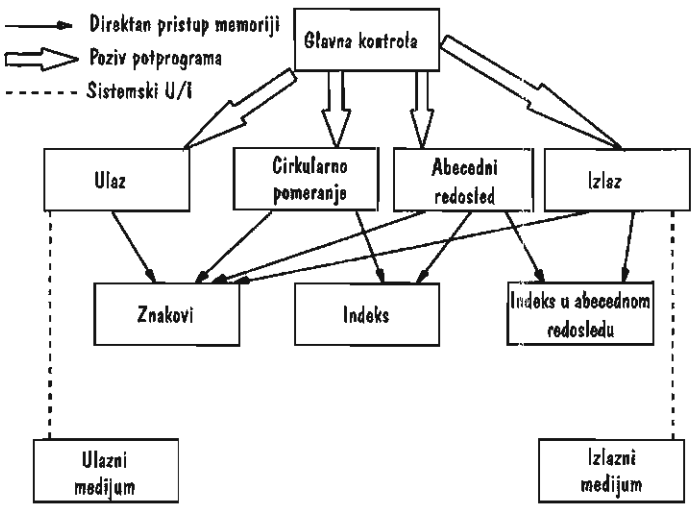
Poređenje dizajna

Merenje različitih atributa dizajna nije korisno samo za predviđanje grešaka već takođe i za poređenje dva dizajna. Na primer, možda smo unapredili dizajn da bi bio otporniji na greške ili smo ispravili problem koji smo otkrili analizom stabla grešaka. Da je novi dizajn zaista bolji od starog, može nas uveriti poređenje zasnovano na merenju. Osim toga, kada napravimo više projekata za istu specifikaciju prema različitim arhitektonskim stilovima, treba da odlučimo koji dizajn je primereniji ciljevima našeg sistema.

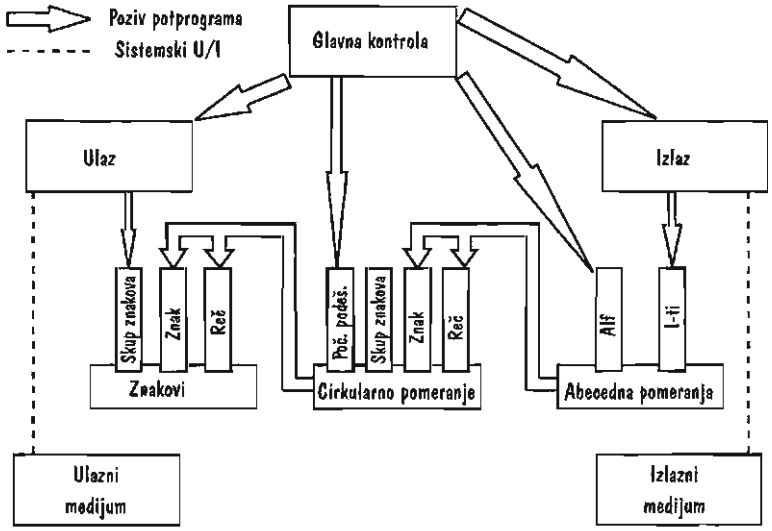
Jedna specifikacija, mnogo projekata. Da biste videli kako različiti stilovi dizajna mogu da se upotrebe za rešavanje istog problema, razmotrite problem koji je postavljen u Parnasovoj (1972) knjizi:

> Indeksni sistem KWIC [key word in context] prihvata uređen skup redova; svaki red je uređen skup reči, a svaka reč je uređen skup znakova. Svaki red sme da se „cirkularno pomera" tako što se više puta uklanja prva reč koja se zatim doda na kraj reda. Indeksni sistem KWIC kao rezultat daje spisak svih cirkularnih pomeranja iz svih redova u abecednom redosledu.

Shaw i Garlan (1996) predstavljaju četiri različita arhitektonska dizajna za implementaciju indeksnog sistema KWIC: zajednički podaci, apstraktni tipovi podataka, implicitno pozivanje i cevi i filtri. Rešenje sa zajedničkim podacima, prikazano na slici 5.22 deli problem na četiri funkcionalna dela: ulaz, cirkularno pomeranje, abecedni redosled i izlaz. Koordinacija je poverena glavnom programu koji poziva te delove po redu. Podaci se nalaze na centralnoj lokaciji, a izračunavanje je podeljeno na komponente. Međutim, Parnas ističe da se ovakav dizajn teško menja zato što promena podataka utiče na sve komponente i zato što dizajn nije pogodan za višekratnu upotrebu.

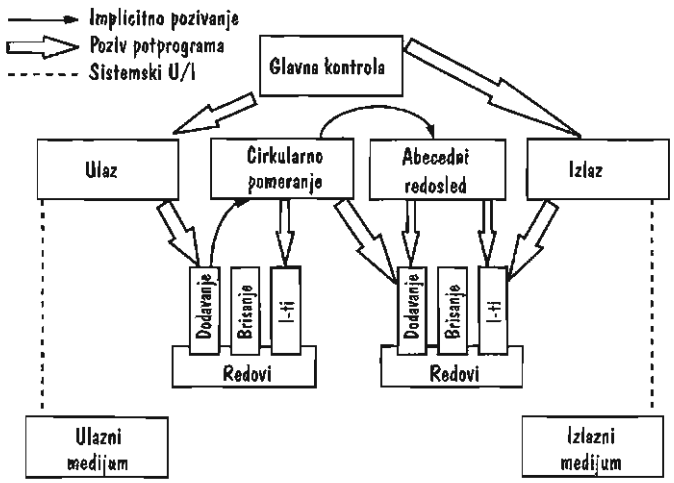


SLIKA 5.22 Rešenje za KWIC pomoću zajedničkih podataka (Shaw i Garlan 1996)

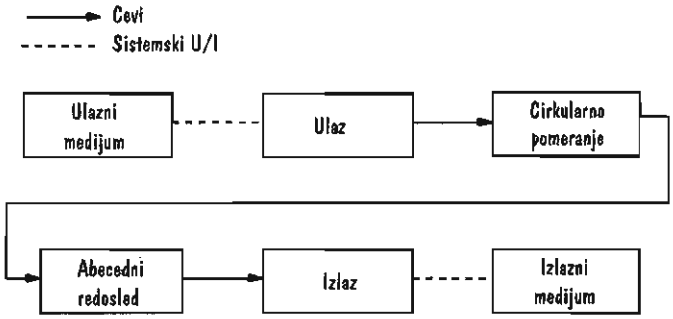


SLIKA 5.23 Rešenje za KWIC pomoću apstraktnih tipova podataka (Shaw i Garlan 1996)

Slika 5.23 ilustruje drugačiji pristup dizajnu. Ovde podaci više nisu u centralnom spremištu i nisu zajednički, ali razlaganje obrade je isto. Podaci i algoritmi se lakše menjaju nego u prethodnom rešenju, a lakši su i za višekratnu upotrebu jer među komponentama ima manje zavisnosti. Međutim, u ovaj dizajn bi bilo teže dodavati nove funkcije. (Detaljan opis ovog dizajna može se naći u Garlan, Kaiser i Notkin [1992].)



SLIKA 5.24 Rešenje za KWIC pomoću implicitnog pozivanja (Shaw i Garlan 1996)



SLIKA 5.25 Rešenje za KWIC pomoću cevi i filtara (Shaw i Garlan 1996)

Na slici 5.24 prikazano je još jedno rešenje sa zajedničkim podacima, ali ovog puta se interfejs prema podacima veoma razlikuje. U prvom dizajnu je svaka komponenta imala direktan pristup formatima podataka; ovde se podacima pristupa apstraktno. Štaviše, izračunavanje se vrši prilikom menjanja podataka, a ne eksplicitno kao u prvom dizajnu. Ovaj pristup obezbeđuje jednostavnije poboljšanje funkcionalnosti, budući da potrebu za njom signaliziraju događaji koji vrše izmene nad podacima. Osim toga, pošto se podacima rukuje apstraktno, promene u predstavljanju podataka neće uticati na deo koji se bavi izračunavanjem. Zbog tih svojstava je ovaj dizajn jednostavniji za višekratno korišćenje od prethodna dva, jer se sve aktivnosti oslanjaju na okidače događaja.

S druge strane, nije lako kontrolisati redosled u kojem se vrši obrada, te zato posmatramo rešenje sa cevima i filtrima (slika 5.25) gde se redosled obrade kontroliše redosledom filtara. Pošto se svaki filtar izvršava nezavisno, ovaj dizajn je veoma pogodan za ponovno korišćenje, a svaki filtar ili cev se veoma lako menja. Međutim, bilo bi teško primeniti ovaj dizajn kod interaktivnih sistema.

Vidimo da svaki dizajn ima svoje prednosti i mane. Zbog toga nam je potreban metod za poređenje različitih dizajna kako bismo izabrali najbolji za našu namenu.

Poredbene tabele. Shaw i Garlan (1996) upoređuju naša četiri dizajna pomoću tabele značajnih atributa, koja je prikazana kao tabela 5.4. Svaki red predstavlja jedan atribut, a za svaki stil dizajna imamo po jednu kolonu. Minus u ćeliji znači da dizajn čija je to kolona nema atribut čiji je to red; plus znači da dizajn poseduje taj atribut. Iz tabele vidimo da nije jasno koji dizajn bi trebalo izabrati; moramo atributima da dodelimo prioritete i da obrazujemo vrednosne ocene da bismo izabrali najbolji dizajn za naše potrebe.

Tabela 5.4 Poređenje predloženih rešenja Shaw i Garlan (1996)

	Zajednički podaci	Apstraktni tipovi podataka	Implicitno pozivanje	Cevi i filtri
Lakoća promene algoritma	-	-	+	+
Lakoća promene predstavljanja podataka	-	+	-	-
Lakoća promene funkcije	+	-	+	+
Dobre performanse	+	+	-	-
Lako višekratno korišćenje	-	+	-	+

Evo još nekih karakteristika koje smo mogli da razmatramo:

- modularnost;
- lakoća testiranja;
- lakoća održavanja;
- efikasnost;
- razumljivost;
- lakoća promene;
- konsistentnost.

Pošto sastavimo potpun spisak atributa koji su nam važni, svakome od njih pridružujemo stepen značaja tako što mu dodeljujemo težinu u skladu sa njegovim prioritetom. Na primer, ako dizajn treba ponovo da se upotrebi u nekoliko drugih proizvoda, mogli bismo atributu višestruke upotrebljivosti dodeliti vrednost „5" (u rasponu od 1 do 5, gde je 5 najpoželjnije - najveći prioritet).

Zatim obrazujemo matricu prikazanu u tabeli 5.5 sa istim nazivima kolona i redova kao u tabeli 5.4. Prva kolona u matrici sadrži pondere koje smo koristili u prethodnom koraku. U drugoj koloni navedeni su prioriteti koje smo dodelili svakom atributu. U preostalim kolonama ocenjujemo svaki dizajn prema navedenim kriterijumima. Ako imamo n dizajna koje treba porediti, njima odgovaraju kolone od 3 do n+2. U te kolone stavljamo ocene od 1 do 5 tako da sadržaj ćelije u i-tom redu i j-toj koloni označava do koje mere dizajn iz kolone j zadovoljava karakteristiku iz reda i.

Na kraju, izračunavamo ocenu svakog dizajna tako što množimo prioritet svakog reda sa ocenom atributa i sabiramo te proizvode za svaki dizajn. Na primer, ocenu za dizajn sa cevima i filtrima izračunali bismo kao 1 x 5 + 4 x 1 + 3 x 5 + 3 x 2 + 5 x 5 = 55. Prema tome, konačne ocene za naša četiri dizajna na osnovu dodeljenih ocena i prioriteta su:

Zajednički podaci	37
Apstraktni tipovi podataka	57
Implicitno pozivanje	40
Cevi i filtri	55

Tabela 5.5 Poređenje dizajna prema vrednosti poređenih komponenti, Shaw i Garlan (1996)

Atributi	Prioritet	Zajednički podaci	Apstraktni tipovi podataka	Implicitno pozivanje	Cevi i filtri
Lakoća promene algoritma	1	1	2	4	5
Lakoća promene predstavljanja podataka	4	1	5	2	1
Lakoća promene funkcije	3	4	1	4	5
Dobre performanse	3	5	4	2	2
Lako višekratno korišćenje	5	1	4	2	5

U ovom slučaju, izabrali bismo dizajn apstraktnih tipova podataka. Imajte na umu da su prioriteti i ocene, kao i izbor atributa, subjektivni, i da zavise od potreba naših kupaca i korisnika, kao i od naših sklonosti u izgradnji i održavanju sistema. Drugi procenitelji bi verovatno došli do drugačijih ocena, pa bi zato doneli drugačije odluke. Kada budemo

naučili više o merenju atributa dizajna i procenjivanju njihovih odnosa sa drugim karakteristikama sistema (na primer, sa cenom ili sklonošću ka greškama) moći ćemo iz ovog ocenjivanja da isključimo jedan deo subjektivnosti. Ali pošto svako od nas ima različite potrebe i shvatanja, u procenama će uvek postojati određena mera subjektivnosti.

Konačna revizija dizajna

Kad je dizajn kompletan, sastajemo se sa svojim kupcima i pregledamo ga pre nego što nastavimo sa razvojem. Proces revizije se izvršava u tri koraka, koji odgovaraju koracima u procesu projektovanja. Prvo, održavamo **preliminarnu reviziju dizajna** na kojoj razmatramo konceptualni dizajn sa kupcima i korisnicima. Zatim u **kritičkoj reviziji dizajna** prikazujemo tehnički dizajn drugim projektantima da bismo proverili detalje pre nego što pređemo na implementaciju. Na kraju, održavamo **programsku reviziju dizajna** da bi programeri dobili povratne informacije o svom dizajnu pre nego što počne implementacija. Konačni cilj svakog dela u postupku revizije je isti: provera da li gradimo ono što kupci žele.

Preliminarna revizija dizajna. Na preliminarnoj reviziji dizajna sastajemo se sa svojim kupcima i korisnicima da bismo proverili valjanost konceptualnog dizajna. To jest, želimo da proverimo da li su svi aspekti zahteva rešeni u našem dizajnu. Da bismo to postigli, na reviziju pozivamo nekoliko ključnih ljudi:

- kupce koji su pomogli u definisanju sistemskih zahteva;
- analitičare koji su pomogli u definisanju sistemskih zahteva;
- planirane korisnike sistema;
- dizajnere sistema;
- moderatora;
- sekretara;
- druge zainteresovane projektante sistema koji inače nisu uključeni u ovaj projekat.

Broj ljudi prisutnih na ovoj reviziji zavisi od veličine i složenosti sistema koji se gradi, i od broja i vrste njegovih korisnika. Svaki učesnik revizije trebalo bi da ima ovlašćenje svoje organizacije da donosi odluke i prihvata obaveze. Ukupan broj učesnika bi trebalo da bude što manji da bi se olakšala rasprava i donošenje odluka.

Moderator vodi raspravu, ali nije zainteresovana strana u samom projektu. On podstiče raspravu, igra ulogu posrednika u slučaju suprotstavljenih mišljenja, pazi da rasprava ne zamre i da ostane objektivna i uravnotežena.

Pošto je teško učestvovati u raspravi i istovremeno beležiti glavne argumente i zaključke, potrebno je nekoga zadužiti da bude sekretar. Sekretar se ne uključuje u raspravu; njegova jedina uloga je da vodi zapisnik. Međutim, znanje stenografije nije dovoljno; sekretar mora da ima dovoljno tehničkog znanja da razume raspravu kako bi zapisao sve relevantne tehničke informacije. U suštini, sekretar često zahteva od izlagača da razjasne suštinu da bi se ona zabeležila na razumljiv način.

Projektanti koji nisu uključeni u projekat obezbeđuju uvid sa aspekta osobe koja nije direktno zainteresovana. Oni mogu da budu objektivni u komentarisanju predloženog

dizajna, jer nisu lično uključeni u njega. U suštini, oni mogu da imaju nove ideje i da ponude nove smerove. Oni takođe igraju ulogu minijaturnog tima za kontrolu kvaliteta, jer proveravaju da li je sve preduzeto da se postignu ispravnost, konzistentnost i karakteristike dobrog dizajna. Svojim učešćem u reviziji oni preuzimaju jednaku odgovornost za dizajn, kao i sami dizajneri. Ta zajednička odgovornost primorava sve prisutne da pažljivo razmotre svaki detalj.

Tokom ove revizije mi svojim slušaocima prikazujemo konceptualni dizajn. Tom prilikom pokazujemo da sistem ima traženu strukturu, funkcionalnost i karakteristike navedene u dokumentima zahteva. Svi zajedno proveravamo da li predloženi dizajn sadrži potreban hardver, interfejse sa drugim sistemima, ulaz i izlaz. Kupac odobrava dijaloge i menije, formate izveštaja i predloženu obradu grešaka. Ako sistem treba da se gradi u fazama, sada opisujemo karakteristike i funkcionalnost svake faze.

Sekretar treba da zabeleži svako odstupanje, a cela grupa da raspravlja o njemu. Manja pitanja rešavaju se na licu mesta. Međutim, ako se naiđe na veće greške i nesporazume, možemo da pristanemo na revidiranje dizajna. U tom slučaju zakazujemo novu preliminarnu reviziju dizajna za procenu novog dizajna. Isto kao što bi porodica Hauel radije prepravila planove za kuću nego da kasnije ruši zidove i temelje, i mi ćemo radije promeniti dizajn sistema u ovoj fazi nego kasnije.

Kritička revizija dizajna. Kada je kupac zadovoljan predloženim proizvodom, vreme je da se održi kritička revizija dizajna na kojoj izlažemo opšti pregled tehničkog dizajna. Učesnici ove revizije su:

- analitičari koji su pomogli u definisanju sistemskih zahteva;
- dizajneri sistema;
- moderator;
- sekretar;
- dizajneri programa za ovaj projekat;
- predstavnik grupe koja će testirati sistem;
- analitičar koji će pisati sistemsku dokumentaciju;
- drugi zainteresovani projektanti sistema koji inače nisu uključeni u ovaj projekat.

Primetićete da je u ovoj grupi više tehničkog osoblja nego u prethodnoj, pošto se kritička revizija dizajna odnosi na tehničke detalje u dizajnu. Kao i ranije, moderator kontroliše tok rasprave i obezbeđuje da ona ostane usredsređena na dva glavna pitanja: Da li su u dizajnu implementirani svi zahtevi? Da li dizajn ima visok kvalitet? Dizajneri programa ne prisustvuju samo da bi kritikovali dizajn, već takođe da bi ga shvatili jer će iz njega izvoditi detaljnije programske dizajne.

Alternativne strategije dizajna objašnjavamo pomoću dijagrama i podataka (zajedno ili odvojeno), kao i načine odlučivanja i razloge zbog kojih smo doneli glavne odluke u dizajnu. Ako smo koristili dizajnerske alatke, njihovi rezultati se daju timu na razmatranje. Kao i ranije, ako se naiđe na veće probleme, dizajn se revidira, a po potrebi se zakazuje nova kritička revizija dizajna, a možda i nova preliminarna i kritička revizija.

Programska revizija dizajna. Kada smo zadovoljni tehničkim dizajnom, dizajneri programa ga pretvaraju u skup zapisa za pojedinačne komponente koje treba da se kodiraju i testiraju. Kada se završi dizajn programa, ali pre nego što počne kodiranje, dizajneri programa predstavljaju svoje planove timu ostalih dizajnera, analitičara i programera kako bi dobili komentare i predloge. Tim koji učestvuje na reviziji sastoji se od:

- analitičara koji su napravili sistemske zahteve;
- dizajnera sistema;
- dizajnera programa;
- predstavnika grupe koja će testirati sistem;
- analitičara koji će pisati sistemsku dokumentaciju;
- razvijalaca;
- moderatora;
- sekretara;
- drugih zainteresovanih projektanata sistema, koji inače nisu uključeni u ovaj projekat.

Kao i ranije, broj prisutnih bi trebalo da odgovara veličini i složenosti projekta, a učesnici bi trebalo da budu ovlašćeni za donošenje obavezujućih odluka. Moderator usmerava i podstiče raspravu, a sekretar dokumentuje tehničku raspravu i konačne odluke. Spoljni posmatrači i dalje obezbeđuju objektivnost revizije pošto analiziraju dizajn i doprinose kontroli kvaliteta.

Značaj revizija dizajna. U svakoj vrsti revizije dizajna prisutni moraju da postave nekoliko važnih pitanja, uključujući i sledeće:

1. Da li dizajn predstavlja rešenje problema?
2. Da li je dizajn modularan, dobro strukturisan i lako razumljiv?
3. Da li se može nešto učiniti da bi se poboljšala struktura i razumljivost dizajna?
4. Da li se taj dizajn može prenositi na druge platforme?
5. Da li je dizajn ponovno iskoristiv?
6. Da li je dizajn lako menjati i proširivati?
7. Da li dizajn može lako da se testira?
8. Da li dizajn obezbeđuje maksimalne performanse tamo gde to odgovara?
9. Da li dizajn koristi komponente iz drugih projekata tamo gde to odgovara?
10. Da li algoritmi odgovaraju ili se mogu poboljšati?
11. Ako ovaj sistem treba razvijati u fazama, da li su faze tako povezane da se olakša prelazak iz jedne faze u sledeću?
12. Da li je dizajn dobro dokumentovan uključujući projektantske odluke i njihovo obrazloženje?
13. Da li dizajn sadrži unakrsne reference komponenti i podataka sa zahtevima?
14. Da li dizajn koristi odgovarajuće tehnike za obradu grešaka i sprečavanje otkazivanja?

U revidiranju dizajna bavimo se otkrivanjem grešaka, a ne njihovim ispravljanjem. Greške se ne odnose samo na to kako dizajn implementira trenutne zahteve, već takođe

i na stepen do kojeg dizajn podnosi promene. To jest, kao što je naglašeno u šestom pitanju, važno je predvideti menjanje i koristiti dizajn koji će obezbediti relativno bezbolne promene, proširivanje i razvoj. U podsetniku 5.6 opisano je kako prerađivanje (brzi metod za reviziju dizajna) može da bude zaista bolno.

Tokom revizije dizajna važno je imati na umu da učesnici istražuju integritet dizajna, a ne dizajnera. Prema tome, značajno je u reviziji naglasiti svim učesnicima da svi imamo isti cilj. Kritike i rasprave tokom revizije ne smeju biti lične i komentari treba da budu usmereni na proces i proizvod, a ne na učesnike. Postupak revizije ohrabruje i jača komunikaciju među članovima tima.

PODSETNIK 5.6 MUKE SA PRERAĐIVANJEM POSTOJEĆEG DIZAJNA

Berry (2002a) primećuje da softverski inženjeri često prvo urade lakše delove i ostavljaju teže probleme za kasnije. Analizirajući ovu sklonost među inženjerima koji koriste brze metode, on ističe da X-treme programeri uče kako ne treba da brinu o eventualnim novim zahtevima sve dok se oni zaista ne pojave. Zatim, kada se zahtevi pojave, dizajn sistema se preraduje: „razmotri se ceo novi skup zahteva i traži nova najjednostavnija arhitektura koja će ih kompletno zadovoljiti". Posto se otkrije ta arhitektura, postojeći kôd se restrukturiše u skladu sa novom arhitekturom. Teoretski, ako su dizajn i kôd modularni, prerađivanje bi trebalo da bude relativno bezbolno. Ali, u praksi ono može biti izuzetno teško.

Kada se nametnu novi zahtevi ili se stari promene, razvoj mora da stane dok arhitekte sistema razmatraju kako da se izbace delovi postojećeg dizajna i koda koji možda savršeno rade i ne napišu se novi programi zbog nastalih promena. Razvijaoci obično rade pod pritiskom strogih vremenskih ograničenja. Elssamadissy i Schalliol (2002) ističu da ta vremenska ograničenja često znače da se u praksi prerađivanje više puta odlaže zato da bi niz izdanja bio završen u planiranim rokovima. Rezultat je, na žalost, da se arhitektura sve više udaljava od novih zahteva. Oni su zaključili da „ako od početka idemo linijom manjeg otpora i zanemarujemo signale, dovodimo sebe u bezizlaznu situaciju". Naravoučenine ove price je sledeće: kada se nađete pred obimnim refaktorisanjem, videćete da ste dotle došli verovatno zato što ste preskočili čitav niz manjih refaktorisanja.

Berry (2002a) smatra da će pristalice brzih metoda odgovoriti da se prvo definišu i projektuju najvažniji zahtevi i da zbog toga problem refaktorisanja nije tako veliki kako ga prikazuju Elssamadissy i Schalliol. Ali, u suštini, u bilo kojem značajnijem sistemu teško je, a ponekad i nemoguće predvideti koji će zahtevi u budućnosti biti najvažniji. Berry kaže da „za svaki novi problem gde je uzbuđenje pred inovacijama [inspiracija za razvijaoce], ne možemo se nadati da ćemo izbeći nemilosrdne promene kad se budemo bolje upoznali sa područjem primene, potrebom za savršenstvom i mukama".

Štaviše, proces je svima koristan ako se greške i problemi pronađu dok je njihovo ispravljanje još lako i jeftino. Daleko je lakše promeniti nešto u apstraktnoj konceptualnoj fazi nego kasnije kada se već implementira. Veliki deo teškoća i troškova kada se greške ispravljaju u kasnijim fazama razvoja odnosi se na traženje uzroka greške. Ako se greška otkrije prilikom revizije dizajna, znaćemo da se problem nalazi negde u dizajnu. Međutim, ako se greška otkrije tek kada sistem već radi, izvor problema može da bude: u hardveru, u softveru, u dizajnu, u implementaciji ili u dokumentaciji. Što pre pronađemo problem, njegov uzrok moramo da tražimo na manje mesta da bismo ga ispravili.

5.8 DOKUMENTOVANJE DIZAJNA

Značajan proizvod procesa projektovanja je skup dokumenata koji opisuju sistem koji treba izgraditi. Kao što smo videli, jedan deo treba da objasni kupcima i korisnicima u prirodnom jeziku šta će sistem da radi; u drugom delu se tehničkom terminologijom opisuju struktura sistema, podaci i funkcije. Prema tome, sadržaj ta dva dela može da se preklapa, ali ne i način na koji se on izlaže.

Dokumentacija dizajna bi trebalo da sadrži i **obrazloženje dizajna**, odnosno kratak pregled najvažnijih pitanja i dilema koje su razmatrane u fazi dizajna. Ako se u izradi dokumentacije rukovodi ovakvom filozofijom, kupci i ostali projektanti će lakše razumeti kako i zašto se određeni delovi dizajna uklapaju.

Dizajn takođe sadrži opise komponenti sistema. Jedan odeljak bi trebalo da opiše kako korisnici upotrebljavaju sistem uključujući i sledeće:

- formate menija i ostale prikaze na ekranu;
- spregu sa čovekom: funkcijske tastere, opise dodirnih ekrana, rasporede tastature i upotrebu miša ili džojstika;
- formate izveštaja;
- ulaz: odakle podaci stižu, kako su formatirani i na kakvim medijima se čuvaju;
- izlaz: gde se podaci šalju, kako su formatirani i na kakvim medijima se čuvaju;
- opšte funkcionalne karakteristike;
- ograničenja performansi;
- procedure arhiviranja;
- pristup obradi grešaka.

Sveukupna organizacija i struktura sistema obično se opisuje kompletom dijagrama ili formalnim jezikom, uključujući sve nivoe apstrakcije.

Ako je sistem distribuiran, dizajn mora biti dovoljno detaljan da prikaže topologiju mreže, način na koji čvorovi mreže pristupaju jedni drugima i alokacije funkcija u čvorovima. Ako sistemski zahtevi sadrže vremenska ograničenja ili je potrebno sinhronizovanje čvorova u mreži, dizajn treba da opiše vremensko usklađivanje. Slično tome, dizajn treba da sadrži informacije o porukama za kontrolu i usmeravanje. On takođe može da obuhvati propise za integritet mreže: kako se obezbeđuje da podaci budu tačni i da se mogu povratiti nakon otkazivanja.

Ako to kupac zahteva, dizajn može da sadrži elemente za nadgledanje performansi sistema. Osim toga, može da postoji sistem ručne obrade za slučaj otkazivanja sistema i u dizajnu se opisuje kako će to da se radi. Odeljci u dokumentu o dizajnu mogu da budu

posvećeni pronalaženju i izolaciji grešaka, ponovnom konfigurisanju sistema ili posebnim merama bezbednosti.

Na kraju, daju se ukrštanja dizajna sa zahtevima da bi se prikazalo kako je dizajn izveden iz njih. Ovaj deo nas primorava da proverimo sveobuhvatnost i konzistentnost. Osim toga, ova ukrštanja će kasnije uveliko olakšati praćenje unapređenja i promena. Na primer, ako se promeni neki od zahteva, ukrštanja ukazuju na odgovarajuće potrebne promene u dizajnu.

5.9 PRIMER IZ INFORMACIONIH SISTEMA

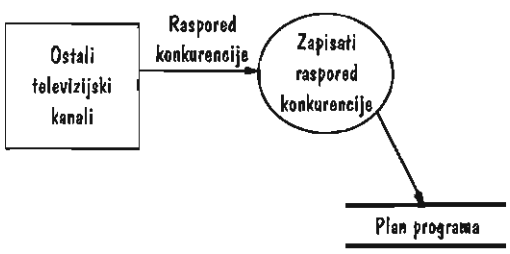
Dizajn može da se dokumentuje na različite načine. Možemo da koristimo formalne jezike, konačne automate, dijagrame tokova podataka, rečnike podataka, objektno orijentisani pristup ili neke od mnogih raspoloživih notacija i tehnika. Važno je da izaberete tehniku ili notaciju koja će najbolje odgovarati vašem dizajnu i ciljevima sistema koji razvijate. Na primer, ako projektujete sistem koji će se višekratno koristiti ili će se u periodu korišćenja često menjati, potrebno je da izaberete notaciju ili tehniku koja će većini dizajnera biti lako razumljiva.

Pogledajmo ponovo sistem TV stanice Piccadilly da bismo videli kako može da se upotrebi kombinacija dizajnerskih tehnika. Na slici 5.26 prikazan je deo specifikacije zahteva, u kome se ukazuje na potrebu da se prati ponuda konkurentskih televizijskih stanica u terminima prikazivanja programa za koje Piccadilly prodaje oglasno vreme. Taj događaj prikazan je dijagramom toka podataka i lako se vidi da će se u planu programa za Piccadilly evidentirati raspored konkurentskih programa.

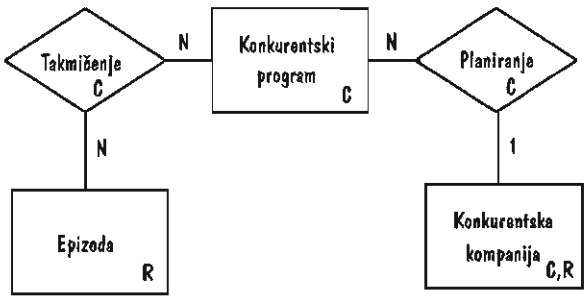
Robertson i Robertson (1994) daju nam mnogo informacija o ovom sistemu, uključujući i opis elemenata podataka u rečniku podataka. Raspored konkurencije se u rečniku zapisuje na sledeći način:

```
Raspored konkurencije = * tok podataka *
  Ime televizijske kompanije
  + {datum emitovanja konkurencije
     + vreme emitovanja konkurencije
     + ime konkurentskog programa
     + (predvidena procena konkurencije)}
```

Vitičaste zagrade su znak da se njihov sadržaj može ponavljati više puta, a mala zagrada označava da element nije obavezan - možda nemamo procenjene ocene za sve emisije konkurencije.



SLIKA 5.26 Početni model za praćenje rasporeda konkurencije (Robertson i Robertson 1994).



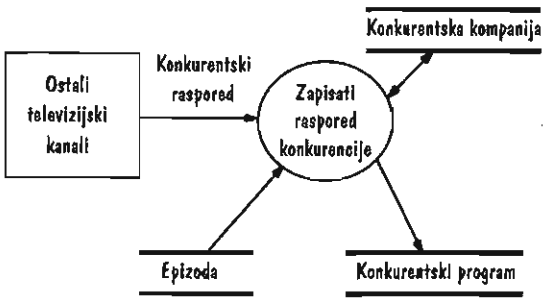
SLIKA 5.27 Model podataka za praćenje rasporeda konkurencije (Robertson i Robertson 1994).

Kad iz ovog zahteva dizajniramo sistem, posmatramo konkurentske televizijske kompanije i označavamo ih u dizajnu. Svaka konkurentska kompanija proizvodi primerke konkurentskih programa, pa njihove relacije upisujemo u model podataka. Za svaku emisiju postoje vreme i datum prema specifikaciji zahteva; ta informacija omogućava našem sistemu da pronađe ponudu stanice Piccadilly koja će se prikazivati u isto vreme. Svaki primerak programa Piccadilly nazivamo epizodom. Kompletan model podataka za ovaj događaj prikazan je na slici 5.27, gde se u svakom okviru nalazi beleška o očekivanoj vrsti pristupa (Napraviti, Pronaći, Ažurirati ili Brisati). Znak N pored linije označava da između okvira može da prelazi više primeraka podataka, a 1 označava da postoji samo jedan primerak. (U knjizi Robertson i Robertson [1994] naći ćete više detalja o notaciji, sistemu i pristupu dizajnu.)

Model prvobitnih zahteva možemo da ažuriramo crtanjem opisa događaja na visokom nivou kao što je prikazano na slici 5.28. Osim toga, crtežu možemo da dodamo opis na prirodnom jeziku onoga što želimo da softver uradi:

Ulaz: *konkurentski raspored*
Za svako *ime televizijske kompanije,* napraviti *konkurentska kompanija.*
  Za svaki *konkurentski raspored,*
    Pronaći *Epizodu* gde je *planirani datum epizode = datum emitovanja konkurencije* I *početno vreme epizode = vreme emitovanja konkurencije*
    Napraviti primerak *konkurentskog programa*
  Napraviti relacije *planiranje* i *takmičenje*
Izlaz: spisak *konkurentskih programa*

Ovaj opis je svakako na visokom nivou, pa su potrebni detaljniji nivoi apstrakcije da bi bilo sasvim jasno šta programi treba da rade. Na primer, dizajnu su potrebne informacije o tome kada se programi emituju u slično, ali ne u sasvim isto vreme. (Program stanice Piccadilly od 09.00 do 10.30 mogao bi da se preklapa sa nekim drugim programom koji ide od 08.00 do 10.00 ili, na primer, od 10.00 do 11.00-) Moramo takođe da odlučimo šta treba da se radi u slučaju greške; u ovom slučaju, *raspored konkurencije* možda sadrži neispravan datum, kao što je 31. januar ili neispravno vreme kao što je 29.00 časova. Najniži nivo detalja trebalo bi da sadrži opise komponenti tako da se pojedinačne komponente mogu dodeliti programerima na kodiranje i testiranje. I pored toga, ovaj primer nam pokazuje da već na početku procesa dizajniranja možemo da koristimo komplementarne tehnike dizajna, pa čak i tehnike za proširivanje specifikacije zahteva.



SLIKA 5.28 Model bitnih procesa za praćenje rasporeda konkurencije (Robertson i Robertson 1994).

5.10 PRIMER IZ PRAKSE

U ovom poglavlju smo učili o obradi izuzetaka i o potrebi da se u dizajnu obrati pažnja na izuzetne uslove. Komisija koja je istraživala otkazivanje programa Ariane-5 utvrdila je da „sve konverzije [u odnosu na Ariane-4] nisu bile zaštićene jer se za računar SRI zahtevalo maksimalno ciljno opterećenje od 80%" (Lions i dr. 1996). Drugim rečima, zbog zahteva performansi su projektanti Ariane-5 odlučili da izostave kôd kojim se obrađuju neki od mogućih izuzetaka iz Ariane-4. Ova vrsta analize obično se obavlja tokom procesa dizajna; često se za ispunjenje nefunkcionalnih zahteva pribegava kompromisima ne samo radi performanse već i da bi se uštedelo vreme za kodiranje i testiranje.

„Da bi se odredila ranjivost nezaštićenog koda, izvršena je analiza svake operacije koja bi mogla da dovede do... operandske greške. Zato je dodata zaštita za četiri [od sedam] promenljive... Međutim, tri promenljive su ostale nezaštićene" (Lions i dr. 1996). Dizajneri su smatrali da u nekim slučajevima ne može doći do prekoračenja. Njihova pogrešna analiza zasnivala se na pretpostavkama iz Ariane-4, a ne iz Ariane-5; u stvari, parametri putanje za Ariane-5 bili su različiti i zaista je došlo do prekoračenja, pa je raketa uništena.

Jézéquel i Meyer (1997) smatraju da su se dizajnom po ugovoru problemi Ariane-5 mogli predvideti u ranim fazama razvojnog procesa. Oni ističu da za komponentu koja je ponovo iskorišćena iz Ariane-4 nije postojala precizna specifikacija. Nejasan deo izjave o misiji sistema glasio je da bi promenljive koje predstavljaju horizontalni nagib trebalo da stanu u 16 bita, ali u kodu se taj uslov nije ispitivao. Da je taj uslov eksplicitno postavljen, mogao bi da izgleda ovako:

```
convert (horizontal_bias: DOUBLE): INTEGER is
 require
       horizontal_bias <= Maximum_bias
 do
       ...
   ensure
       ...
end
```

Obratite pažnju na to da se u preduslovu *require* tačno navodi koji uslov ulaz mora da zadovolji.

Uglavnom, tvrdnja iz ugovora bi se tokom testiranja automatski ispitivala i greška bi bila otkrivena pre nego što je Ariana poletela. Ili, da se uslov kontrolisao tokom izvršava-

nja, kršenje uslova bi ispalilo izuzetak. Najbolje je dokumentovanje ove tvrdnje uključiti u dizajn po ugovoru ili u neki drugi oblik dizajna. Tada ona prolazi kroz temeljnu reviziju dizajna, a ocenjivači bi tom prilikom verovatno pažljivo proverili tvrdnje i proverili da li je svaka od njih obrađena u potpunosti. U ovom zadnjem slučaju problem bi bio otkriven u najranije moguće vreme, pa bi se brzo i najjeftinije moguće ispravio, i ne bi bio ugrađen u implementaciju.

5.11 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju pogledali smo šta znači projektovati sistem. Videli smo da dizajn počinje na visokom nivou sa važnim odlukama o arhitekturi sistema na osnovu sistemskih zahteva, poželjnih atributa dizajna i nameravanog dugoročnog korišćenja sistema. Dok dizajnirate, trebalo bi da imate na umu nekoliko karakteristika: modularnost, stepen apstrakcije, spregnutost, koherentnost, otpornost na greške, prototipove i dizajn korisničkog interfejsa. Merenja mogu biti korisna za procenu kvaliteta komponenti, kao i za predviđanje koje će komponente verovatno biti skupe za izgradnju ili za održavanje.

5.12 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Dizajniranje se sastoji od mnogih timskih aktivnosti. Pošto se rešenja grade od komponenti, međusobna dejstva komponenti i podataka moraju biti dobro dokumentovana. U proces projektovanja spadaju česte diskusije sa ostalim članovima tima, ne samo da bi se koordinisalo međudejstvo različitih komponenti, već i da bi se bolje shvatili zahtevi, a za svaku odluku koju donesete, procenile njene posledice u kodu.

Takođe morate da radite i sa korisnicima dok projektujete interfejs sistema. Možete da napravite nekoliko prototipova kojima ćete korisnicima prikazati mogućnosti i utvrditi koji od njih zadovoljava zahteve performansi, ili da biste sami procenili koji od njih ostavlja najbolji utisak.

Mada je vaš izbor za metod dizajna ličan, morate da uzmete u obzir ko će ga čitati i kome treba da bude razumljiv. U nekim slučajevima se po završetku dizajniranja vaši pojedinačni dizajni prevode na zajedničku notaciju da bi oni koji budu kasnije menjali dizajn bolje shvatili komponente kao celinu. U drugim slučajevima se koristi unakrsno referenciranje da bi se bolje videlo koji delovi dizajna utiču na koje komponente i podatke. U svakom slučaju, bitno je da dizajn dokumentujete jasno i potpuno, sa opisom opcija koje ste imali i odluka koje ste doneli.

Kao član tima takođe ćete učestvovati u revizijama dizajna, u procenjivanju dizajna u njegovim različitim fazama i davaćete predloge za njegovo poboljšanje. Imajte na umu da kritikujete dizajn, a ne dizajnera, i da se softver najbolje razvija kada se iz tehničkih diskusija isključi lični aspekt.

5.13 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Dizajn softvera je kritični deo razvoja i u njemu se mnoga područja još uvek istražuju. Neki istraživači rade na ispitivanju prirode dizajna: Koje su bitne aktivnosti? Kakvi su najbolji dizajneri? Koje su karakteristike dobrog i lošeg dizajna? Drugi istraživači se usredsređuju na notacije i tehnike, u nadi da će napraviti metode koje omogućavaju automatsko proveravanje dizajna i unakrsne reference, a ujedno su razumljivi i lako se menjaju.

Shaw i Garlan (1996) su postavili osnove za studiju arhitekture sistema. Pitanja koja oni pokreću dovešće do značajnog unapređenja višekratne upotrebljivosti dizajna, njihove razumljivosti i lakoće menjanja. Posebno moramo da naučimo kako da osposobimo komponentu za višekratno korišćenje u velikom broju drugih aplikacija, kao i koje karakteristike dizajna proizvode sistem sa što manje grešaka.

Dizajn hardvera je više napredovao od dizajna softvera, zahvaljujući brojnim automatizovanim tehnikama i alatkama kojima se proverava kvalitet dizajna hardvera. Istraživači će i dalje primenjivati hardverske tehnike na dizajn softvera: analiza stabla grešaka, analiza režima otkazivanja i drugi pristupi verovatno će se proširivati, pa će se razmatrati ceo sistem umesto da se razdvajaju hardver i softver.

5.14 SEMINARSKI RAD

Osim što je dizajn inženjerski poduhvat, on takođe ima elemente umetnosti i kreativnosti. Razni stručnjaci za dizajn imaju veoma različite pristupe tome kako smišljaju i dokumentuju dizajn, ali svi proizvode solidne i elegantne projekte. Dizajneri dizajnu kao poslu pristupaju sa stanovišta kontinuuma, u rasponu od onog što zovemo dizajnom usredsređenim na zadatak do dizajna usredsređenog na korisnika. **Dizajn usredsređen na zadatak** započinje tako što dizajneri razmišljaju o onome što sistem mora da postigne. S druge strane, **dizajn usredsređen na korisnika** počinje od načina na koji korisnik sarađuje sa sistemom da bi se ti zadaci obavili. Ta dva pristupa se međusobno ne isključuju i često su komplementarna. Međutim, često jedna filozofija preovlađuje nad drugom. U okviru seminarskog rada razvijte dva različita pristupa dizajnu *Aranžera pozajmica*: jedan usredsređen na zadatak, a drugi usredsređen na korisnika. Koji stil dizajna ste izabrali za svaki od njih? Uporedite i analizirajte rezultate. Koji dizajn se lakše menja? Koji se lakše testira?

5.15 KLJUČNE REFERENCE

Postoje mnoge dobre knjige o softverskom dizajnu. Prva koju bi trebalo da pročitate je Shaw i Garlan (1996) da biste dobili arhitektonski okvir za način proučavanja dizajna. Druge korisne knjige su: Ward i Mellor (1986), Hatley i Pirbhai (1987), Shumate i Keller (1992), McConnelly (1993) i Gomma (1995) od kojih se većina bavi posebnim potrebama uporednih sistema i sistema u realnom vremenu. Takođe možete da se poslužite specijalizovanim knjigama o dizajnu interfejsa (Hix i Hartson 1993; Shneiderman 1997) i o bazama podataka (Weiderhold 1988).

Nekoliko časopisa posvetilo je posebna izdanja dizajnu korisničkih interfejsa. Izdanje za april 1993. časopisa *Communications of the ACM* bavilo se grafičkim korisničkim interfejsima, a izdanje za april 1996. dizajnom usmerenim na učenje. Izdanja za juli 1990. i juli 1997. časopisa *IEEE Software* detaljno obrađuju dizajn korisničkih interfejsa.

John McDermid je na čelu istraživačke grupe na univerzitetu u Yorku koja na analizu problema softverskog dizajna primenjuje analizu rizika, stabla grešaka i otkazivanja.

Mnogi problemi u vezi sa dizajnom obrađeni su u okviru foruma Risks koji je dostupan na internetu, a rekapitulacija se štampa u svakom izdanju časopisa *ACM Software Engineering Notes*. Stavke u forumu opisuju rizike i otkazivanja vezana za softver, a učesnici u raspravi često navode i razlog (koji je često vezan za dizajn).

5.16 VEŽBE

1. Koja vrsta arhitektonskog stila je predstavljena na modelu NIST/ECMA na slici 5.4?
2. U dizajnu Ariane-5, projektanti su doneli svesnu odluku da izostave obradu izuzetaka u tri slučaja od mogućih sedam slučajeva. Koje su zakonske i etičke posledice te odluke? Ko je zakonski i moralno odgovoran za katastrofu do koje je došlo? Da li je tim koji je radio na testiranju kriv što nije otkrio tu grešku u dizajnu?
3. Pregledajte ponovo arhitektonske stilove koje predlažu Shaw i Garlan (1996). Za svaki od tih stilova procenite da li komponente na višem nivou imaju visoku ili nisku koheziju, i jaku ili slabu spregu?
4. Za svaku vrstu kohezije napišite opis jedne komponente u kojoj se vidi ta vrsta kohezije.
5. Za svaku vrstu sprege navedite primer dve komponente koje su spregnute na taj način.
6. Nacrtajte dijagram sistema za sistem sa više nivoa međusobno spregnutih komponenti za neki projekat koji ste već razvili na nekim drugim predavanjima. Koliko je vaš sistem bio modularan? Koja vrsta sprege je postojala u njemu? Da li su komponente bile kohezivne? Da li se vaš sistem može preurediti tako da se poveća kohezija, a smanji spregnutost komponenti?
7. Može li sistem ikad da bude sasvim „nespregnut"? To jest, može li se stepen spregnutosti toliko smanjiti da među komponentama nema nikakve sprege?
8. Postoje li neki sistemi koji ne mogu da budu sasvim funkcionalno kohezivni? Zašto?
9. Za svaki atribut kvaliteta u modelima kvaliteta iz poglavlja 1, objasnite na koji način karakteristike dobrog dizajna doprinose kvalitetu proizvoda. Na primer, na koji način sprege, kohezija i modularnost utiču na pouzdanost i mogućnost praćenja?
10. U Bulovoj algebri je zbir promenljive i njenog komplementa jednak jedan. Koristeći ovo svojstvo proverite da li su izrazi za promenljive u tabeli 5.2 ekvivalentni jednostavnijim izrazima koji su prikazani.
11. Projektujte jednostavan editor na punom ekranu za terminalski monitor. Editor treba da dozvoli da se tekst dodaje, briše i menja. Delovi teksta mogu da se „iseku" iz jednog dela datoteke i „zalepe" na drugi deo datoteke. Korisnik može da navede tekstualni string, a editor treba da bude u stanju da pronađe sledeće pojavljivanje tog stringa. Putem editora korisnik može da navede širinu margine, dužinu stranice i položaje tabulatora. Nakon toga, procenite kvalitet dizajna.
12. Projektujte jednostavan interpreter. Vaš sistem će prihvatati niz znakova i utvrđivati da li on predstavlja važeću komandu u tom jeziku. Ako nije važeća, izdaće poruku o grešci, a ako jeste, izvršiće komandu. Kako vaš dizajn obrađuje greške? Opišite prednosti i mane različitih strategija za otkrivanje grešaka, za njihovo sprečavanje i za otpornost na greške u vašem dizajnu.

13. Rekurzivna je ona komponenta koja sama sebe poziva, ili na neki način referencira samu sebe. Prema smernicama koje smo izložili u ovom poglavlju, da li je dobro ili loše koristiti rekurzivne komponente? Zašto?
14. Navedite primer sistema kod kojeg razvijanje prototipa ne bi dovelo do značajne uštede vremena potrebnog za razvoj.
15. Navedite karakteristike sistema za koji najbolje odgovara prototip.
16. Objasnite zbog čega su modularnost i generatori aplikacija nerazdvojni pojmovi. Navedite primer generatora aplikacija sa kojim ste radili.
17. Objasnite zašto rešenje dizajna iz knjige Shaw i Garlan (1996) prikazano na slici 5.22 nije podesno za ponovnu upotrebu.
18. Navedite karakteristike koje mogu da se upotrebe u matrici za procenu dizajna. Za svaki od sledećih sistema odredite težine pojedinih karakteristika: operativni sistem, sistem za obradu teksta i sistem za praćenje satelita.
19. U mnogim projektima na predavanjima se traži da sami razvijete svoje programe. Napravite grupu studenata i održite reviziju dizajna za neki od tih projekata. Neka nekoliko studenata preuzme ulogu kupca i korisnika. Za preliminarnu reviziju dizajna izrazite sve zahteve i karakteristike sistema jezikom koji nije tehnički. Uporedite vreme potrebno za promene u fazi dizajna sa vremenom potrebnim za promenu postojećih programa.
20. Jedna firma za računarski konsalting vas je angažovala da razvijete paket koji će računati porez na prihod za jednu knjigovodstvenu firmu. Projektovali ste sistem prema zahtevima kupca i predstavili dizajn na reviziji dizajna. Koja od sledećih pitanja mogu da se jave na preliminarnoj reviziji dizajna? A na kritičkoj reviziji dizajna? Koja se pitanja mogu pojaviti na obe revizije? Objasnite svoje odgovore.

    (a) Na kom računaru će se to izvršavati?

    (b) Kako će izgledati ekrani za unos podataka?

    (c) Kakvi će se izveštaji proizvoditi?

    (d) Koliko će moći da bude istovremenih korisnika?

    (e) Da li ćete koristiti višekorisnički operativni sistem?

    (f) Opišite detalje algoritma za amortizaciju.

6 Razmatranje objekata



U ovom poglavlju obradivaćemo:
- posebnu prirodu objektno orijentisanog razvoja;
- slučajeve korišćenja;
- upotrebu UML-a;
- objektno orijentisani dizajn sistema;
- objektno orijentisani dizajn programa;
- objektno orijentisano merenje.

Sada kada umemo da prikupimo zahteve i dizajniramo sistem, napravićemo digresiju i pažljivije ispitati načela sadržana u objektno orijentisanom (OO) razvoju. Zašto bi objektno orijentisani razvoj zaslužívao posebnu pažnju? Zato što veliki broj novih sistema koji se danas razvijaju delimično ili potpuno prihvata objektnu orijentaciju. U ovom poglavlju detaljno opisujemo po čemu se objektno orijentisani razvoj razlikuje od ostalih tipova razvoja. Predstavićemo pojmove vezane za OO proces i pokazaćemo kako se oni koriste u prikupljanju zahteva, dizajnu sistema i dizajnu programa. Kao ilustraciju ovih pojmova koristićemo UML (*Unified Modeling Language* - objedinjeni jezik za modelovanje) u cilju projektovanja sistema servisne stanice Royal na osnovu zahteva navedenih u podsetniku 6.1.

Mnoge primere iz ovog poglavlja prvobitno je razvio profesor Guilherme Travassos iz COPPF/Sistemas na brazilskom saveznom univerzitetu u Rio de Janeiru. Više detalja o ovim i ostalim primerima može se naći na profesorovoj veb adresi http://www.cos.ufrj.br/~ght.

6.1 ŠTA JE OO?

Objektna orijentacija predstavlja pristup razvoju softvera u kojem se problem i njegovo rešenje organizuju kao skup diskretnih objekata pri čemu ta predstava obuhvata i strukturu podataka i ponašanje. OO repreznetacija se može prepoznati na osnovu svojih sedam karakteristika: identiteta, apstrakcije, klasifikacije, enkapsulacije, nasleđivanja, polimorfizma i perzistencije (trajanja). U nekim reprezentacijama koristi se samo podskup ovih karakteristika i mada se nazivaju objektno orijentisanim reprezentacijama, za njih često kažemo da su zasnovane na objektima.

PODSETNIK 6.1 ZAHTEVI ZA SISTEM SERVISNE STANICE ROYAL

1. Servisna stanica Royal pruža korisnicima tri vrste usluga: prodaju goriva, održavanje vozila i parkiranje. Klijent može da sipa gorivo u rezervoar svog vozila (automobila, motocikla ili kamiona); da popravi vozilo ili ga parkira na parkiralištu servisne stanice. Klijent može da plaća automatski neposredno po završetku korišćenja usluge (kupovine goriva, održavanja ili parkiranja) ili na osnovu računa koji mu se dostavlja mesečno. U oba slučaja, klijenti mogu da plate gotovinski, kreditnom karticom ili čekom građana. Gorivo se u servisnoj stanici Royal prodaje prema ceni po galonu u skladu sa vrstom goriva: dizel, normal ili premijum. Servisiranje se naplaćuje zavisno od cene delova i radne snage. Parkiranje se naplaćuje po dnevnoj, nedeljnoj ili mesečnoj tarifi. Cene goriva, servisiranja i parkiranja mogu da se menjaju, ali ih može unositi i menjati jedino Mani, rukovodilac servisne stanice. Mani takođe ima diskreciono pravo da odobri popust određenom kupcu, pri čemu taj popust nije isti za sve kupce. Na sva plaćanja primenjuje se lokalni porez na promet od 5%.
2. Sistem treba da prati račune na mesečnoj, a proizvode i usluge na dnevnoj osnovi. Rezultate tog praćenja rukovodilac servisne stanice može da zahteva u formi izveštaja.
3. Rukovodilac servisne stanice koristi sistem za kontrolu dobara. Sistem treba da signalizira ako količina na zalihama padne ispod zadate granice i automatski formira porudžbenice za delove ili gorivo.
4. Sistem treba da prati zaduženja i šalje opomene kupcima koji kasne sa plaćanjem. Računi se kupcima šalju prvog dana u mesecu nakon kupovine. Plaćanje dospeva prvog dana narednog meseca. Ako se račun ne plati u roku od 90 dana od dana izdavanja računa, kupcu se ukida kredit.
5. Sistem se primenjuje samo na stalne kupce. Stalnim kupcem se smatra kupac identifikovan imenom, adresom i datumom rođenja, koji je koristio usluge stanice bar jednom mesečno u poslednjih šest meseci.
6. Sistem mora da rukuje podacima koji se koriste za spregu sa drugim sistemima. Sistem kreditnih kartica koristi se za obradu transakcija sa kreditnim karticama kod plaćanja proizvoda i usluga. Sistem kreditnih kartica koristi broj kartice, ime, datum važenja i iznos kupovine. Posto prihvati ove informacije, sistem kreditnih kartica vraća odgovor da je transakcija odobrena ili odbijena. Sistem za naručivanje delova prihvata šifru dela i broj komada. On vraću datum kada će se delovi isporučiti. Sistem za naručivanje goriva zahteva da opis porudžbine sadrži vrstu goriva, količinu u galonima, naziv stanice i identifikacionu šifru stanice. On vraća datum kada će se gorivo isporučiti.
7. Sistem mora da evidentira porez i odgovarajuće informacije, uključujući iznos poreza koji je platio svaki kupac, kao i iznos poreza po stavkama.
8. Rukovodilac stanice na zahtev dobija pregled evidencije zapisa o porezu.
9. Sistem treba da povremeno šalje kupcima poruke kojima ih podseća da je potrebno servisiranje njihovog vozila. U normalnim okolnostima servis je potreban svakih šest meseci.

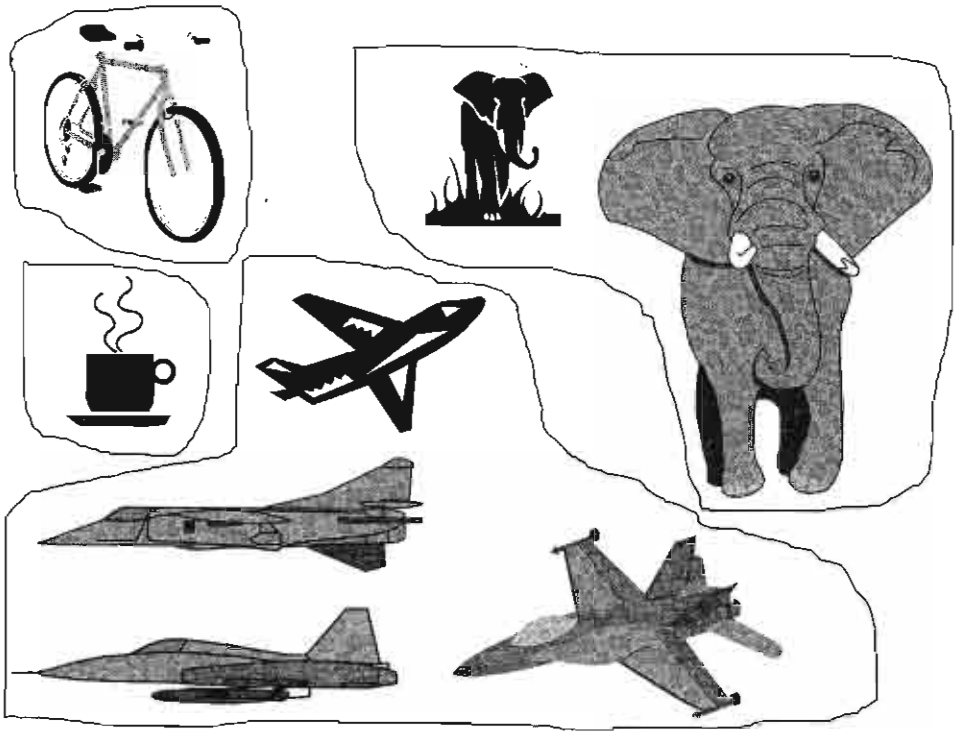
10. Kupci mogu da zakupe mesto za parkiranje na dnevnoj osnovi. Svaki kupac može od sistema da zahteva uvid u raspoloživi prostor za parkiranje. Rukovodilac stanice može da pregleda mesečni izveštaj u kojem se vidi koliko je mesta bilo raspoloživo, a koliko zauzeto.
11. Sistem održava evidenciju o računima, kojoj se može pristupati na osnovu broja računa i imena kupca.
12. Rukovodilac stanice treba da na zahtev pregleda informacije o izdatim računima.
13. Sistem može, na zahtev rukovodioca stanice, da generiše izveštaj koji sadrži analizu cena i datih popusta.
14. Sistem treba da automatski upozori vlasnike neaktivnih računa. To jest, šalje se poruka kupcima koji u periodu od dva meseca nisu ništa kupili u servisnoj stanici.
15. Sistem ne sme biti nefunkcionalan duže od 24 sata.
16. Sistem mora da štiti informacije o kupcima od neovlašćenog pristupa.

Identitet postoji zbog činjenice da su podaci organizovani u diskretne, jedinstveno prepoznatljive celine, koje zovemo objekti. Objekat ima svojstvena stanja i ponašanja. Na primer, u sistemu koji kontroliše branu na reci, sama brana može da predstavlja objekat. Brana može da ima nekoliko stanja kao što su *potpuno_otvorena* (kada rečna voda neometano teče kroz branu) i *potpuno_zatvorena* (kada kroz branu ne prolazi nimalo vode). Ponašanja vezana za branu mogu da budu *zvučno_upozorenje*: kada brana treba da pređe iz zatvorenog stanja u otvoreno stanje, ona može da emituje zvučno upozorenje turistima da će nivo vode da poraste. Svaki objekat u OO sistemu obično poseduje **ime** koje se ponekad naziva **referenca** (ili „hendl") a obezbeđuje međusobno razlikovanje objekata.

U poglavljima 1, 4 i 5 videli smo da je *apstrakcija* bitna za izgradnju svakog sistema, bez obzira na to da li je on OO ili ne. Apstrakcije u OO sistemu pomažu da se predstave različiti aspekti sadržani u sistemu koji je predmet projektovanja. Združene apstrakcije formiraju hijerarhiju koja prikazuje međusobne odnose različitih pogleda na sistem.

OO koristi **klasifikacije** za grupisanje objekata sa zajedničkim svojstvima i ponašanjem. Na primer, na slici 6.1 prikazan je skup različitih objekata. U jednoj grupi imamo dva objekta slon, a u drugoj četiri aviona. U zasebnoj klasi imamo jedan bicikl, a u drugoj šolju.

Grupa slonova može se smatrati klasom. Klasi slonova možemo dodeliti atribute kao što su *boja*, *veličina* i *poreklo* (tj. afrički slon ili azijski slon). Štaviše, ovoj klasi možemo dodeliti operacije ili ponašanja: slon se *kreće*, *hrani* i *pere*. Slično tome, klasa *avion* poseduje atribute kao što su *boja*, *veličina*, *brzina* i *broj_putnika* i operacije kao što su *poletanje*, *sletanje* ili *remont*. Klasu možemo da predstavimo okvirom kao na slici 6.2. U ovom primeru svaki objekat klase slon poseduje atribute kao što su visina, težina, boja, poreklo i broj kljova. Ponašanje uključuje praćenje hranjenja i promenu mesta boravka.



SLIKA 6.1 Primeri objekata grupisanih u klase



SLIKA 6.2 Okvir koji predstavlja ime objekta, atribute i ponašanja

Imajte u vidu da grupisanje na slici 6.1 odražava shvatanja osobe ili tima koji grupiše objekte. Na primer, na slici imamo zasebne klase za bicikle i avione. Međutim, jednako smislena klasifikacija bi mogla bicikl da pridruži avionima i formira klasu prevoznih sredstava. Važno je imati na umu da definicije i hijerarhije klasa imaju za cilj da predstave problem ili njegovo rešenje; dva potpuno različita predstavljanja mogu biti podjednako ispravna i korisna.

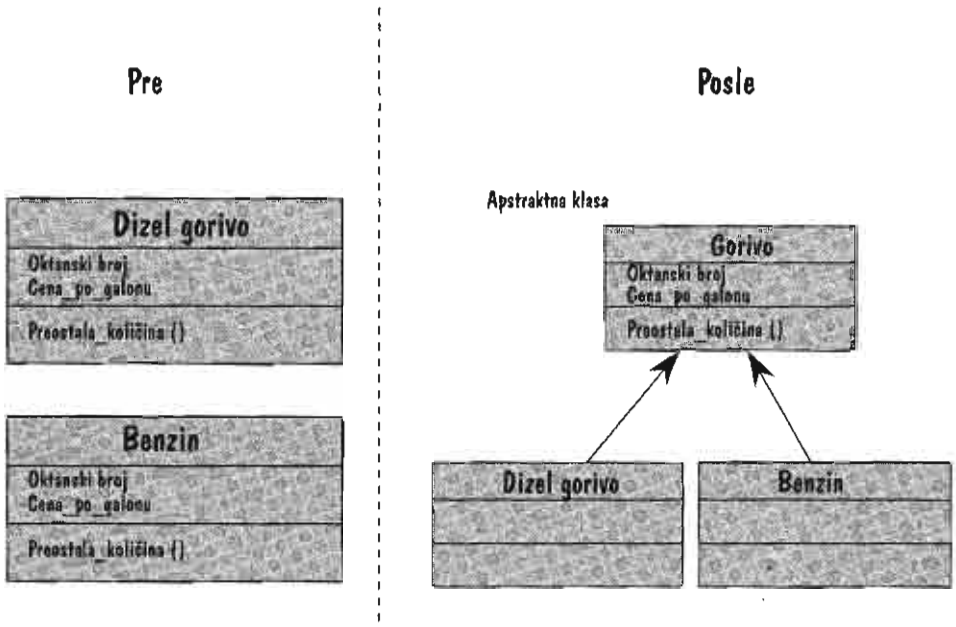
Za svaki objekat kažemo da je **primerak (instanca)** neke klase. Svaki primerak poseduje vlastite vrednosti atributa (tj. način da se u svakom trenutku opiše njegovo stanje) ali sa ostalim primercima iste klase deli imena atributa i ponašanja. O slonovima možemo da razgovaramo kao o klasi i razmatramo njihova zajednička svojstva kao što su kljove ili uši. Ali takođe možemo iskoristiti vrednost atributa za razlikovanje pojedinačnih pripadnika klase; na primer, azijski slonovi imaju neke karakteristike koje su različite od karakteristika afričkih slonova. Prema tome, klasa opisuje skup objekata sa zajedničkom strukturom i ponašanjem pri čemu nam vrednosti atributa omogućavaju da konkretne objekte međusobno razlikujemo.

OO sistem se oslanja na *enkapsuliranje*, pojam uveden u poglavlju 4. Klasa enkapsulira ponašanja i atribute objekta i sakriva implementacione detalje. Međutim, Berard (2000) ističe da enkapsuliranje nije isto što i sakrivanje informacija. Zaštitna granica koju enkapsuliranje stvara može da bude providna ili poluprovidna. To jest, nekada su stvari iza zaštitne granice vidljive (ona je providna i vidljiva) dok su nekada iza nje skrivene (ona je poluprovidna). Berard napominje da je „apstrahovanje tehnika koja nam pomaže da uočimo koja konkretna informacija treba da bude vidljiva, a koja skrivena. Enkapsuliranje je otuda tehnika za pakovanje informacija na takav način da se sakrije ono što treba da bude skriveno, a da se pri tome vidi ono što želimo da bude vidljivo".

Jasno je da su neki atributi zajednički za sve članove date klase. Na primer, i afrički i azijski slonovi imaju isti broj kljova, ali se razlikuju po drugim karakteristikama kao što je oblik ušiju. Klase možemo organizovati u hijerarhiju na osnovu sličnosti ili razlika, pri čemu ta hijerarhija definiše strukturu **nasleđivanja**. Da bismo izgradili hijerarhiju, počinjemo od sveobuhvatne definicije klase, a zatim je rafiniramo u specijalizovane potklase. Potklasa može da nasleđuje kako strukturu tako i ponašanje i atribute nadređene klase. Na primer, na slici 6.3 prikazano je kako se dve vrste goriva mogu uključiti u hijerarhiju klasa goriva. Ponekad u cilju pojednostavljenja hijerarhije koristimo **apstraktnu klasu,** koja sama ne instancira objekte već se oni instanciraju jedino kao objekti potklasa.

Ponašanje je akcija ili transformacija koju vrši objekt ili koja se nad njim vrši. Ponašanje objekta se pokreće prijemom neke poruke ili ulaskom u neko stanje. Ponekad se isto ponašanje različito manifestuje kod različitih klasa ili potklasa i ta osobina je poznata pod nazivom **polimorfizam**. Na primer, razmotrimo klasu poligona. Svaki poligon ima površinu, ali se površina trougla drugačije izračunava od površine pravougaonika. Prema tome, izračunavanje površine zavisi od objekta na koji se primenjuje.

Konkretna implementacija operacije za određenu klasu naziva se **metoda**. U sistemu sa polimorfizmom, više različitih metoda implementira istu operaciju. Na primer, može da postoji jedan metod za izračunavanje površine trougla, a drugi za izračunavanje površine kvadrata. Objektno orijentisani programski jezik projektovan je da automatski bira korektan metod za implementiranje operacije, parametre i naziv klase objekta. U našem primeru sa površinom, objekat će izabrati pravilnu metodu za računanje površine na osnovu parametara koji opisuju poligon. Polimorfizam omogućava dodavanje nove klase bez promene postojećeg koda. Tako na primer, u hijerarhiju sa slike 6.3 lako može da se doda novo gorivo kao što je prirodni gas ili ugalj. (Međutim, možda ćemo morati bolje da razmislimo o metodima. Da li se pojam oktana može primeniti na ugalj ili prirodni gas? A šta je sa cenom po galonu?)



SLIKA 6.3 Formiranje hijerarhije

Sedmo svojstvo OO sistema je persistentnost (**trajnost**): sposobnost da ime, stanje i ponašanje objekta opstanu u vremenu i prostoru. Drugim rečima, ime, stanje i ponašanje objekta se čuvaju prilikom njegove transformacije. Na primer, možda želimo da naš sistem čuva informaciju o ceni za svaki galon dizel goriva. Cena može da se menja svaki dan, tako da kupac koji danas plaća 1,50 dolara po galonu može sutra da plaća 1,55 dolara. Ali, mi hoćemo da dnevna cena bude trajno sačuvana da bismo mogli da poredimo prošlogodišnju cenu sa ovogodišnjom ili da nacrtamo dijagram promene cene u vremenu. U tom slučaju objekat je trajan, iako se njegovi atributi menjaju.

6.2 OO PROCES RAZVOJA

Jedna prednost objektno orijentisanog razvoja je konsistentnost jezika. Problem i njegovo rešenje možemo da opišemo istim pojmovima: klasama, objektima, metodima, atributima i ponašanjima. Tokom celog procesa razvoja trebalo bi da vodimo računa o konsistentnosti terminologije. Na primer, bez obzira na to u kom delu procesa predstavljamo neki objekt, moramo da navedemo njegovo ime, atribute i ponašanja. Na sledećem, višem nivou, klase se opisuju sa tri OO aspekta predstavljanja: statičkog, dinamičkog i restriktivnog. **Statički prikaz** sadrži opise objekata, atributa, ponašanja i relacija. **Dinamički prikaz** opisuje komunikaciju, kontrolu/vremenski redosled, stanja i prelaze stanja. **Restrikcije** opisuju ograničenja vezana za strukturu (na primer, vrednosti ili kardinalnost atributa) i dinamičko ponašanje.

Ta konsistentnost u celom procesu predstavlja ključnu razliku između tradicionalnijih procesa razvoja i onog koji je objektno orijentisan. OO proces oblikuje nezavisne jedinice (objekte) pomoću enkapsuliranja podataka i ponašanja. Ista semantika se koristi za predstavljanje sistema, od formulisanja zahteva, pa sve do implementacije i testiranja aplikacije. Prema tome, OO je filozofija za predstavljanje problema i rešenja, a ne životnog ciklusa softvera. Zaista, OO može da se upotrebi u mnogim modelima životnog ciklusa softvera, od modela vodopada do spiralnog modela. A kao što ćemo videti u poglavlju 12, pošto se u OO-u koristi enkapsuliranje i sakrivanje informacija, mnogi objektno orijentisani projektanti često posmatraju objekte i klase iz aspekta mogućnosti njihovog ponovnog korišćenja. U opštem slučaju, OO se bavi zahtevima, dizajnom višeg nivoa, dizajnom nižeg nivoa, kodiranjem i testiranjem, ali ne nužno na sekvencijalan način. Redosled diktira model životnog ciklusa a ne OO predstava.

U tabeli 6.1 navedene su različite karakteristike softverskog proizvoda ili projekta. Desna kolona sugeriše malu verovatnoću promena nekih karakteristika, dok druge imaju veliku verovatnoću promene u toku procesa razvoja. Stavke iz ove tabele mogu pomoći pri izboru procesa razvoja koji je najpogodniji za razmatranu situaciju.

OO zahtevi

Bez obzira na model životnog ciklusa, proces objektno orijentisanog razvoja zahteva korake formulisanja zahteva, dizajniranja sistema, dizajniranja programa, kodiranja i testiranja. OO analiza zahteva obično se iskazuje jezikom korisnika i opisuje pojmove i moguća scenarija iz domena primene. Pojmovi obuhvataju informacije, usluge i odgovornosti. Domenska znanja omogućavaju projektantima da shvate kontekst u kojem će se sistem koristiti i opišu zahteve na, za korisnika, razumljiv način.

Na primer, drugi zahtev servisne stanice Royal konstatuje da:

> Sistem mora da prati račune na mesečnom nivou, a proizvode i usluge na dnevnoj osnovi. Rezultati tog praćenja mogu da se prosleđuju rukovodiocu servisne stanice na njegov zahtev, u formi izveštaja.

Tabela 6.1 Težnja za promenama kada se koristi OO paradigma (Jacobson i dr. 1995)

Karakteristike softverskog proizvoda/projekta	Verovatnoća promene
Objekti izvedeni iz aplikacije	Mala
Dugovečne informacione strukture	Mala
Pasivni atributi objekta	Srednja
Sekvence ponašanja	Srednja
Sprega sa spoljnim svetom	Velika
Funkcionalnost	Velika

Značenje zahteva ostaje isto, nezavisno od načina na koji projektanti odluče da implementiraju sistem. To znači da definicija zahteva može biti nezavisna od njenog predstavljanja u formi objekata.

OO dizajn

Kao što smo videli u poglavlju 4, specifikacija zahteva (tj. dokument namenjen dizajnerima) obično se iskazuje objektno, sa ciljem da projektantima omogući izvođenje dizajna na osnovu opisa problema. Međutim, u mnogim slučajevima je OO opis problema isti ili sličan prvim koracima u pronalaženju OO rešenja. Na taj način, u toku OO razvoja koraci u izradi specifikacije zahteva mogu da u stvarnosti predstavljaju prve korake dizajniranja sistema. U oba slučaja potrebno je da se prepoznaju objekti i opišu njihove međusobne veze. Zbog toga se u OO razvoju često izbegava razdvajanje postupka specifikovanja zahteva od izrade dokumenta o zahtevima.

Postoje dva uputstva koja se mogu primeniti na objektno orijentisano predstavljanje dizajna sistema. Prvo, važno je prepoznati i predstaviti klase i objekte. Moramo da prepoznamo ne samo objekte realnog sveta (domen problema), već takođe i detalje vezane za svojstva i ponašanja svakog objekta. Drugo, moramo da prepoznamo međusobna dejstva i odnose među objektima i klasama: njihove asocijacije, kompozicije, agregacije i veze nasleđivanja.

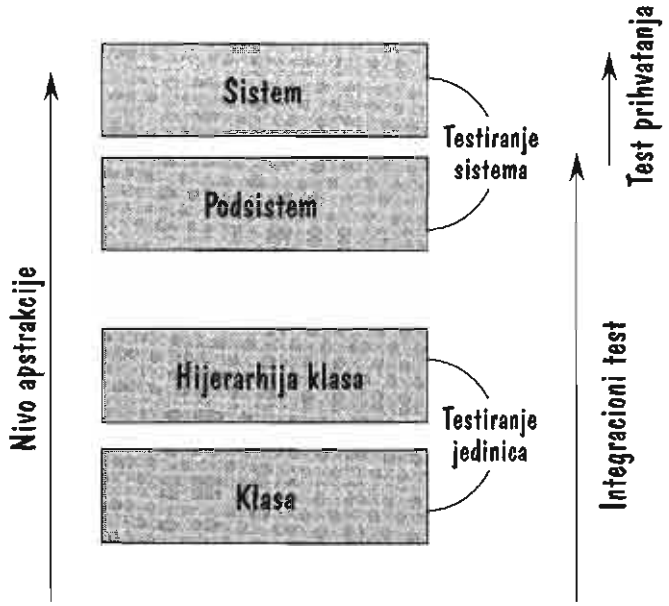
Dizajn sistema predstavlja apstrakciju visokog nivoa onoga što će na kraju predstavljati dizajn programa. Polazeći od dizajna sistema dizajneri programa obavljaju niz koraka u cilju obezbeđenja detalja koji su bitni za implementaciju:

- ubacuju izračunavanja u modele;
- ubacuju neke detalje biblioteke klasa, obično oslanjajući se na pristup odozdo nagore;
- razmatraju nefunkcionalne zahteve, kao što su performanse i bezbednost, i u tom smislu unapređuju dizajn.

OO kodiranje i testiranje

Kada se završi dizajn programa, sistem je opisan na veoma niskom nivou pomoću modela objekata, atributa i ponašanja. Kodiranjem se modeli prevode na OO programski jezik. To se ne radi mehanički. Obično je potrebno da implementatori prečiste hijerarhijske strukture i izvrše podešavanja u skladu sa sazrevanjem i evolucijom zahteva. U nekim situacijama se prilikom implementacije teži uopštavanju sistema kako bi objekti i klase bili pogodniji za kasniju ponovnu upotrebu.

Testiranje OO sistema obuhvata neke aktivnosti koje se javljaju u testiranju drugih vrsta sistema. Koderi vrše testiranje jedinica koje su sami razvili, a zatim sarađuju sa timom za testiranje pri integracionim testovima, testiranju sistema i sprovođenju završnog testa prihvatanja. Na slici 6.4 prikazano je kako nivo apstrakcije utiče na različite vrste testiranja. Različite vrste testova će biti detaljnije objašnjene u poglavljima 8 i 9. Međutim, neke karakteristike objektne orijentacije zahtevaju posebnu pažnju prilikom testiranja; njih opisujemo u poglavlju 8.



SLIKA 6.4 Odnosi vrsta testiranja prema OO strukturi (prirađeno iz Jacobson i dr. 1995)

6.3 SLUČAJEVI KORIŠĆENJA

Kada se sistem razvija pristupom orijentisanim na objekt, korisno je zahteve iskazati i predstaviti tehnikom slučajeva korišćenja. Kao što je opisano u poglavlju 4, **slučaj korišćenja** opisuje konkretnu funkcionalnost koja se očekuje od sistema, tako što modelira dijalog koji će korisnik, spoljni sistem ili drugi entitet, voditi sa sistemom koji razvijamo. Na primer, slučaj korišćenja može da opisuje interakciju klijenta banke sa bankomatom dok podiže novac sa svog računa. Ili može detaljno da opisuje interakciju ATM-a sa bazom podataka evidencije klijenta da bi se odredilo koliko novca on može da podigne. Entitet koji je u interakciji sa sistemom naziva se **učesnik**, a to može da bude korisnik, uređaj ili neki drugi sistem.

Svaki slučaj korišćenja opisuje mogući scenario interakcije spoljnog entiteta sa sistemom. Često se slučaj korišćenja predstavlja kao dijagram koji sadrži relevantne objekte uz dodatak kratkog tekstualnog opisa (**skript scenarija**) načina rada funkcije. Za svaki scenario, slučaj korišćenja identifikuje sve događaje koji mogu da nastupe kao i reakcije sistema na njih. Sveukupnost slučajeva korišćenja formira kompletan opis načina na koji entiteti mogu da koriste razmatrani sistem. Prema tome, skup slučajeva korišćenja odslikava sveukupnu funkcionalnost sistema.

Slučajevi korišćenja su posebno pogodni za komunikaciju sa kupcima, projektantima sistema i timovima za testiranje. Kupci mogu da pročitaju slučajeve korišćenja i provere da li se sistem projektuje tako da obuhvati sve željene funkcije. Dizajneri sistema mogu da upotrebe slučajeve korišćenja da bi izdvojili objekte kojima treba manipulisati i elemente podataka koje na odgovarajućim mestima treba skladištiti. Tim za testiranje može da upotrebi slučajeve korišćenja kao osnov za testiranje sistema. Kada je sistem dovršen, kupac takođe može, na osnovu slučajeva korišćenja, da formuliše plan testa prihvatanja

tako da se svaka nit funkcionalnosti uključi bar u jedan test. U stvari, čak se i sistemska dokumentacija može bazirati na slučajevima korišćenja. Prema tome, slučajevi korišćenja odgovaraju svim fazama životnog ciklusa i služe kao sredstvo za komunikaciju među ulogama. Iz tog razloga su slučajevi korišćenja prihvaćeni u većini OO zajednica kao dopuna formalnim metodama za modelovanje objekata u toku analize sistema.

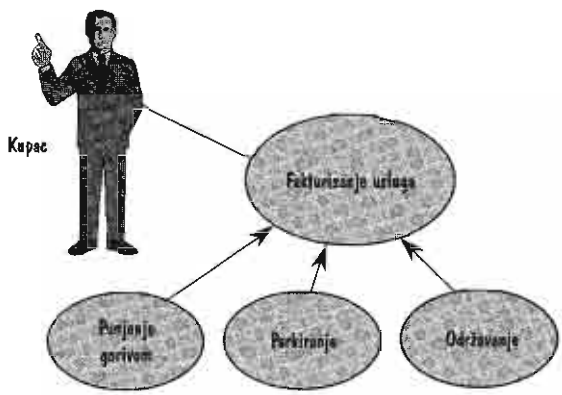
Dijagrami slučajeva korišćenja sadrže četiri elemenata: *učesnike, slučajeve, proširenja* i *korišćenja*. Kao što je gore napomenuto, učesnik je uloga koju entitet igra u odnosu na sistem. **Slučaj** je opis nekog aspekta funkcionalnosti sistema koji je vidljiv onom učesniku iz čije perspektive se slučaj korišćenja odigrava. **Proširenje** proširuje slučaj korišćenja da bi se ilustrovala drugačija ili dublja perspektiva. **Korišćenje** je u suštini ponovna upotreba ranije definisanog slučaja korišćenja.

Na primer, razmotrimo sistem za implementiranje zahteva servisne stanice Royal. Prikaz visokog nivoa mogao bi da izgleda kao na slici 6.5. Ovde servisna stanica izvršava tri vrste usluga: punjenje gorivom, parkiranje i održavanje. Kupac je učesnik u ovoj sceni, a za svaku vrstu usluge dobija fakturu.

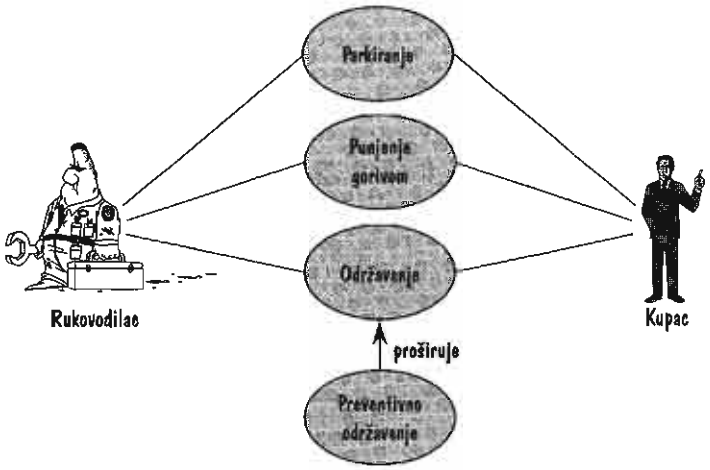
Slučajeve korišćenja upotrebljavamo da bismo lakše razumeli kupce i njihove probleme. Za svaki dijagram slučajeva korišćenja pišemo scenario u kojem opisujemo funkcionalnost koju sistem treba da zadovolji. Kako postavljamo pitanja, pronalazimo aspekte problema koji nisu bili vidljivi ili dovoljno jasni u ranijim opisima slučajeva korišćenja. Na primer, možemo da proširimo deo dijagrama slučajeva korišćenja sa slike 6.5, promenimo ga i dodelimo elemente prikazane na slici 6.6. U novom opisu imamo dodatnog učesnika, rukovodioca stanice i novu uslugu: preventivno održavanje. Ovde preventivno održavanje proširuje pojam održavanja za jednu konkretnu vrstu usluge.

Zatim uzimamo u obzir činjenicu da će mnogi kupci plaćati račune kreditnim karticama. Sistem za kreditne kartice mora da ima interfejs prema našem sistemu i zato ga uključujemo kao drugo proširenje prikazano na slici 6.7.

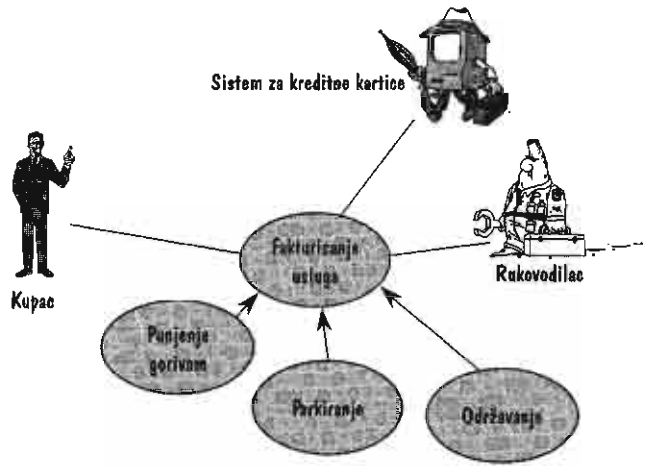
Na kraju, proširujemo sistem tako da obuhvati sve učesnike. Da bismo to uradili, tražimo sve entitete u okruženju (tj. izvan sistema koji se gradi) koji dolaze u dodir sa sistemom da bi izvršili zadatak ili pružili uslugu. Entiteti mogu biti bilo koje vrste: ljudi koji koriste sistem, drugi sistemi, druge organizacije ili spoljni uređaji. Mi posmatramo šta ti učesnici rade i gde se u obradi sistema taj zadatak izvršava. Na primer, neki učesnici pokreću događaje. Interakcija drugih učesnika sa sistemom je rezultat događaja. U svakom slučaju, učesnika posmatramo kao ulogu, a ne kao konkretnu osobu. John i James su različiti kupci, ali je njihova interakcija sa sistemom ista.



SLIKA 6.5 Prikaz servisne stanice Royal



SLIKA 6.6 Prvo proširivanje dijagrama servisne stanice Royal, kojim se uključuje preventivno održavanje.



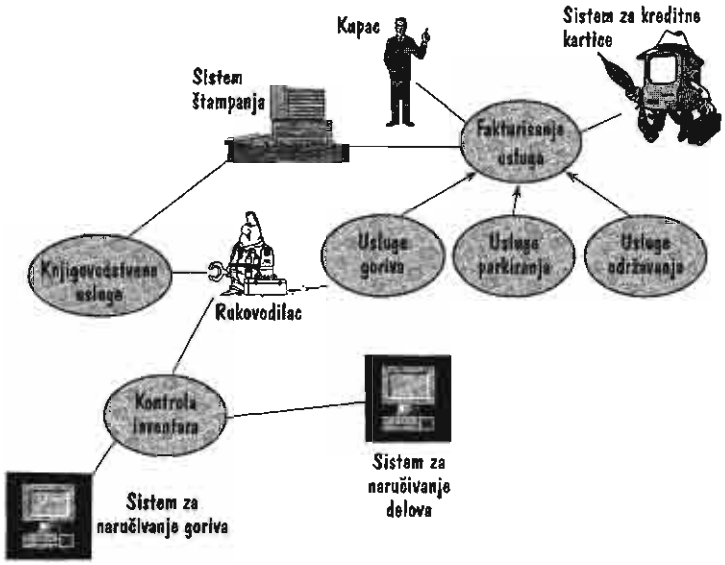
SLIKA 6.7 Drugo proširivanje dijagrama servisne stanice Royal, sa dodavanjem sistema za kreditne kartice.

U cilju lakše identifikacije učesnika možemo da koristimo sledeći niz pitanja:

- Koji korisnici ili grupe upotrebljavaju sistem da bi izvršili neki zadatak?
- Koji su korisnici ili grupe potrebni da bi sistem mogao da obavi svoje funkcije?
- Koji spoljni sistemi koriste naš sistem da bi obavili neki zadatak?
- Koji spoljni sistemi, korisnici ili grupe šalju informacije sistemu?
- Koji spoljni sistemi, korisnici ili grupe primaju informacije iz sistema?

Na osnovu ovih pitanja ćemo možda doneti odluku da u dijagram sistema servisne stanice Royal dodamo sistem za štampanje računa, sistem za naručivanje dodatnog goriva kada su zalihe nedovoljne i sistem za naručivanje delova potrebnih za održavanje i popravke. Možda ćemo takođe odlučiti da u sistem dodamo još dva podsistema: knjigovodstveni

koji će se baviti porezom i sistem za kontrolu inventara koji će se baviti gorivom i delovima koje servisna stanica drži na zalihama. Rezultat bi mogao da liči na ono što je prikazano na slici 6.8. Slika nije potpun slučaj korišćenja. Kao što ćemo videti u sledećem odeljku, za svaku stavku dijagrama moramo da napravimo opis: šta ta stavka radi, kada to radi i koji podaci i ponašanja se tu uključuju. Rezultat je sveobuhvatan opis načina na koji će sistem raditi, uključujući i međudejstva sa sistemima izvan njegovih granica.



SLIKA 6.8 Treće proširivanje dijagrama servisne stanice Royal, kojim se uključuje inventarisanje i knjigovodstvo.

Ovaj proces analize sistema korak po korak sa različitih aspekata daje nam nešto više od proširenog skupa slučajeva korišćenja. Dajući nam različite predstave funkcionalnosti sistema, slučajevi korišćenja nam pružaju alternativne načine razmišljanja o zahtevima sistema. Osim što proširuju i razjašnjavaju zahteve, slučajevi korišćenja mogu da pomognu u otkrivanju grešaka u zahtevima. Ako je opis zahteva na prirodnom jeziku urađen u skladu sa pravilima, transformacija zahteva u slučajeve korišćenja je relativno jednostavna. Međutim, u zahtevima iskazanim prirodnim jezikom problemi su ponekad prikriveni; ne vide se jasno dok čitamo formulacije zahteva. Oni tek izlaze na videlo u fazi prevođenja zahteva u slučajeve korišćenja. Na taj način je prevođenje zahteva na slučajeve korišćenja samo po sebi dobra provera kvaliteta zahteva.

Kada napravimo sve slučajeve korišćenja, možemo da ih ispitamo i tražimo probleme koji već postoje ili one koji se mogu pojaviti. Na primer, možemo da postavimo ovakva pitanja:

- Da li se u slučajevima korišćenja upotrebljava pravilna terminologija? To jest, da li se za isti entitet ili učesnika u zahtevima koriste dva različita izraza ili više njih?
- Da li su aktivnosti usklađene sa očekivanjima? To jest, da li slučajevi korišćenja opisuju međudejstvo sistema i učesnika na način koji je konsistentan sa opisom tog učesnika?

- Da li su opisi aktivnosti i učesnika jasni i potpuni? Da li se opis može interpretirati na više načina?
- Postoji li opis interakcije za koju nedostaje odgovarajući opis učesnika?
- Postoji li spoljni sistem ili skup učesnika koji je opisan, a da u stvarnosti nije uključen u aktivnost ili interakciju sa sistemom?
- Da li je jasno kada se svaka aktivnost započinje i završava?

Ova pitanja naglašavaju značaj granica sistema koje smo opisali u poglavlju 1. Ispitivanjem slučaja korišćenja, dobijamo mogućnost da bolje shvatimo šta se nalazi unutar, a šta izvan sistema, i šta je u interakciji sa sistemom, a šta je od njega sasvim odvojeno.

U isto vreme možemo dodatno da preispitamo i samu funkcionalnost. Nije uvek lako opisati ponašanje sistema na način prihvatljiv za sprovođenje opisa u praksu. Razmislite kako biste opisali zahteve na osnovu kojih bi neko naučio da vozi bicikl. Na prvi pogled izgleda dovoljno da opišemo način okretanja pedala nogama, pokrete ruku kojima se vrši upravljanje, kočenje i eventualno menjanje brzina. Tek kasnije nam postaje jasno da postoje suptilni pokreti celog tela koje je teško opisati, a neophodni su za održavanje ravnoteže i za skretanje. Na isti način moramo da u slučajeve korišćenja uključimo sve one male, ali značajne stvari, bitne za pravilno funkcionisanje sistema. Videćemo da neke stvari mogu da se precizno opišu, dok su neke subjektivne i neodređene. Kako povećavamo stepen razumevanja sistema, posredstvom izrade prototipova ili vizija različitih opcija projekta, vrlo je verovatno da ćemo se ponovo vratiti slučajevima korišćenja i preciznije ih definisati.

Jedan od načina za povećanje razumevanja je dokumentovanje veza među slučajevima korišćenja. Neophodno je evidentirati koji slučajevi korišćenja zavise jedni od drugih. Odakle potiče svaki entitet? Gde on nastaje? Gde se njegov „život" završava? Posmatramo učesnike uključene u pojedinačne slučajeve korišćenja i proveravamo da li svaki slučaj korišćenja ima sve potrebne učesnike. Slično tome, ispitujemo zavisnosti između slučajeva korišćenja: Koji slučajevi moraju da se završe da bi razmatrani slučaj mogao da počne? Koji slučajevi mogu da počnu nakon što se konkretan slučaj završio? Postoje li neki konflikti ili nedoslednosti? Tako formirana matrica praćenja može da bude korisna i za procenjivanje uticaja promene u slučaju da kupac, dizajner ili tim za testiranje predlože reviziju zahteva ili dizajna. Kao što ćemo videti u poglavljima 9 i 11, mogućnost praćenja i kontrola konfiguracije bitni su ne samo za izgradnju sistema, već i za njegovo održavanje.

6.4 PREDSTAVLJANJE OO: PRIMER UPOTREBE UML-A

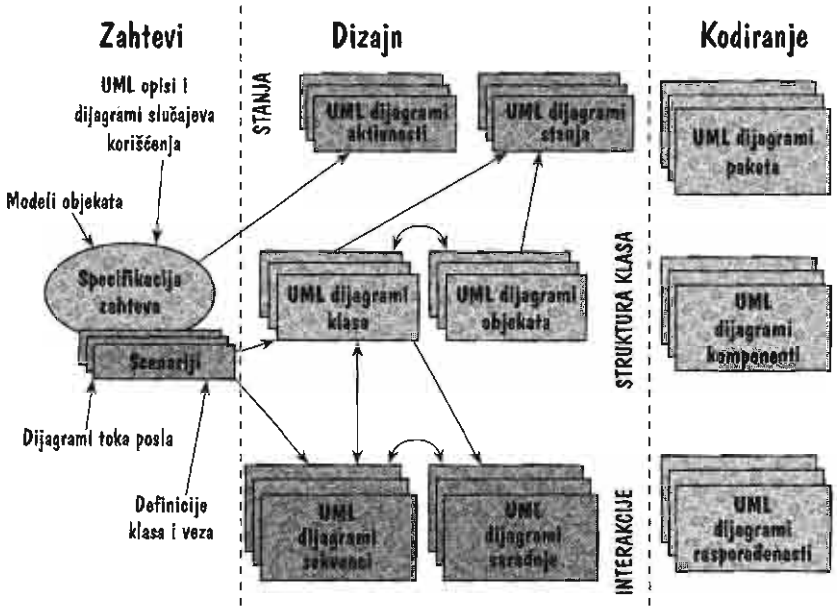
UML (*Unified Modeling Language*) je popularna notacija za opisivanje OO rešenja. Ona se može podesiti u skladu sa različitim situacijama u razvoju i modelima životnog ciklusa softvera. U stvari, neke organizacije kao što je na primer, Object Management Group prihvatile su UML kao standardnu OO notaciju. U ovom odeljku dajemo pregled elemenata UML-a koji su uvedeni u poglavlju 4 i koristimo UML u cilju formulisanja dizajna na visokom nivou apstrakcije za problem servisne stanice Royal. Opširniji uvod u UML možete naći u knjizi Larman (1998).

UML može da se koristi za vizaulizaciju, specifikovanje ili za dokumentovanje problema. Posebno je koristan za opisivanje različitih varijanti projektnih rešenja a posebno za dokumentovanje proizvoda procesa projektovanja. UML dijagrami obuhvataju: dina-

mički prikaz sistema, statički prikaz, ograničenja i formalizaciju. Dinamički prikaz se izvodi putem slučaja korišćenja, spiska aktivnosti, dijagrama interakcije koji modeluju saradnju objekata i sekvence u obavljanju aktivnosti, i dijagramima stanja koji ilustruju stanja sistema i prelaze stanja. Statički prikaz se definiše na osnovu dijagrama klasa sa njihovim vezama (asocijacija, generalizacija, zavisnost i implementacija) i proširenjima (ograničenja, ključne vrednosti i stereotipi). Osim toga, statički prikaz obuhvata pakete i raspodelu. Ograničenja i formalni zahtevi se izražavaju oslanjanjem na OCL (*Object Constraint Language*). U nastavku odeljka razradićemo problem servisne stanice Royal pomoću UML-a i nekih njegovih elemenata.

UML u procesu

Pošto su OO koncepti primenljivi u svim fazama razvoja, UML može da se koristi tokom celokupnog procesa razvoja softvera. Na slici 6.9 prikazan je način na koji se UML može koristiti u fazama specifikacije zahteva, dizajna i kodiranja. Treba primetiti da UML koristi većinu notacija formulisanih u poglavlju 4. Svaka od njih naglašava neki aspekt sistema, tako da njihovom kombinacijom dobijamo detaljnu sliku problema ili predloženog rešenja. U procesu formulisanja zahteva **dijagrami toka posla** definišu ceo poslovni proces posredstvom pojedinačnih aktivnosti koje se izvršavaju u kontekstu posla. Kao što smo videli, dijagrami slučajeva korišćenja mogu da opišu sistem koji se projektuje na osnovu opšteg opisa postupaka koje sistem mora da sprovede. Osim toga, slučajevi korišćenja mogu da obuhvate različite scenarije u kojima se opisuju način rada sistema i interakcija korisnika sa sistemom. Ovi dijagrami se mogu dopuniti **modelom objekata,** koji definiše klase objekata imajući u vidu njihove funkcije. Klase i njihove veze se u startu, dok traje faza razrade pojedinačnih korisničkih scenarija, neformalno definišu da bi se zatim, kako dizajn napreduje, sve strožije formalizovale.



SLIKA 6.9 Kako UML podržava razvojni proces

Kada su zahtevi dobro definisani, dizajniranje počinje UML dijagramima stanja i dijagramima aktivnosti. Dijagrami aktivnosti prikazuju sve aktivnosti do kojih može doći u sistemu kako se menjaju vrednosti objekta. Na primer, objekat može da proizvede kratak izveštaj za ulaz tipa A, a duži za ulaz tipa B; objekat-izveštaj proizvodi odgovarajući rezultat na osnovu ulaza. U kombinaciji sa dijagramima aktivnosti razvijamo dijagrame stanja da bismo prikazali sva moguća stanja objekta. Prelazak iz jednog stanja u drugo pokreće se porukom (koja predstavlja događaj) koja se šalje od jednog objekta ka drugom. Prema tome, dijagrami stanja su neophodni ako neka klasa može da ima više promena stanja.

Zatim se dizajn nastavlja razmatranjem statičke strukture objekata. Skup dijagrama strukture i dijagrama objekata opisuje saradnju klasa, uključujući i odnose nasleđivanja. Saradnja klasa ilustruje se dijagramima interakcije. Postoje dve vrste tih dijagrama: dijagrami sekvenci i dijagrami saradnje. **Dijagrami sekvenci,** koje smo uveli u poglavlju 4, prikazuju kako se poruke kreću od jednog objekta ka drugom formalnim opisivanjem događaja i zahteva. **Dijagrami saradnje** koriste informacije o objektima i sekvencama u cilju formulisanja toka događaja između objekata.

Na kraju, dizajn može da se implementira oslanjajući se na pakete, komponente i dijagrame raspoređenosti. **Dijagrami paketa** prikazuju kako se klase logički dele na module. **Dijagrami komponenti** su često identični dijagramima paketa, osim što obuhvataju stvarne finalne module sistema. Pošto je krajnji sistem često deo neke veće strukture, **dijagram raspoređivanja** pokazuje čvorove mreže koji su uključeni u razvijanu aplikaciju.

6.5 OO DIZAJN SISTEMA

Da biste videli kako se UML može koristiti, uzećemo za primer dizajn višeg nivoa za sistem servisne stanice Royal. Napravićemo dizajn sistema na osnovu zahteva formulisanih u podsetniku 6.1. Opis problema servisne stanice Royal postoji, a naš zadatak je da formulišemo rešenje. Za početak koristimo UML dijagrame klasa. Ti dijagrami opisuju tipove objekata i njihove statičke relacije. Konkretno, želimo da prikažemo povezanost objekata (na primer, kupac je povezan sa fakturom) i relacije tip-podtip (na primer, dizel gorivo je podtip tipa gorivo). Želimo da dijagrami ilustruju atribute svakog objekta, pojedinačna ponašanja i ograničenja za svaku klasu ili objekat.

Proces projektovanja počinje iskazivanjem zahteva. Zatim pokušavamo da izdvojimo imenice i na osnovu tih konkretnih elemenata formulišemo polazni skup klasa. Tražimo:

- strukture,
- spoljne sisteme,
- uređaje,
- uloge,
- radne postupke,
- mesta,
- organizacije,
- stvari sa kojima projektovani sistem manipuliše.

Na primer, razmotrite ovaj deo prvog zahteva servisne stanice Royal.

> Klijent može da plaća automatski u toku kupovine (goriva, održavanja ili parkiranja) ili na osnovu računa koji mu se dostavlja mesečno. U oba slučaja, klijenti mogu da plate gotovinski, kreditnom karticom ili čekom građana. Gorivo se u servisnoj stanici Royal prodaje prema ceni po galonu u skladu sa vrstom goriva: dizel, normal ili premijum. Servisiranje se naplaćuje zavisno od cene delova i radne snage. Parkiranje se naplaćuje po dnevnoj, nedeljnoj ili mesečnoj tarifi. Cene goriva, servisiranja i parkiranja mogu da se menjaju i jedino Mani, rukovodilac servisne stanice, može da unosi ili menja cene. Mani takođe ima diskreciono pravo da odobri popust određenom kupcu pri čemu taj popust nije isti za sve kupce. Na sva plaćanja primenjuje se lokalni porez na promet od 5%.

Na osnovu ovako iskazanog zahteva moguće je izdvojiti nekoliko potencijalnih klasa:

- lični ček,
- štampanu fakturu,
- kreditnu karticu,
- kupca,
- rukovodioca stanice,
- kupljene stvari,
- gorivo,
- usluge,
- popuste,
- porez,
- parkiralište,
- servisera,
- gotovinu,
- cene.

Sledeća pitanja mogu da posluže kao uputstvo za pronalaženju kandidata za klase:

1. Šta je to što treba da se na neki način „obradi"?
2. Koje stavke poseduju više atributa?
3. Kada postoji više objekata u jednoj klasi?
4. Šta potiče od samih zahteva, a nije izvedeno iz našeg tumačenja zahteva?
5. Koji atributi i operacije se mogu uvek primeniti na neku klasu ili objekat?

Ova pitanja mogu da nam pomognu u grupisanju kandidata za klase i objekte, kao što je prikazano u tabeli 6.2.

Zatim, ispitujemo ostale zahteve da bismo videli kako oni proširuju postojeću tabelu atributa i klasa. Na primer, peti zahtev glasi:

Tabela 6.2 Prvo grupisanje atributa i klasa: korak 1

Atributi	Klase
lični ček	kupac
faks	serviser
cena	usluge
gotovina	parking
kreditna kartica	gorivo
popusti	faktura
	kupljene stvari
	rukovodilac stanice

Sistem se primenjuje samo na stalne kupce. Stalnim kupcem se smatra kupac identifikovan imenom, adresom i datumom rođenja, koji je koristio usluge stanice bar jednom mesečno u poslednjih šest meseci.

Slično tome, deveti zahtev glasi:

Sistem treba da povremeno šalje kupcima poruke kojima ih podseća da je potrebno servisiranje njihovog vozila. U normalnim okolnostima servis je potreban svakih šest meseci.

Prema tome, imamo dodatne kandidate za klase, kao što su stalni kupac, ime, adresa i povremene poruke. Zato ćemo od tabele 6.2 napraviti tabelu 6.3. Međutim, „stalni kupac" već postoji i zato ga ne dodajemo.

Za kompletan skup zahteva moramo proširiti našu tabelu i stići do tabele 6.4.

Zatim razmatramo ponašanja koja moraju da se opišu u našem dizajnu. Iz specifikacije zahteva izdvajamo glagole. Ponovo tražimo konkretne elemente koji opisuju ponašanja:

- zapovedne glagole,
- pasivne glagole,
- akcije;
- stvari ili podsetnike,

Tabela 6.3 Prvo grupisanje atributa i klasa: korak 2

Atributi	Klase
lični ček	kupac
faks	serviser
cena	usluge
gotovina	parkiralište
kreditna kartica	gorivo
popusti	faktura
datum rođenja	kupljene stvari
ime	povremene poruke
adresa	rukovodilac stanice

Tabela 6.4 Prvo grupisanje atributa i klasa: korak 3

Atributi	Klase
lični ček	kupac
faks	serviser
cena	usluge
gotovina	parkiralište
kreditna kartica	gorivo
popusti	faktura
datum rođenja	kupljene stvari
ime	povremene poruke
adresa	rukovodilac stanice
	pismo upozorenja
	delovi
	računi
	inventar
	sistem kreditnih kartica
	sistem za naručivanje delova
	sistem za naručivanje goriva

- uloge,
- radne postupke,
- usluge koje pruža neka organizacija.

Ponašanja će postati aktivnosti ili obaveze koje preuzima klasa ili objekt ili aktivnosti koje se vrše nad klasom ili objektom. Na primer, fakturisanje kupcu je ponašanje. To ponašanje izvršava deo sistema servisne stanice Royal (tj. sistem servisne stanice Royal je obavezan da izvrši fakturisanje), a faktura utiče na kupca.

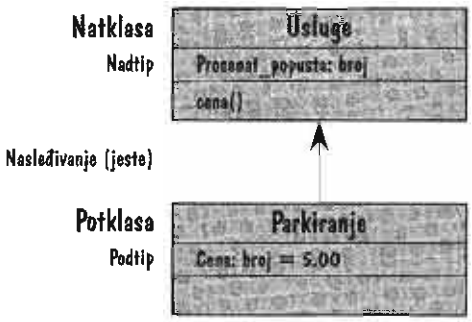
U cilju lakšeg rukovanja objektima, klasama i ponašanjima koristićemo UML dijagrame za opis njihovih odnosa. Na slici 6.10 prikazan je UML ram koji se koristi za ilustrovanje delova klase. Gornja trećina rama sadrži ime klase, srednja trećina informacije o atributima, a donja opis operacija. Svaki atribut se opisuje imenom, tipom i početnom vrednošću. Slično se svaka operacija opisuje imenom, listom argumenata i tipom povratne vrednosti.



SLIKA 6.10 Ram klase koji predstavlja fakturu

РЕПУБЛИКА СРПСКА
Природно-математички
факултет Бања Лука
БИБЛИОТЕКА

UML ramovi se postavljaju na dijagram tako da se uoče veze između klasa na koje se odnose. Kada se jedan ram nalazi iznad drugog, a povezani su strelicom kao što je prikazano na slici 6.11, tada gornji predstavlja natklasu donjeg. Kao što smo ranije videli, ta veza nasleđivanja ponekad se naziva vezom *jeste(je)*, a omogućava da donji ram iz gornjeg nasledi atribute i ponašanja.



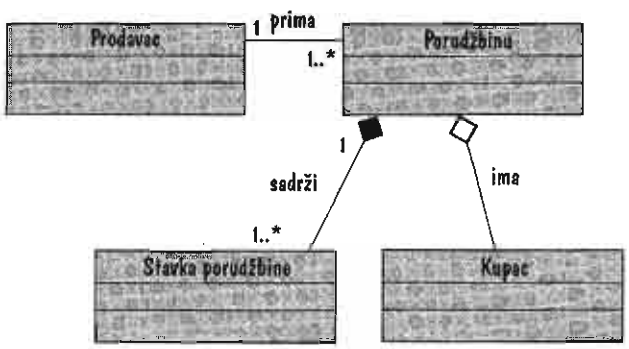
SLIKA 6.11 Odnosi nasleđivanja

Relacije obično pripadaju jednom od četiri tipa, a to su: generalizacija, asocijacija, agregacija i kompozicija. Kada imamo relaciju nasleđivanja, natklasa generalizuje potklasu. Na primer, klasa *gorivo* predstavlja generalizaciju klase *dizel gorivo*.

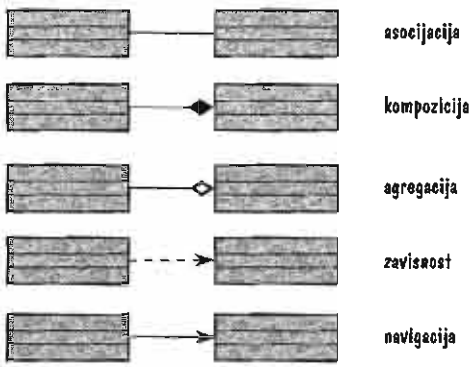
Dve klase **su u asocijaciji** ako se javljaju zajedno i ako navedena veza mora da se održi tokom određenog vremena. Na primer, na slici 6.12 se vidi da je svaka porudžbina pridružena nekom prodavcu. Asocijacija je prikazana ravnom linijom, a brojevi na krajevima te linije opisuju kardinalitet svakog člana relacije. U ovom slučaju jedan prodavac može da primi jednu porudžbinu ili više njih.

Kada je jedna klasa deo druge klase, imamo primer **agregacije**. Na primer, na slici 6.12 stavka porudžbine je deo porudžbine. Relaciju *je_deo* označavamo linijom koja se završava popunjenim rombom; romb označava veći entitet koji sadrži povezanu klasu kao njegov deo. I ovde možemo da na krajevima linije definišemo kardinalitet. Na slici smo pokazali da jedna porudžbina može da sadrži više naručenih stavki.

SLIKA 6.12 Asocijacija klasa



Prazan romb ukazuje na agregaciju koja nije relacija nasleđivanja. U našem primeru kupac poseduje porudžbinu, ali klasa *kupac* nije potklasa klase *porudžbina*. Na slici 6.13 vidimo nekoliko drugih načina za označavanje relacija među klasama.

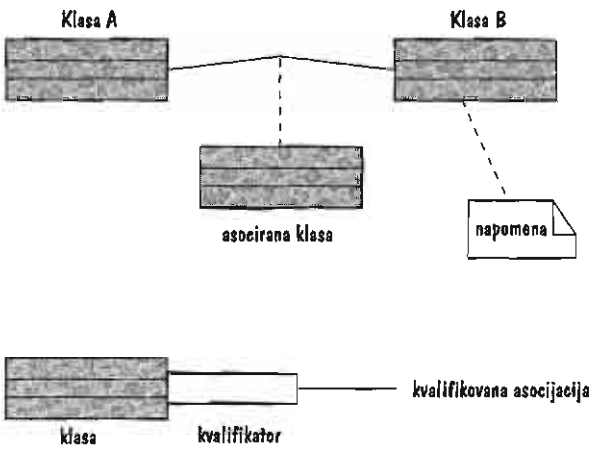


SLIKA 6.13 Tipovi relacija među klasama

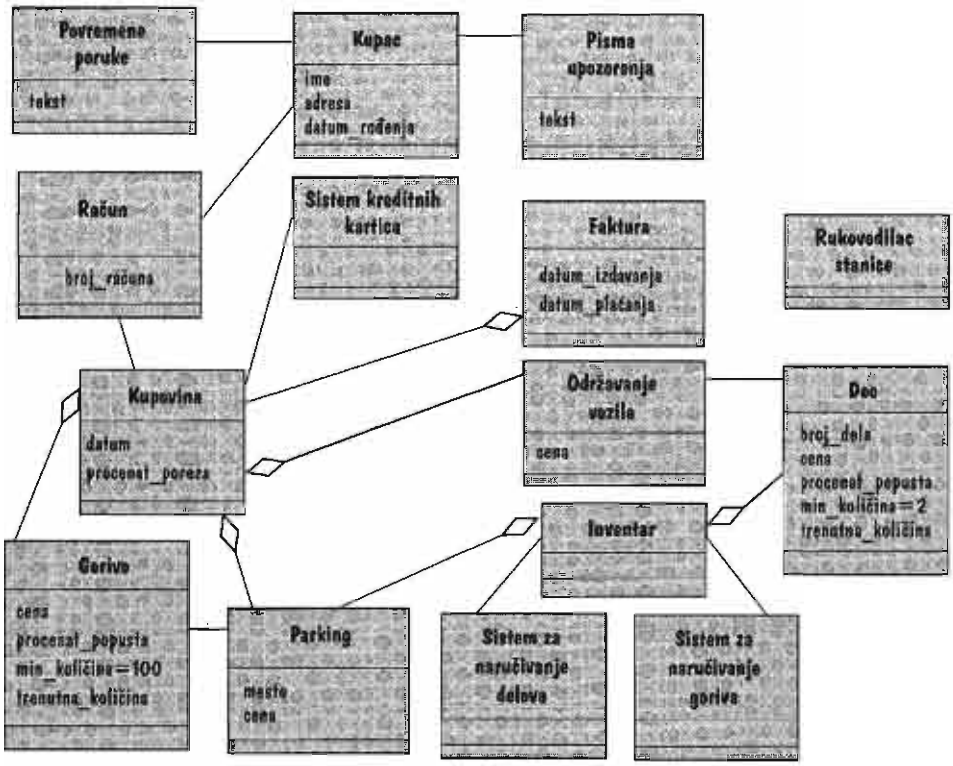
UML poseduje i druge načine za predstavljanje asocijacije klasa, uključujući napomene o pojedinačnim klasama. Na slici 6.14 vidimo kako se asocijacija klasa može poboljšati pomoću napomena i kvalifikatora. Opširniji opis relacija i UML notacije pročitajte u knjizi Larman (1998).

Primenom ove notacije možemo nacrtati ramove klasa početnog modela za servisnu stanicu Royal. Petnaest ramova sa slike 6.15 prikazuje petnaest klasa za koje smatramo da će biti potrebne u našem objektno orijentisanom prikazu rešenja. Upotrebili smo UML notaciju da bismo prikazali kako smo razumeli relacije među klasama.

Razmotrimo ovaj prikaz problema servisne stanice Royal i rešenje predstavljeno pomoću klasa i relacija. Ako uporedimo taj dizajn sa zahtevima iz podsetnika 6.1, vidimo da dijagram klasa može značajno da se unapredi. Prvo, možemo da dodamo klasu poruka, pa će pisma upozorenja i povremene poruke biti različite vrste poruka. Zatim, primećujemo da klasa goriva treba da se poveže sa klasom inventar, pošto moramo da pratimo koliko goriva imamo na zalihama da bismo ga naručili kada količina padne ispod one koja je dozvoljena kao minimalna.



SLIKA 6.14 Dodatna UML notacija za napomene i kvalifikacije



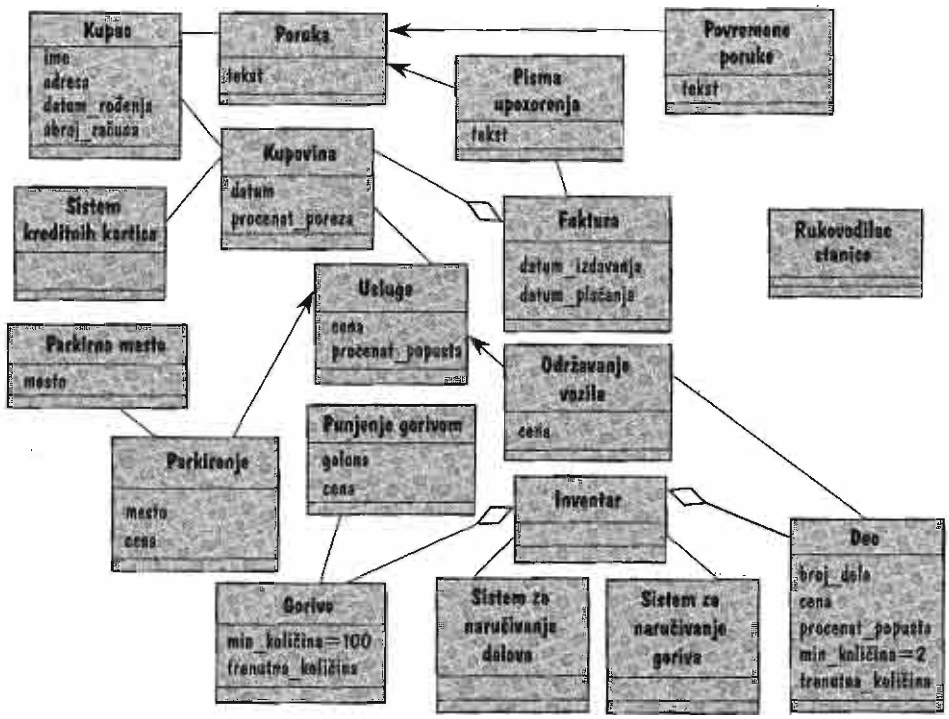
SLIKA 6.15 Polazni model za servisnu stanicu Royal

Dizajn možemo da uprostimo brisanjem klase *račun* (pošto ima samo jedan atribut) i dodavanjem broja računa u klasu *kupac*. Štaviše, možemo da dodamo tri nove klase, *punjenje gorivom*, *parkirno mesto* i *usluge*, tako da klasa *usluge* obuhvati cene i popuste za svaku vrstu usluge. Uvođenjem klase *usluge* koristimo polimorfizam; cene i popusti mogu da budu različiti zavisno od tipa usluge. Možemo takođe da uklonimo procenat popusta iz klase *gorivo* i dodamo ga u klasu *deo*.

Pošto smo uklonili cenu iz klase *gorivo* i dodali je u klasu *punjenje gorivom*, u klasi *gorivo* sada pratimo samo količinu, a broj litara i cena konkretne kupovine nalaze se u klasi *punjenje gorivom*. Na sličan način stavljamo cenu parkirnog mesta u klasu *parking*, a mesto u klasu *parkirno mesto*. Na taj način klasa *parkirno mesto* prati različita mesta na kojima se vozila mogu parkirati, što nam omogućava da napravimo neophodne izveštaje o trendovima korišćenja parkirališta. Rezultirajući model je prikazan na slici 6.16.

Ne moramo ovde da stanemo. Ako sada pogledamo drugi model, videćemo da klasa *rukovodilac stanice* nije u relaciji ni sa jednom drugom klasom; prema tome, brišemo je iz modela. Takođe vidimo da nam je potrebna dodatna klasa *servisna stanica* da bismo pratili neaktivne račune. Zatim, svakoj klasi dodajemo operacije da bismo prikazali funkcionalnost koja se zahteva u podsetniku 6.1. Na primer, klasi *faktura* potrebna je operacija kojom se izračunava ukupna cena i operacija kojom se izračunava porez za tu kupovinu. Slično tome, klasi *inventar* potrebne su operacije za komunikaciju sa sistemima za naručivanje goriva i delova (*naručivanje_goriva*, *naručivanje_delova*), a klasi *parkirno mesto*

potrebna je operacija kojom se utvrđuje da da li mesto *je_slobodno*. Na kraju, u model dodajemo kardinalitete u cilju boljeg razumevanja relacija između klasa. Rezultat je prikazan na slici 6.17.



SLIKA 6.16 Druga verzija modela za servisnu stanicu Royal

Ostali UML dijagrami

Potrebno nam je više različitih dijagrama u cilju upotpunjavanja modela sistema. Prvo, svaka klasa mora detaljnije da se opiše na osnovu *šablona za opis klasa*. Šablon ukazuje na položaj klase u hijerarhiji (s obzirom na dubinu nasleđivanja), kontrole izvoženja, kardinalitet (tj. koliko objekata treba očekivati u klasi) i asocijacije sa ostalim klasama. Tu se takođe navode operacije koje klasa sadrži i javni interfejs klase. Dajemo jedan primer šablona za opis klase za klasu *punjenje gorivom* u sklopu modela koji vam je predstavljen.

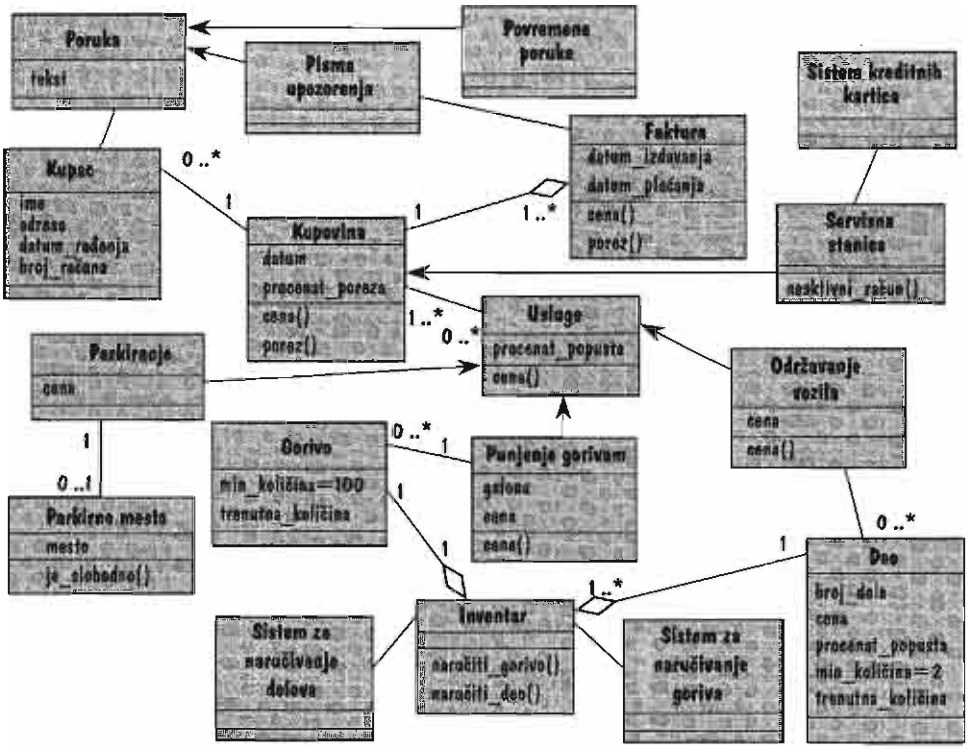
Ime klase: ***punjenje gorivom***

Kategorija: usluge

Spoljni dokumenti:

Kontrola izvoženja: javni

Kardinalnost: n



SLIKA 6.17 Treća i konačna verzija modela za servisnu stanicu Royal

Hijerarhija:

Natklase: usluge

Pridruživanja:

pridruženo gorivo<ime>

Ime operacije: cena

Javni član klase: punjenje gorivom

Dokumentacija:

//Izračunava konačnu cenu goriva

Preduslovi:

broj galona > 0

Dijagram objekata: (nije naveden)

Semantika:

konačna_cena = galoni * cena

Dijagram objekata: (nije naveden)

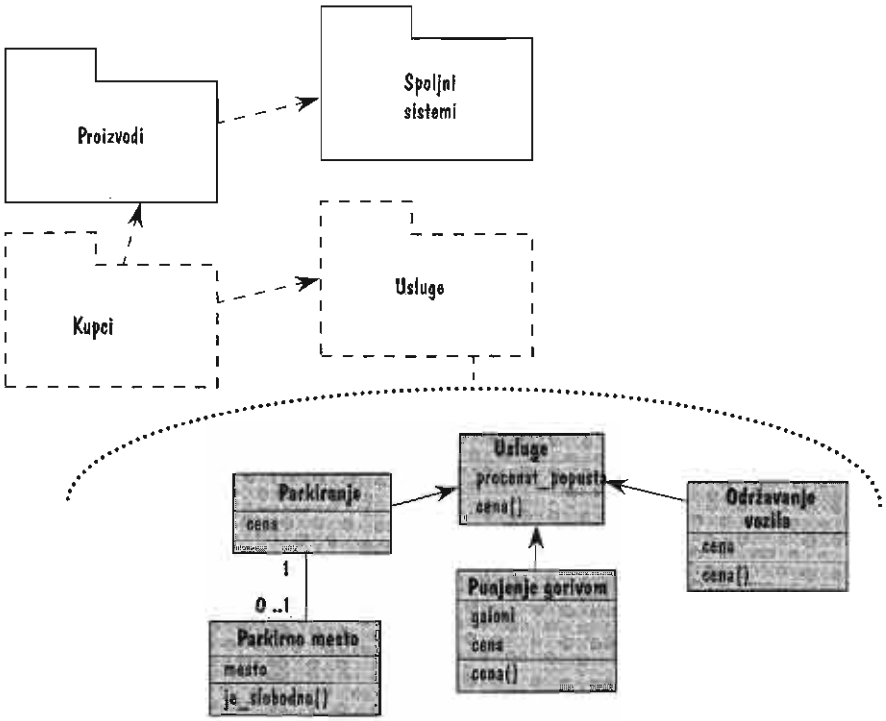
Konkurentnost: sekvencijalna

Javni interfejs:
Operacije:
cena
Privatni interfejs:
Atributi:
galoni
cena
Implementacija:
Atributi:
galoni
cena
Automat stanja: ne
Konkurentnost: sekvencijalna
Trajnost: privremena

Obratite pažnju da šablon za opis klase postavlja osnovu za dizajn programa. Drugim rečima, šablon za opis klase sadrži informacije koje su bitne programerima za implementaciju dizajna u kodu. Na primer, šablon sadrži obrasce koji će se koristiti u operacijama klase i razdvaja javni interfejs od privatnog. **Privatni interfejs** klase je mehanizam koji ograničava pristup članicama klase; drugi objekti ne mogu da vide atribute ili (neke) metode te klase. **Javni interfejs** omogućava pristup nekim metodima, ali ne i svim atributima, pošto bi javni atribut predstavljao prekršaj principa enkapsulacije.

UML obuhvata i **dijagrame paketa** tako da možemo da posmatramo sistem kao kolekciju paketa koji se zatim mogu proširiti u veći skup klasa. Dijagrami paketa prikazuju zavisnosti među klasama ukoliko one pripadaju različitim paketima. Kažemo da su dve stavke **zavisne** ako promena u definiciji jedne stavke može da dovede do promene u drugoj. Na primer, dve klase mogu da budu zavisne ako jedna klasa šalje drugoj klasi poruku, ako jedna klasa poseduje podatke potrebne drugoj klasi ili ako jedna klasa koristi drugu klasu kao parametar za izvršavanje neke operacije. Konkretno, dva paketa su **zavisna** ako postoji zavisnost između klasa u jednom i drugom paketu. Kao i u svakom dobrom dizajnu, uvek težimo da nepotrebne sprege svedemo na minimum; zato se trudimo da imamo što manje zavisnosti.

Paketi i njihove zavisnosti su posebno važni u fazi testiranja. Kao što ćemo videti u poglavljima 8 i 9, slučajeve testiranja moramo tako da projektujemo da obuhvatimo sve moguće interakcije među komponentama. UML dijagrami paketa pomažu da shvatimo zavisnosti i generišemo test slučajeve. Na slici 6.18 je prikazan primer jednog UML dijagrama paketa za servisnu stanicu Royal. Na slici postoje četiri glavna paketa: spoljni sistem, kupci, proizvodi i usluge. Svaki paket se sastoji od klasa iz dizajna prikazanog na slici 6.17. Na primer, paket usluga se sastoji od pet ključnih klasa: *usluge*, *punjenje gorivom*, *parkiranje*, *parkirno mesto* i *održavanje vozila*; prikazali smo ih na slici da bismo ilustrovali veze među klasama i paketima, ali se oni obično ne uključuju u dijagram paketa. Isprekidane strelice prikazuju zavisnosti među paketima. Na primer, paket usluga zavisi od paketa kupac. Kao što vidite, dijagram paketa daje pregled celokupnog sistema na najvišem nivou i ukazuje na zavisnosti na visokom nivou apstrakcije.

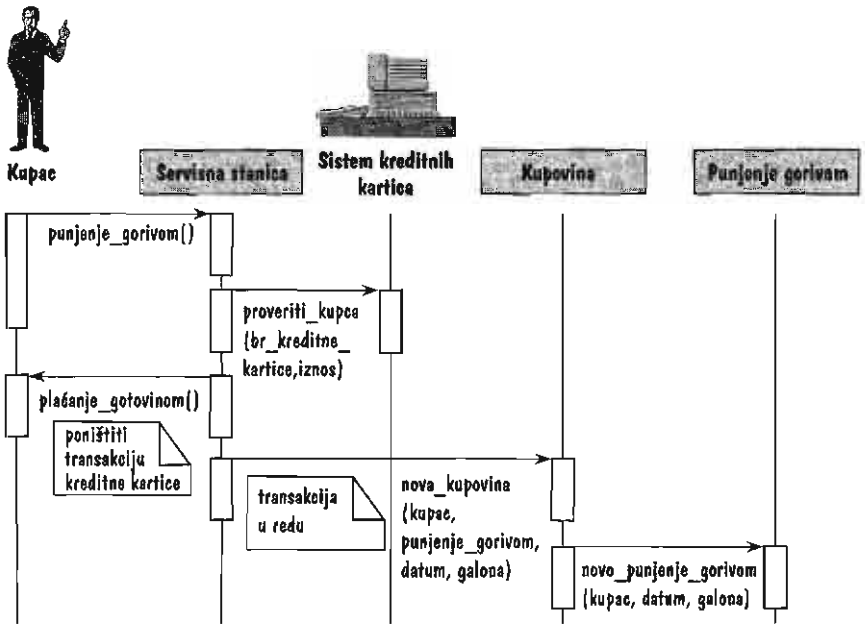


SLIKA 6.18 Dijagram paketa za servisnu stanicu Royal

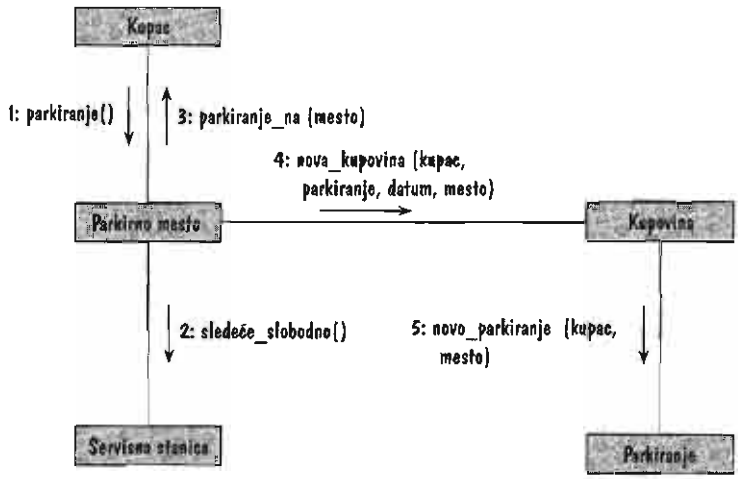
UML takođe sadrži **dijagrame interakcije**, koji opisuju kako objekti u modelu vrše operacije i primenjuju ponašanja. Obično pravimo po jedan dijagram interakcije za svaki slučaj korišćenja da bismo ukazali na poruke i operacije koje definišu sveukupnu funkcionalnost sistema.

Postoje dve vrste dijagrama interakcije: dijagrami sekvence i i dijagrami saradnje. **Dijagram sekvence** pokazuje redosled kojim se odvija neka aktivnost ili ponašanje. Objekat se prikazuje kao ram na vrhu vertikalne linije koja predstavlja njegovu **liniju života**. Uzani okvir na liniji života označava početak ili kraj poruke. Strelica između dve linije života predstavlja poruku između dva objekta i nosi oznaku sa imenom poruke, a ponekad i uslovom koji mora biti ispunjen da bi se poruka poslala. Zvezdica na strelici označava da se poruka šalje više puta različitim objektima primaocima. Kada se strelica poruke povezuje na ram odakle je krenula (na isti objekt), objekt šalje poruku samom sebi; ta vrsta poruke naziva se **samodelegiranje**. Na slici 6.19 dat je primer dijagrama sekvence za servisnu stanicu Royal koji prikazuje slučaj korišćenja za klasu *punjenje gorivom*.

Dijagram saradnje se takođe zasniva na slučaju korišćenja i pokazuje kako su objekti statički povezani. Kao i kod dijagrama sekvence, objekti su ikone, a strelice se koriste za prikazivanje poruka. Međutim, za razliku od dijagrama sekvence, redosled poruka označava se rednim brojevima. Na primer, dijagram saradnje sa slike 6.20 prikazuje slučaj korišćenja parkiranja. Poruka *parkiranje*, sa rednim brojem „1" šalje se od klase *kupac* klasi *servisna stanica*. Zatim se poruka *sledeće_slobodno* šalje od klase *servisna stanica* klasi *parkirno mesto*; i nosi broj „2". U ovom slučaju korišćenja postoji pet vrsta poruka.



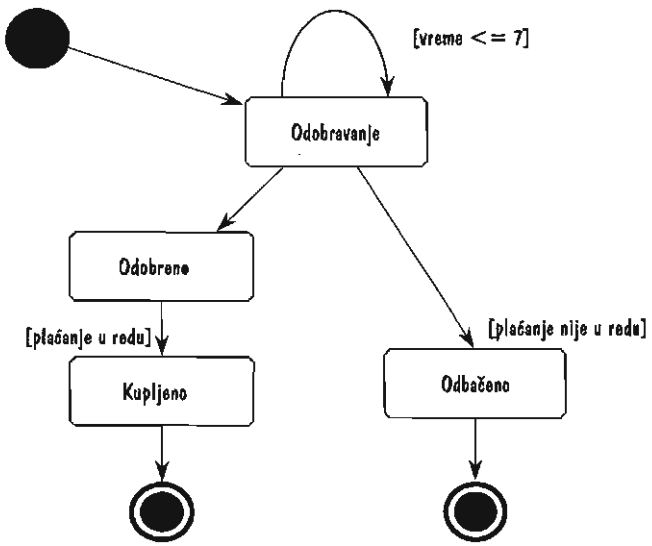
SLIKA 6.19 Dijagram sekvence za slučaj korišćenja *punjenje gorivom*



SLIKA 6.20 Dijagram saradnje za slučaj korišćenja *parkiranje*

Do sada smo koristili UML da bismo opisali statička svojstva sistema. UML podržava još dve vrste dijagrama za dokumentovanje dinamičkog modela sistema, *dijagram stanja* i *dijagram aktivnosti*. **Dijagram stanja** prikazuje sva moguća stanja u kojima objekat može da se nađe, događaje koji pokreću prelaze stanja i aktivnosti koje rezultiraju prelazom stanja. Obično je stanje objekta vezano za skup vrednosti atributa, a događaji nastupaju kada se poruke šalju ili primaju. Prema tome, dijagram stanja je potreban samo za one klase kod kojih objekti iskazuju dinamičko ponašanje, uz mnoštvo različitih vrednosti atributa i poruka.

SLIKA 6.21 Primer dijagrama stanja

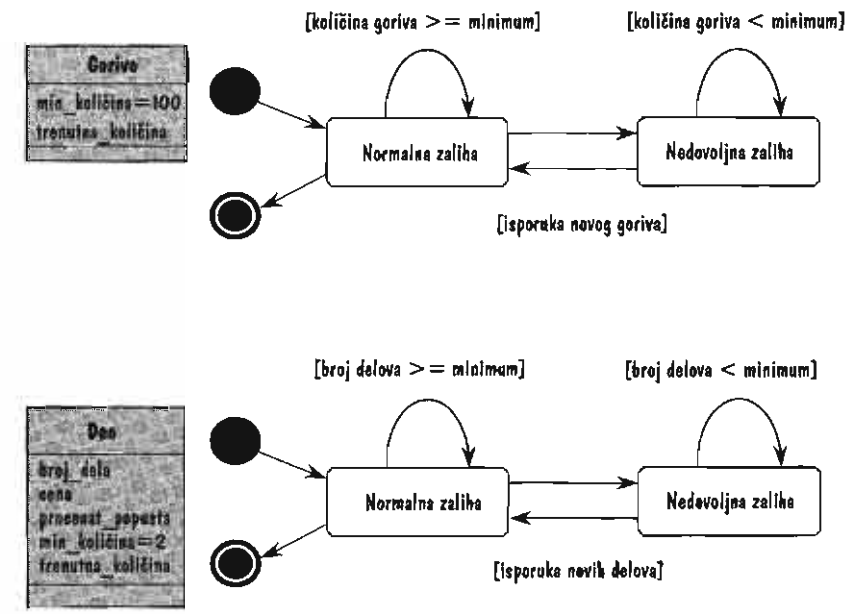


Notacija za dijagram stanja slična je notaciji za dijagrame prelaza stanja opisane u poglavlju 4. Kao što se vidi na slici 6.21, početni čvor je predstavljen crnom tačkom, a završni manjom crnom tačkom unutar bele tačke. Pravougaonik predstavlja stanje, a strelica pokazuje prelaz stanja. Uslov se beleži kao izraz u uglastoj zagradi pored strelice. Na primer, kupovina se ne smatra odobrenom ako funkcija odobravanja nije proizvela poruku „plaćanje u redu".

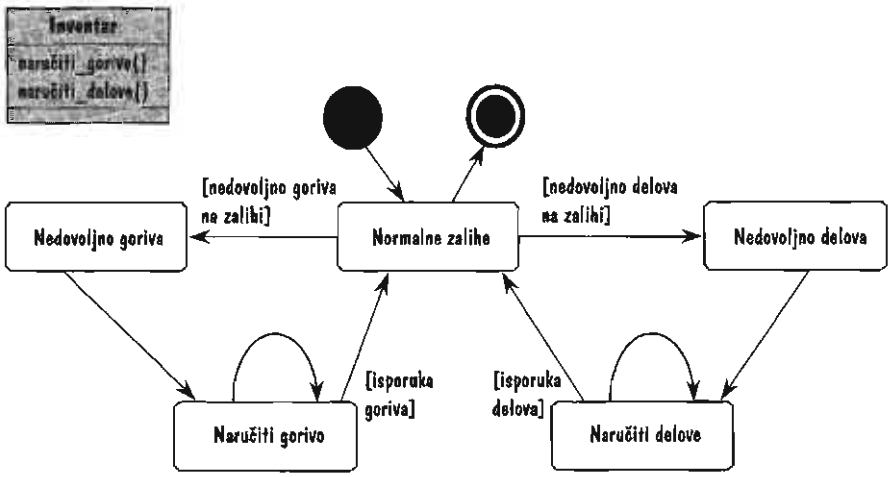
Možemo da napravimo dijagrame stanja za sistem servisne stanice Royal. Na primer, možemo da nacrtamo dijagram sa slike 6.22 za klase *gorivo* i *delovi*. Primetićete da je dijagram stanja za delove u gornjem delu slike identičan dijagramu stanja za gorivo u donjem delu slike, razlikuju se jedino po rečima „gorivo" i „delovi" koje su definisane u uslovima. Ta sličnost može da se upotrebi da bi se uprostio model. Umesto da pravimo dve odvojene, ali veoma slične funkcije za svaku klasu, te funkcije možemo staviti u klasu *inventar*.

Klasa *inventar*, prikazana na slici 6.23, naručuje delove ili gorivo; svako stanje pokreće drugačiji uslov. Klasa kontroliše da li su zalihe na normalnom nivou ili su pale ispod zadatog minimuma. Prema tome, modelovanje u UML notaciji pomaže da bolje uočimo probleme i mogućnosti za pojednostavljivanja dizajna. Kao što smo videli, dizajn može da se unapredi ako ga posmatramo iz različitih perspektiva.

UML koristi **dijagram aktivnosti** za modelovanje tokova postupaka ili aktivnosti u klasi. Kada za odlučivanje koju aktivnost treba pozvati postoje definisani uslovi, na dijagram aktivnosti stavljamo čvor odluke gde su predstavljene ponuđene mogućnosti. Na slici 6.24 prikazana je notacija koja se koristi za UML dijagram aktivnosti. Kao i u slučaju dijagrama stanja, početni čvor prikazuje se crnom tačkom, a završni manjom crnom tačkom unutar bele. Pravougaonik predstavlja aktivnost, a strelica prelazak sa jedne aktivnosti na drugu. U ovom primeru, pošto se završi aktivnost B, potrebno je doneti odluku. Ako je ispunjen jedan uslov, nekoj drugoj klasi se šalje X. Inače se poziva aktivnost C ili D. Duga horizontalna linija iznad aktivnosti C i D označava da poruka iz jedne aktivnosti (u ovom slučaju B) može difuzno da se emituje drugim aktivnostima (u ovom slučaju C i D). Zatim se C i D mogu pozvati pojedinačno, redom jedna pa druga, ili paralelno.

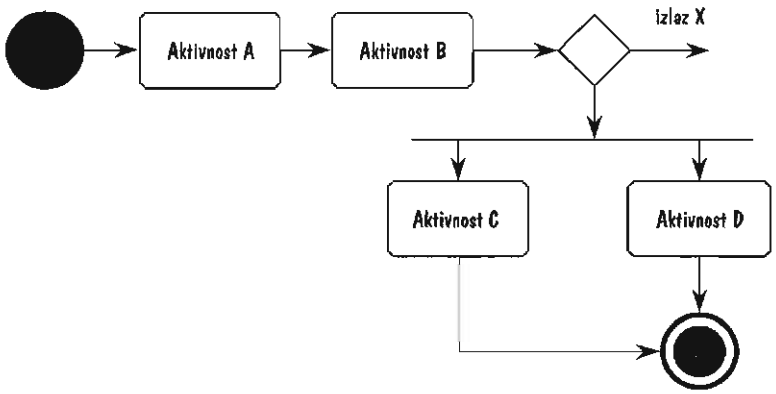


SLIKA 6.22 Dijagram stanja za klase *gorivo* i *delovi*



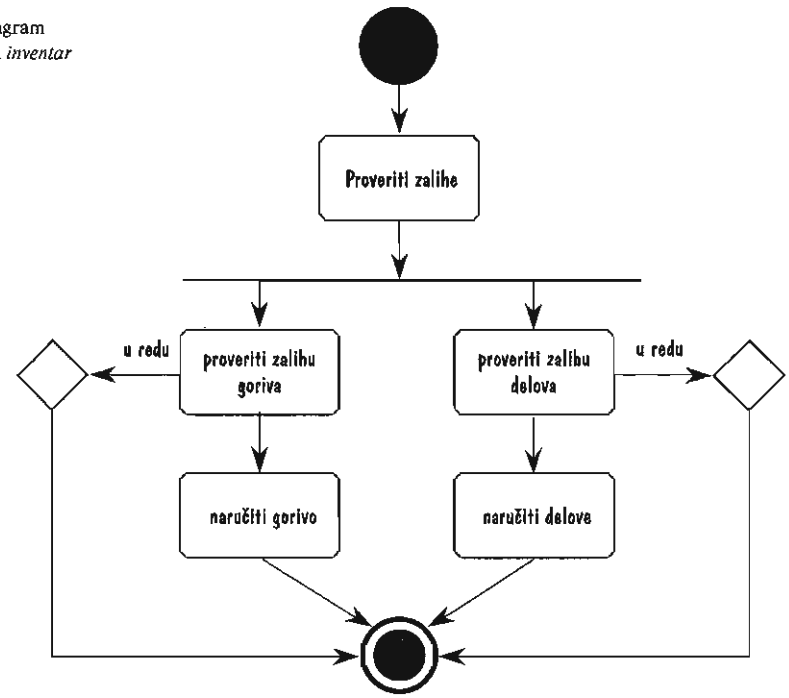
SLIKA 6.23 Dijagram stanja za klasu *inventar*

SLIKA 6.24
Notacija za dijagram aktivnosti



Nacrtaćemo dijagram aktivnosti za klasu *inventar* koju smo napravili dok smo posmatrali dijagrame stanja. Kao što vidimo na slici 6.25, dijagram aktivnosti sadrži dve odluke: jednu za proveravanje da li ima dovoljno goriva, a drugu za proveravanje da li ima dovoljno delova na zalihama. Ako su zalihe bilo čega nedovoljne, dijagram aktivnosti pokazuje da se pozivaju aktivnosti za naručivanje delova i goriva. Obratite pažnju na to da horizontalna linija dozvoljava pokretanje obe aktivnosti. Takva situacija može da nastupi ako su istom kupcu potrebni i delovi i gorivo. Na primer, kupac može da dođe u servisnu stanicu Royal i požali se na veliku potrošnju goriva (tj. broj pređenih kilometara po galonu je manji nego obično). Mani i njegova ekipa mogli bi da otkriju da je rezervoar za gorivo probušen. U tom slučaju će odlučiti da montiraju novi rezervoar za gorivo i da ga napune gorivom. U ovom slučaju može doći do automatskog naručivanja goriva i rezervoara za gorivo ukoliko sistem inventara utvrdi da su preostale zalihe nedozvoljeno niske.

SLIKA 6.25 Dijagram aktivnosti za klasu *inventar*



6.6 OO DIZAJN PROGRAMA

Pošto smo završili dizajn sistema, njegove rezultate možemo da upotrebimo za dizajn programa: dizajn iz kojeg će koderi implementirati sistem. Naš rad na dizajnu programa polazi od objekata i klasa iz dizajna sistema, ali ih moramo doraditi i dopuniti dodatnim razmatranjima:

- nefunkcionalni zahtevi kao što su performansa i U/I ograničenja;
- višekratna upotrebljivost komponenti iz ranije projektovanih sistema;
- višekratna upotrebljivost komponenti namenjenih korišćenju u budućim sistemima;
- zahtevi korisničkog interfejsa;
- struktura podataka i detalji u vezi sa upravljanjem.

U dizajnu programa može da se pojavi više objekata nego u dizajnu sistema. Na primer, mnoge objekte iz programskog dizajna korisnik ne vidi i za mnoge ne postoje odgovarajući objekti u realnom svetu. U dizajnu sistema može da se koristi opis višeg nivoa za način organizovanja i pristupanja podacima, ali detaljni dizajn sistema sadrži informacije o strukturi podataka (na primer, o povezanim listama ili matricama) koje su potrebne programerima. U fazama specifikacije zahteva i dizajna sistema verovatno nećemo sa kupcima raspravljati o strukturi podataka; te strukture nisu vidljive, nisu neophodne za razumevanje problema, niti za opis rešenja na višem nivou apstrakcije. Ali, u fazi dizajna programa moramo doneti odluke o detaljima u vezi sa podacima. Prema tome, nivo apstrakcije na kojem radimo tokom dizajna programa razlikuje se od nivoa apstrakcije tokom projektovanja sistema.

U fazi dizajna programa moramo takođe da odredimo osobine pojedinačnih interfejsa svakog objekta sa ostatkom sveta. Konkretno, moramo definisati **signaturu operacije** za svaku operaciju. Svakoj operaciji dodeljujemo naziv, objekte koje uzima kao parametre i vrednosti koje vraća. U mnogim slučajevima te informacije možemo da izvedemo iz dijagrama sekvence. **Interfejs** jednog objekta predstavlja skup svih signatura njegovih operacija. Pošto definišemo interfejse, možemo da ih klasifikujemo po tipu i izgradimo hijerarhiju tipova interfejsa u kojoj neki interfejsi nasleđuju osobine drugih. Ova hijerarhija je posebno važna pošto su objekti vidljivi i dostupni drugim objektima jedino preko interfejsa.

Interfejsi nam takođe omogućavaju da iskoristimo svojstvo polimorfizma. Kao što smo videli, objekat može da primi zahtev i protumači ga različito za različite primerke. Na primer, objekat *poligon* može drugačije da izračunava površinu trougla a drugačije površinu kvadrata. Kada se zahtev pridružuje operaciji konkretnog objekta u trenutku izvršenja, imamo OO sistem sa **dinamičkim vezivanjem**. Slično tome, različiti objekti mogu da prime istu vrstu zahteva, ali da ih obrade na različite načine. Na primer, u servisnoj stanici Royal zahtev za poručivanje upućen objektu za poručivanje delova može drugačije da se obradi u odnosu na zahtev za poručivanje goriva.

Nakon definisanja interfejsa sledeći zadatak je implementacija. Setite se da je svaki objekat primerak neke klase. Kada se pravi primerak konkretnog objekta, prevodilac alocira memoriju za podatke potrebne tom objektu; zatim se tim podacima pridružuju operacije. Mada je implementiranje uglavnom zadatak programera, dizajner programa je taj koji odlučuje da li će i na koji način klase koristiti prednosti mehanizma nasleđivanja. Na primer, dizajneri programa obično biraju između komponovanja objekata ili mehanizma nasleđivanja klasa. Kada jedna klasa nasleđuje svojstva neke druge klase, unutrašnje karakteristike roditeljske klase obično su vidljive klasi potomku; zbog toga nasleđivanje ponekad nazivamo **višekratnim korišćenjem tipa bele kutije**. Međutim, klase mogu da se grade i od komponenti, što pomalo podseća na pravljenje strukture od lego kockica. Ovde se izgradnja komponovanjem često naziva **višekratnim korišćenjem tipa crne kutije**, zato što projektantu nisu vidljive unutrašnje karakteristike gradivnih elemenata.

Svaka paradigma konstrukcije ima svoje prednosti i mane. Pristup sa nasleđivanjem daje statički prikaz sistema u fazi prevođenja, dok vidljivost obezbeđuje lakše i jednostavnije izmene. Međutim, nasleđivanje ne može lako da se menja u fazi izvršavanja, a promene roditeljske klase mogu da utiču na potomke. U stvari, značajne promene mogu da dovedu do složenih hijerarhija klasa u kojima je teško pratiti i održavati odnose zavisnosti.

S druge strane, pristup sa komponovanjem nameće ugradnju enkapsulacije u dizajn sistema. To jest, ono što je zamišljeno da bude sakriveno, ostaje sakriveno. Štaviše, komponovanje nam omogućava da lako zamenimo jednu ugrađenu komponentu drugom iste vrste; ta tehnika je slična kao da zamenjujemo crvenu lego kockicu plavom, samo ako su im veličina i oblik isti. Zbog toga nas manje brine zavisnost pri promeni sistema izgrađenog komponovanjem, nego u slučaju modifikacije sistema konstruisanog mehanizmom nasleđivanja. Međutim, i kompozicija ima svoje probleme. Sistemi projektovani komponovanjem se dinamički definišu u vreme izvršavanja, pa nije uvek jasno koji objekti referenciraju druge objekte. Interfejsi objekata moraju pažljivo da se projektuju da ne bi ograničenja interfejsa sprečila upotrebu datog objekta u trenutku kada je on potreban.

Jedan od načina da se ublaže efekti komponovanja i da ono bude sličnije mehanizmu nasleđivanja jeste omogućavanje objektu da svoje operacije delegira drugom objektu. Na primer, pretpostavimo da objekat A šalje poruku objektu B. Kao primalac poruke, B može da pošalje sebe delegiranom objektu C, a delegirana operacija ponovo referencira objekat B. Takva struktura izgleda nezgrapna i teška za razumevanje, ali u suštini ona omogućava da se ponašanja komponuju u fazi izvršavanja i da se lako menjaju ako se ukaže potreba.

Pomoćni elementi u projektovanju

Nijedan način projektovanja nije najbolji za sve situacije. Jedini savet koji važi za sve sisteme je sledeći: planirajte promene. Bez obzira na to koji sistem gradite, on će se verovatno u nekom trenutku promeniti. Postoje mnoge OO tehnike kojima se dobijaju fleksibilniji sistemi koji se lako održavaju. Osim biranja strategije fleksibilnog dizajna možete koristiti komplete alatki za izgradnju sistema. **Komplet alatki** je skup povezanih višekratno upotrebljivih klasa koje omogućavaju dobro definisani skup funkcionalnosti. Kompleti alatki su veoma slični bibliotekama podrutina u proceduralnim jezicima. Na pri-

mer, umesto da sami pišete kôd od nule, možete da upotrebite komplet alata i napravite korisnički interfejs pomoću dugmadi i prozora.

Okvirna rešenja i šabloni takođe predstavljaju pomoćne elemente u projektovanju. Oni se razlikuju od kompleta alatki po tome što su više usredsređeni na višekratno korišćenje dizajna nego na višekratno korišćenje koda. **Šablon** je model apstraktnih arhitektonskih elemenata koji se može koristiti kao smernica u projektovanju. Prvobitno zamišljeni za opisivanje strukturnih elemenata građevina i gradova (Alexander 1979), oni poseduju osobine koje su opšteprihvaćene kao dobar dizajn. Isto kao što je Alexander kodifikovao najbolju arhitektonsku praksu, dizajneri softvera su sredinom 1990-ih uočili neke probleme koji se stalno ponavljaju u dizajnu softvera i predložili su šablone za njihovo rešavanje. Na primer, pretpostavimo da projektujete sistem za interaktivno prikazivanje uskladištenih vrednosti. Gamma i dr. (1995) predlažu upotrebu šablona *model-pogled-kontroler*. *Model* je skup klasa koje sadrže uskladištene informacije. *Pogled* je skup klasa koje implementiraju korisnički interfejs. *Kontrolerske* klase kontrolišu međudejstva korisničkog interfejsa i uskladištenih podataka.

Svaki šablon poseduje kontekst i dejstva. **Kontekst** objašnjava situacije u kojima se taj šablon može primeniti. **Dejstva** predstavljaju elemente konteksta koji se donekle menjaju. Ako se vaša situacija poklapa sa dejstvima i kontekstom jednog šablona, onda će on odgovarati vašoj aplikaciji. Međutim, šabloni nisu toliko korisni ako limitiraju fleksibilnost razmatrane situacije. U podsetniku 6.2 navedeno je nekoliko mogućnosti za pravljenje šablona.

Okvirno rešenje je ponovno iskoristivi deo dizajna pogodan za određeni domen primene. Specijalizovanije je od šablona i u svojoj specifikaciji može da sadrži šablone. Okvirno rešenje se proširuje u granicama projektovanog sistema. Proširenje uključuje pravljenje dodatnih klasa bilo nasleđivanjem ili komponovanjem, u cilju proširenja postojeće funkcionalnosti unutar okvirnog rešenja. Prema tome, šabloni više liče na skele, dok je okvirno rešenje samostalni projekat u koji morate da ugradite neke od glavnih delova koji nedostaju.

PODSETNIK 6.2 PRISTUPI U PRAVLJENJU ŠABLONA

Gamma i dr. (1995) predlažu nekoliko načina da se naprave šabloni:

Apstraktna fabrika (Abstract Factory) Predstavlja interfejs za pravljenje familija logički ili funkcionalno povezanih objekata bez navođenja njihovih konkretnih klasa.

Graditelj (Builder) Razdvaja izgradnju složenog objekta od njegove predstave da bi isti proces mogao da proizvede različite predstave.

Proizvodni metod (Factory Method) Definiše interfejs za pravljenje objekta pri čemu potklasa donosi odluku o instanciranju.

Prototip (Prototype) Koristi instancu prototipa za kreiranje novog objekta kopiranjem prototipa.

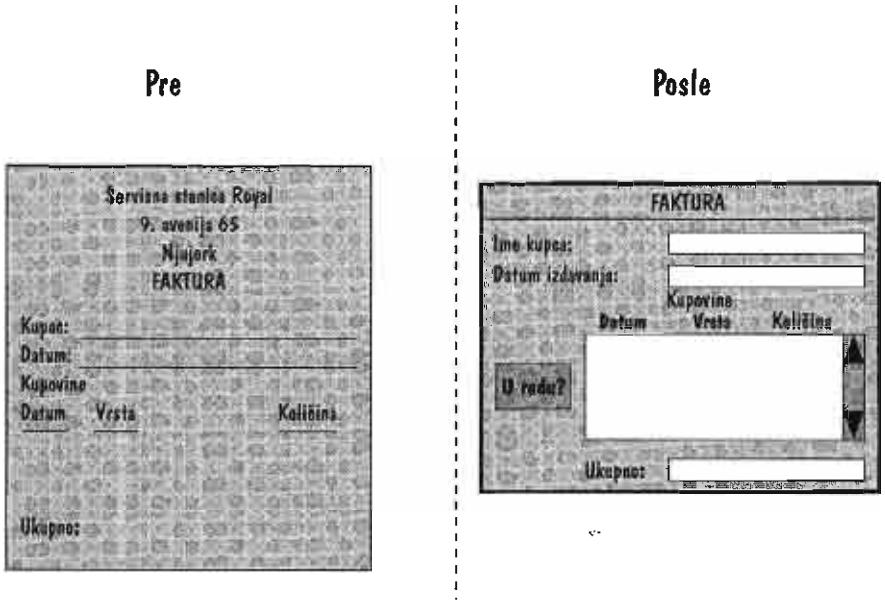
Unikat (Singleton) U ovom slučaju klasa poseduje samo jednu instancu sa globalno definisanom pristupnom tačkom.

Dizajn korisničkog interfejsa

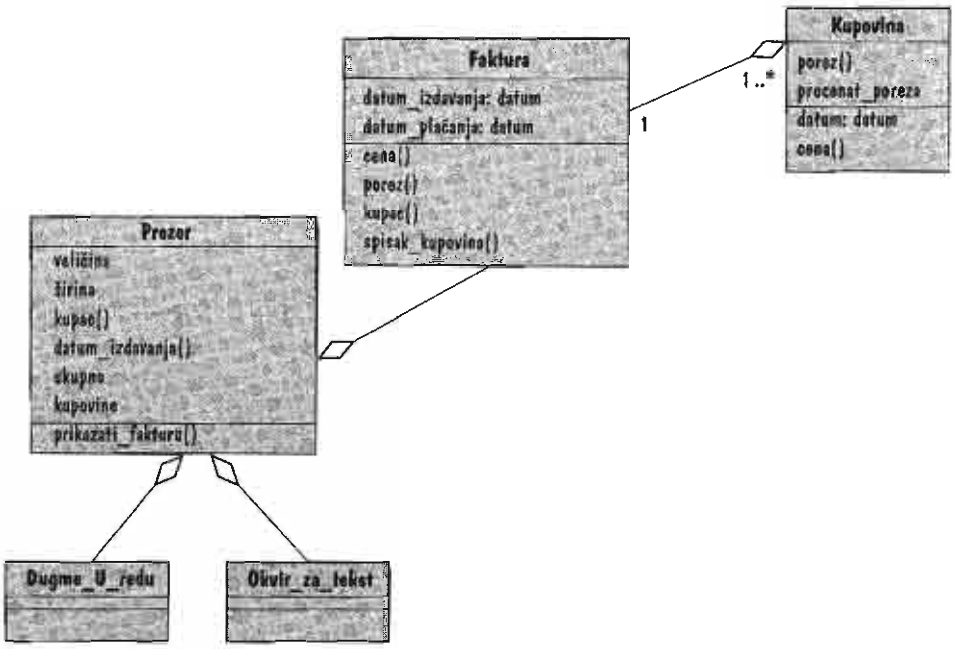
Pažljivije pogledajmo nekoliko aspekata OO dizajna programa, počevši od korisničkog interfejsa. Da bismo dovršili deo programskog dizajna koji se odnosi na korisnički interfejs moramo da se pozabavimo sledećim pitanjima:

- kakvi treba da budu ljudi koji će raditi sa sistemom;
- scenarijima za svaki od načina na koji sistem može da obavi zadatak;
- projektom hijerarhije korisničkih komandi;
- rafinisanjem sekvenci interakcija korisnika sa sistemom;
- projektom relevantnih klasa u hijerarhiji za implementaciju odluka vezanih za dizajn korisničkog interfejsa;
- integrisanjem klasa korisničkog interfejsa u sveukupnu hijerarhiju klasa u sistemu.

Prvi korak u dizajnu korisničkog interfejsa je prikazivanje interakcije oslanjajući se na papirne dokumente. Da bi se to postiglo, neophodno je utvrditi tokove papirnih dokumenata u postojećem sistemu; papirni dokumenti mogu da ukažu na adekvatne korisničke interfejse u sklopu automatizovanog sistema. Na primer, na levoj strani slike 6.26 prikazana je faktura u papirnom obliku, koju Mani koristi u servisnoj stanici Royal. Da bismo automatizovali postupak fakturisanja, možemo da predložimo ekransku formu prikazanu na desnoj strani slike.



SLIKA 6.26 Prelaz sa papira na ekransku formu



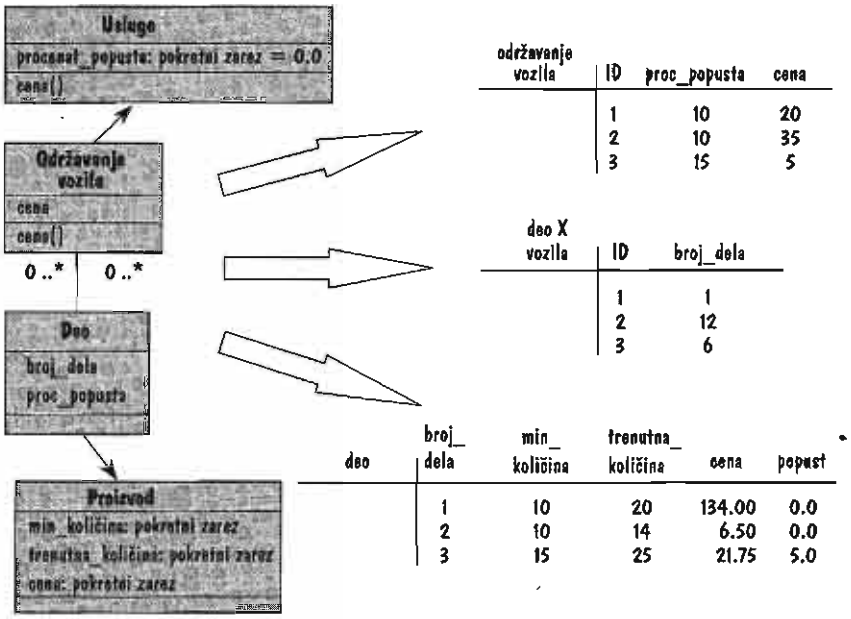
SLIKA 6.27 Mogući model ekrana za fakturisanje

Ako Mani pristane na ponuđeni dizajn ekrana, sledeći korak će biti projektovanje jedne klase ili više klasa za implementaciju tog ekrana. Možda ćete odlučiti da upotrebite model prikazan na slici 6.27. Primetićete da ovaj model sadrži klase za dugme <OK> i za polje za unos teksta. Te objekte verovatno niste pominjali u vašim početnim razgovorima sa Manijem dok ste na osnovu njegovih zahteva izvodili slučajeve korišćenja. Oni odražavaju aspekte vašeg rešenja, a ne njegovog problema. Uobičajeno je da se skup objekata i klasa širi u životnom ciklusu od razumevanja problema do izrade rešenja. Ovo povećanje broja klasa i usložnjavanje hijerarhije predstavlja važan razlog zbog kojeg prilikom definisanja rešenja treba imati na umu fleksibilnost dizajna.

Projektovanje rukovanja podacima

Dizajn programa mora takođe da se bavi načinom čuvanja i obnavljanja trajnih objekata. Upravljanje podacima treba da zadovolji zahteve sistema u pogledu performanse i potrebnog prostora. Dizajn objekata i njihovih operacija formulišemo zavisno od shvatanja zahteva i ograničenja vezanih za podatke. Taj zadatak možemo da obavimo u četiri koraka, a to su:

1. Prepoznavanje podataka, strukture podataka i njihovih međusobnih relacija.
2. Projektovanje usluge za rukovanje strukturom podataka i relacijama.
3. Pronalaženje alatki, kao što su sistemi za upravljanje bazama podataka, za implementiranje nekih poslova vezanih za rukovanje podacima.
4. Projektovanje klasa i hijerarhija klasa za nadgledanje funkcija rukovanja podacima.



SLIKA 6.28 Implementiranje klasa pomoću relacione baze podataka

Objektno orijentisano rešenje može da koristi obične datoteke ili relacione baze podataka, ali se najlakše povezuje sa objektno orijentisanim bazama podataka. Primer tog povezivanja je praćenje održavanja vozila i delova dato na slici 6.28. Kad bismo koristili obične datoteke, za izvršavanje zadataka koji se zahtevaju od sistema morali bismo da odredimo po jednu datoteku za svaku klasu i programski rešimo veze među datotekama. Sa relacionom bazom podataka bi nam posao bio nešto lakši; kao što se vidi na slici, treba urediti tabele, a potrebne su i dodatne tabele za evidentiranje relacija između klasa. Na primer, ne bi nam bila dovoljna tabela delova i tabela održavanja, već bi nam pored klasa koje smo već definisali bila potrebna i tabela koja sadrži veze delova sa vozilima. Objektno orijentisana ili OO relaciona baza podataka je najlakše rešenje zato što za implementiranje upravljanja podacima koristi klase koje već postoje u rešenju.

Projektovanje rukovanja poslovima

Upravljanje poslovima je bitan deo dizajna programa. Moramo pažljivo da analiziramo zahteve i utvrdimo kako treba koordinirati aktivnosti koje sistem mora da izvršava. **Posao** je proces u sistemu; on može biti upravljan događajima ili vremenom. **Posao koga pokreće događaj** aktivira se kada nastupi konkretan događaj kao što je na primer, pomeranje miša, pritisak na dugme ili kada neki senzor pošalje, odnosno primi poruku. **Posao koji pokreće vreme** poziva se u određenom trenutku, na primer, svakih *n* minuta.

Upravljanje poslovima možemo da projektujemo u četiri koraka, a to su:

1. Prepoznavanje poslova koje treba izvršiti i utvrđivanje da li ih pokreću događaji ili vreme.
2. Određivanje prioriteta poslova. Za svaki par poslova treba utvrditi koji od njih ima prednost, ako se istovremeno pozovu.

3. Projektovanje zadatka koji koordinira ostale.
4. Projektovanje objekata za svaki posao i definisanje relacija između njih.

Svaki posao mora biti formalno definisan da bi programeri znali kako da ga pravilno implementiraju. Kao pomoć programerima trebalo bi da navedemo ime posla, opis, prioritet, usluge, mehanizam komunikacije i mesto u hijerarhiji. Na primer, posao naručivanja delova mogao bi da ima sledeću definiciju:

Ime posla: naručivanje delova

Opis: svrha je da se automatski naruče novi delovi kada količina na zalihi padne ispod minimalne.

Prioritet: visok. Mora se aktivirati kada inventar izda upozorenje da su zalihe nedovoljne.

Uključene usluge: proveriti inventar.

Upravljanje: sistem servisne stanice.

Komunikacija: putem modema sa sistemom naručivanje_delova.

U knjizi Gamma i dr. (1995) opisan je niz šablona koji mogu da nam pomognu u odlučivanju kako da upravljamo poslovima; oni su navedeni u podsetniku 6.3. Ispitaćemo detaljno jedan od šablona da biste videli njihovu primenu na dizajn programa.

Šablon *posmatrač* koristan je kada neka apstrakcija ima bar dva aspekta i kada su neki od njih zavisni od drugih. On preporučuje da se među objektima definiše zavisnost „jedan prema više" tako da kada jedan objekt promeni stanje, svi zavisni objekti dobiju obaveštenje o tome i automatski se ažuriraju. Enkapsuliranje aspekata u zasebne objekte ima mnoge prednosti. Kada promena u jednom objektu zahteva promene u neodređenom broju drugih objekata, enkapsuliranje olakšava nezavisnu promenu i višekratnu upotrebljivost objekata. Tako objekti nisu čvrsto vezani; jedan objekt može da obavesti druge objekte i ne mora da zna kojim objektima je obaveštenje potrebno.

Šablon *posmatrač* zahteva četiri glavna elementa: *subjekt*, *posmatrač*, *konkretan subjekt* i *konkretan posmatrač*. **Subjekt** je svestan svojih posmatrača i nudi interfejs za povezivanje i razdvajanje objekata posmatrača. Glavna funkcija **posmatrača** jeste da definiše interfejs za ažuriranje objekata koje treba obavestiti o promenama subjekta. **Konkretan subjekt** čuva stanje koje je važno za jedan ili više objekata konkretnih posmatrača i obaveštava svoje posmatrače kada se to stanje promeni. **Konkretan posmatrač** održava referencu prema konkretnom subjektu; čuva stanje koje treba da ostane konsistentno sa stanjem subjekta i u skladu sa tim ažurira interfejs. Na slici 6.29 prikazani su odnosi ta četiri elementa.

Kako ti elementi sarađuju može se prikazati dijagramom sekvence prikazanim na slici 6.30. Objekti posmatrači posmatraju jednog subjekta ili više njih. Nastojimo da izbegnemo situaciju kada stanje zabeleženo kod posmatrača nije konsistentno sa stvarnim stanjem subjekta. Dijagram saradnje pokazuje da konkretan subjekt obaveštava svoje posmatrače kad god dođe do promene zbog koje bi stanje kod posmatrača postalo nekonsistentno sa njegovim stanjem. Kada konkretan posmatrač primi takvo obaveštenje, on može od subjekta da zatraži dodatne informacije. Konkretan posmatrač zatim koristi te informacije u cilju izjednačavanja evidentiranog i stvarnog stanja.

PODSETNIK 6.3 ŠABLONI ZA UPRAVLJANJE ZADACIMA KOJE SU UVELI GAMMA I SARADNICI

Lanac odgovornosti (Chain of Responsibility) Izbegava neposredno vezivanje pošiljaoca zahteva sa njegovim primaocem, dajući šansu da više objekata rukuje zahtevom. Ulančava objekte primaoce i prosleđuje zahtev niz lanac dok ga neki objekat ne obradi.

Komanda (Command) Enkapsulira zahtev kao objekt, čime omogućava da sistem parametrizuje klijente sa različitim zahtevima, redove zahteva na čekanju ili zahteve za formiranje dnevnika rada, i da podrži operacije koje se ne mogu obaviti.

Interpretator (Interpreter) Za dati jezik, definiše predstavljanje njegove gramatike i interpretera koji tu predstavu koristi za interpretiranje iskaza na tom jeziku.

Iterator (Iterator) Obezbeđuje sekvencijalni pristup elementima objekta agregata, nezavisno od njegove unutrašnje predstave.

Posrednik (Mediator) Definiše objekat koji enkapsulira interakciju skupa objekata. Posrednik uvodi slabo vezivanje, jer sprečava eksplicitno referenciranje objekata i omogućava nezavisnu izmenu načina njihove interakcije.

Podsetnik (Memento) Bez narušavanja enkapsulacije, preuzima i čini dostupnim unutrašnje stanje objekta da bi kasnije to stanje objekta restaurirao.

Posmatrač (Observer) Definiše zavisnost jedan prema više među objektima tako da, prilikom promene stanja jednog objekta, svi zavisni objekti dobiju poruku o tome i automatski se ažuriraju.

Stanje (State) Omogućava da objekat promeni ponašanje kada mu se promeni unutrašnje stanje. Spolja deluje kao da je objekat promenio pripadnost klasi.

Strategija (Strategy) Definiše familiju algoritama, enkapsulira svaki od njih i omogućava njihovu međusobnu zamenu. Strategija omogućava da se algoritam menja nezavisno od klijenata koji ga koriste.

Šablonski metod (Template Method) Definiše kostur nekog algoritma u sklopu operacije, prepuštajući implementaciju određenih koraka potklasi. Šablonski metod dozvoljava da potklase redefinišu neke korake algoritma ne menjajući njegovu strukturu.

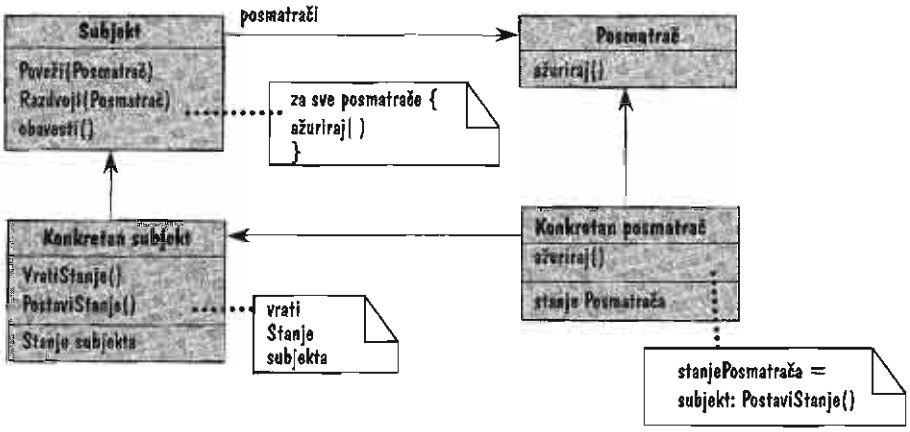
Posetilac (Visitor) Predstavlja operaciju koju treba izvesti nad elementima strukture nekog objekta. Posetilac dozvoljava definisanje nove operacije bez izmena klasa elemenata nad kojima radi.

Ova vrsta šablona dozvoljava apstraktno vezivanje između jednog subjekta i jednog posmatrača i podržava komunikaciju emisionog tipa. Međutim, subjekti i objekti moraju često da nadgledaju svoja stanja u cilju sprečavanja nekonsistentnosti koje mogu nastati kao posledica neočekivanih ažuriranja.

6.7 OO MERENJA

Kada posedujemo OO implementaciju sistema, kako možemo da merimo njegove osobine? Na primer, u poglavlju 5 smo videli da postoje poželjne karakteristike dizajna kao što su slaba spregnutost i jaka kohezija, kao i mogućnost da se mere osobine kao što je

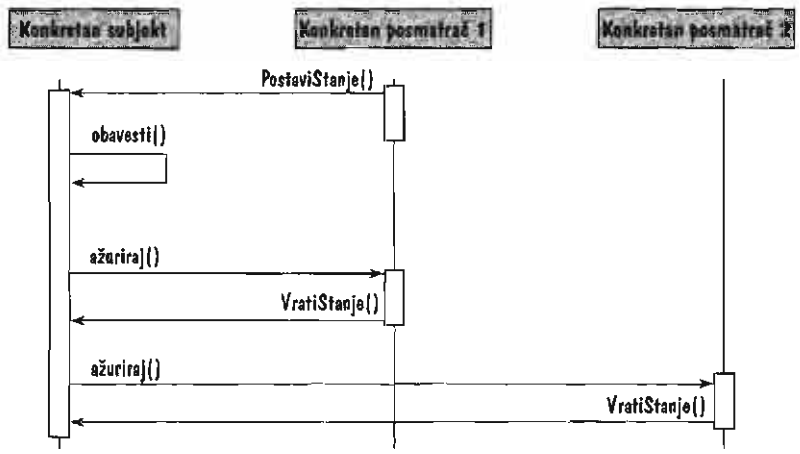
složenost dizajna. Kako se mere takve karakteristike OO sistema? Da li su ta merenja korisna za sagledavanje, kontrolu i predviđanje? U ovom odeljku ispitujemo relativno novo polje OO merenja.



SLIKA 6.29 Relacije između komponenti šablona *posmatrač*

OO mere veličine

Kao što smo napomenuli, OO sistemi postaju sve veći kako idemo od analize zahteva prema dizajnu, a zatim prema kodiranju i testiranju. U tom pogledu se ovi sistemi ne razlikuju od sistema koji se razvijaju oslanjajući se na druge paradigme. Međutim, za razliku od ostalih, kod OO pristupa se koristi isti ili sličan jezik u svim fazama životnog ciklusa, te je on zato pogodan za iskazivanje veličine istim merama od samog početka razmatranja sistema do njegove isporuke i održavanja.



SLIKA 6.30 Dijagram sekvence za šablon *posmatrač*

Istraživači su iskoristili taj zajednički rečnik za merenje veličine i na veličini zasnovanih predviđanja. Na primer, Pfleeger (1991) je koristila objekte i metode kao osnovne jedinice mere za njen pristup proceni napora; utvrdila je da je za predviđanje ukupnih napora potrebnih za izgradnju nekoliko OO sistema njen metod bio precizniji od metoda COCOMO. Olsen (1993) je primenio istu tehniku na jedan komercijalni projekt i procene su bile izuzetno precizne. Jasna je prednost jednoobrazne procene veličine tokom celog razvoja; tehnika procene može se primenjivati tokom celog razvoja, a ulazi za procene su direktno uporedivi. Drugim rečima, lako je oceniti početnu veličinu i pratiti je kako proizvod raste.

Lorenz i Kidd (1994) proširili su OO merenje veličine u smislu dodavanja finijih detalja. Oni su definisali devet aspekata veličine koji odražavaju ne samo ukupnu „masu" sistema, već i način na koji karakteristike klasa utiču na proizvod. Prvo posmatraju proizvod. Svakom slučaju korišćenja pridružuju scenario jednog posla koji sistem izvršava. Lorenz i Kidd mere **broj skriptova scenarija** (NSS) u slučajevima korišćenja. Utvrdili su da je ta mera u direktnoj vezi sa veličinom aplikacije i brojem test slučajeva. Iz tog razloga, NSS je koristan na bar dva načina. Kao mera veličine može se uključiti u model za predviđanje potrebnih napora i trajanja. Kao procena broja test slučajeva pomaže da testerski tim pripremi primere i dodeli resurse neophodne za aktivnosti testiranja koje slede.

Lorenz i Kidd koriste i **broj osnovnih klasa** (tj. domenskih klasa) u sistemu. Ta mera namenjena je proceni dizajna na visokom nivou apstrakcije i ukazuje na to koliko će rada biti potrebno za izgradnju sistema. Oni takođe određuju i **broj klasa za podršku** kada na osnovu dizajna na nižem nivou apstrakcije procenjuju potreban rad. **Prosečan broj klasa za podršku po osnovnoj klasi** i **broj podsistema** korisni su za praćenje strukture sistema koji se razvija.

Oni takođe definišu veličinu sa aspekta pojedinačnih klasa. **Veličina klase** je zbir ukupnog broja operacija i broja atributa; ovde osim svojstava pojedinačne klase brojimo i nasleđena svojstva. Da bi procenili uticaj nasleđivanja, oni koriste **broj operacija koje potklasa redefiniše** (NOO), kao i **broj operacija koje uvodi potklasa**. Pomoću ovih brojeva oni definišu **indeks specijalizacije**, SI, kao

$$SI = (NOO \times nivo)/(ukupan\ broj\ metoda\ klase)$$

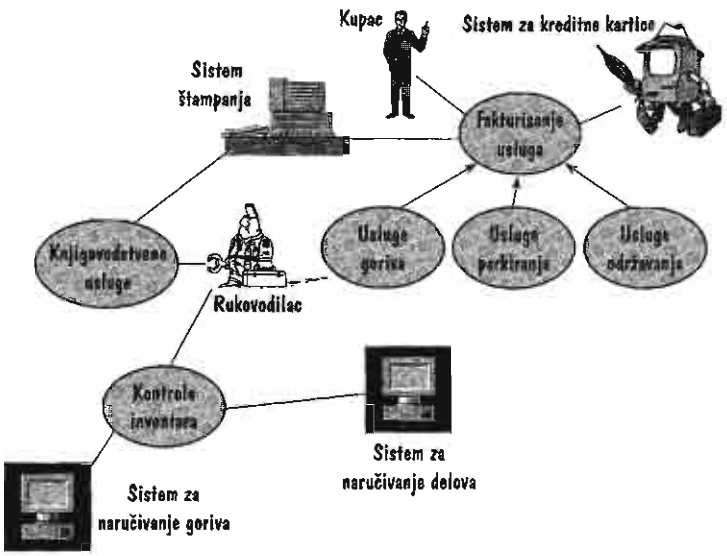
Svaka od ovih mera može se primeniti u različitim fazama razvoja, kao što se vidi u tabeli 6.5. Broj skriptova za slučajeve korišćenja i broj osnovnih klasa mogu se meriti u ranim fazama prikupljanja zahteva. Ta vrsta mere veličine obično je konkretnija i preciznija od šema za merenje zahteva koje se koriste u tradicionalnim metodima razvoja i korisna je rukovodiocima koji moraju na vreme da dodele resurse.

Primenićemo ova merenja na problem servisne stanice Royal. Na slici 6.31 dat je opšti pregled naše objektno orijentisane analize problema. Kada ponovo pregledamo podsetnik 6.1 i elipse na ovoj slici, videćemo da postoji šest elipsi koje predstavljaju šest osnovnih slučajeva korišćenja koje planiramo za ovaj sistem. Prema tome, prema metrici Lorenza i Kidda broj skriptova u scenariju jednak je 6.

Tabela 6.5 Kolekcija mera Lorenza i Kidda u različitim fazama razvoja

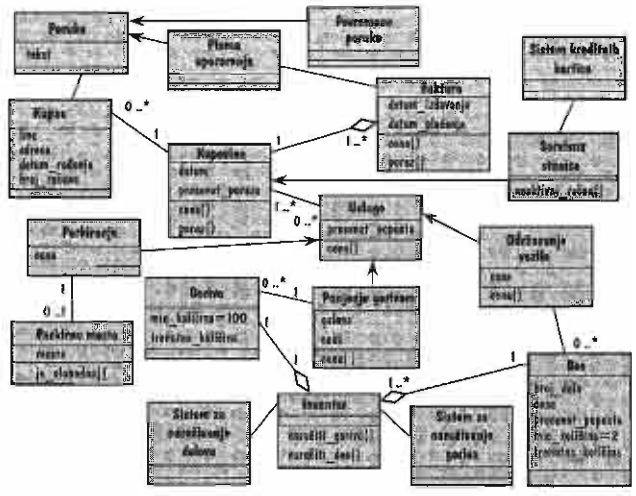
Metrika	Faza:	Opis zahteva	Dizajn sistema	Dizajn programa	Kodiranje	Testiranje
Broj skriptova scenarija		X				
Broj osnovnih klasa		X	X			
Broj klasa za podršku				X		
Prosečan broj klasa za podršku po jednoj osnovnoj klasi				X		
Broj podsistema				X	X	
Veličina klase			X	X	X	
Broj operacija koje potklasa redefiniše.			X	X	X	X
Broj operacija koje potklasa dodaje			X	X	X	
Indeks specijalizacije			X	X	X	X

Neke od ostalih mera možemo da odredimo analizom hijerarhije klasa u modelu sistema koji smo prikazali ranije u ovom poglavlju. Na slici 6.32 ponovo prikazujemo tu hijerarhiju i navodimo minimalne i maksimalne vrednosti za svaku meru Lorenza i Kidda. Te mere možemo koristiti kao polaznu liniju od koje ocenjujemo rast tokom razvoja sistema. Te mere mogu takođe da ukažu na verovatni uticaj predloženih promena, a možemo i da odredimo da li će promena otežati testiranje i razumevanje.



SLIKA 6.31 Objektno orijentisani pogled na servisnu stanicu Royal

Metrika	Min	Max
Broj osnovnih klasa	17	17
Veličina klase	1	7
Broj nadjačanih operacija	0	1
Broj dodatih operacija	0	0
Indeks specijalizacije	0	1



SLIKA 6.32 Hijerarhija klasa za servisnu stanicu Royal, plus metrika Lorenza i Kidda.

Mere u OO dizajnu

Chidamber i Kemerer (1994) razvili su i niz mera za objektno orijentisani razvoj. Njihov rad je više usredsređen na dizajn nego na procenu veličine, tako da dopunjava rad Lorenza i Kidda. Pored mera veličine kao što su vrednosni metodi po klasi, dubina nasleđivanja i broj potomaka, Chidamber i Kemerer mere i spregnutost objekata, reakcije klase i nedostatak kohezije kod metoda. U tabeli 6.6 vidimo u kojim fazama razvoja može da se primeni svaka od ovih mera. Pošto su u širokoj upotrebi i prerasle su u standard sa kojim se porede druge OO mere, detaljnije ćemo analizirati neke od njih.

U poglavlju 5 smo napomenuli da su neki aspekti kvaliteta dizajna merljivi, na primer složenost. Kao što ćemo videti u poglavlju 11, postoji više načina da se izmeri složenost konkretnog koda. Chidamber i Kemerer ostavljaju definiciju složenosti otvorenu tako da projektanti mogu da izaberu meru složenosti koja je prikladna za dati projekat. Chidamber i Kemerer koriste složenost kao vrednostni faktor u izračunavanju **vrednosnih metoda po klasi**:

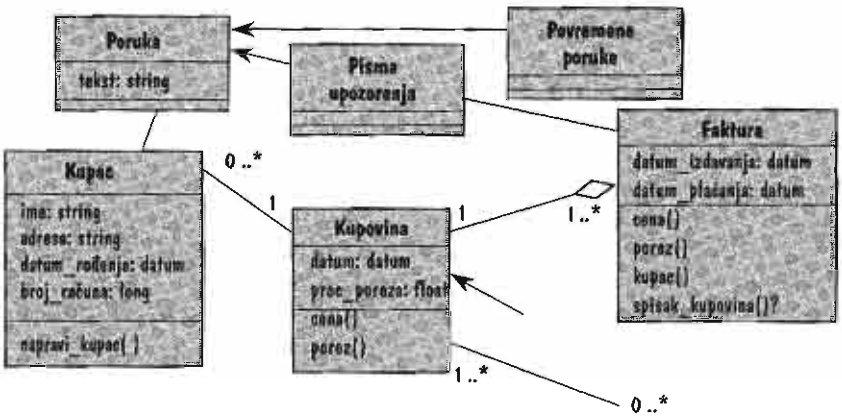
$$\text{vrednosni metodi po klasi} = \sum_{i=1}^{n} c_i$$

To jest, sumiramo skup složenosti metoda da bismo izračunali ukupne vrednosne metode za klasu. Ako je složenost svakog metoda 1, tada vrednosni metodi po klasi jednostavno predstavljaju broj metoda u tom proizvodu. Broj metoda i njihova složenost nagoveštava ukupno vreme i napor koji su potrebni za izgradnju i održavanje klase; što je veći broj metoda biće potrebno više rada, a veći će biti i uticaj na nasledníke te klase.

Tabela 6.6 Kolekcija mera Chidambera i Kemerera u različitim fazama razvoja

Metrika	Faza: Dizajn sistema	Dizajn programa	Kodiranje	Testiranje
Vrednosne metode po klasi	X	X		X
Dubina nasleđivanja	X	X		X
Broj naslednika	X	X		X
Snaga veza među objektima		X		X
Odzivi klase		X	X	
Nedostatak kohezije metoda		X	X	X

Oni takođe za klasu definišu **dubinu nasleđivanja** kao maksimalnu dužinu putanje u hijerarhiji od te klase do korena stabla nasleđivanja. Prema tome, što se klasa nalazi dublje u hijerarhiji, veća je verovatnoća da će nasleđiti više metoda. Zbog toga se takva klasa teže razume i održava. Slično tome, **broj naslednika** predstavlja broj direktnih potklasa koje su podređene datoj klasi. Ova mera je još jedan indikator jednostavnosti održavanja, testiranja i višekratne upotrebljivosti.



Metrika	Faktura	Kupovina	Pismo upozorenja	Povremene poruke	Poruka	Kupac
Vrednosne metode po klasi	4	2	0	0	0	2
Broj dece	0	0	0	0	2	0
Dubina nasleđivanja	0	0	1	1	0	0
Reagovanje na klasu	-	-	-	-	-	-
Vezivanje među objektima	2	4	2	1	1	2
Nedostatak kohezije metoda	-	-	-	-	-	-

SLIKA 6.33 Metrika Chidamber-Kemerer primenjena na dizajn sistema servisne stanice Royal

Ove koncepte možemo primeniti na model servisne stanice Royal. Na slici 6.33 vidimo deo modela koji se odnosi na fakture, kupovine i poruke. Za klasu *faktura* mera vrednosnih metoda je 4, ali je dubina nasleđivanja i broj naslednika 0.

Spregnutost među objektima je 2, pošto je klasa *faktura* u sprezi sa klasama *kupovina* i *pisma upozorenja*. Međutim, za klasu *kupovina* je vrednost spregnutosti objekata 4 na šta nam ukazuju četiri strelice koje polaze od nje ili se u njoj završavaju. Kao što smo napomenuli u poglavlju 5, preterana spregnutost nije poželjna; uvek težimo da svaka klasa bude što nezavisnija da bi se lakše razvijala, održavala, testirala i više puta koristila.

Izračunavanje nedostatka kohezije u metodama je složenije. Uzmite datu klasu *C* koja ima *n* metoda, od *M1* do *Mn*. Pretpostavimo da je {*Ij*} skup primeraka promenljivih koje metod M_i koristi. Znači da imamo *n* takvih skupova za svaki od *n* metoda. Možemo da definišemo *P* kao kolekciju parova (*Ir*, *Is*) gde *Ir* i *Is* nemaju nijedan zajednički član, a *Q* kao kolekciju parova (*Ir*, *Is*) gde *Ir* i *Is* imaju bar jedan zajednički član. U formalnoj notaciji definišemo da je *P*

$$P = \{(I_r, I_s) \mid I_r \cap I_s = \varnothing\}$$

a da je *Q*

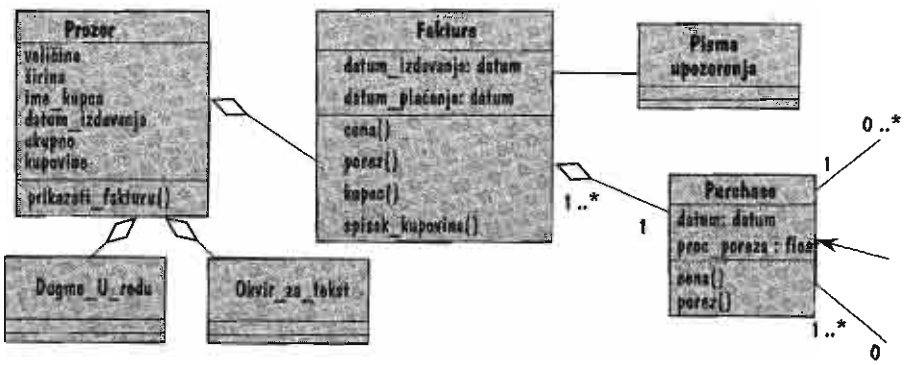
$$Q = \{(I_r, I_s) \mid I_r \cap I_s \neq \varnothing\}$$

Nedostatak kohezije u metodima za *C* definišemo kao $|P| - |Q|$ ako je $|P|$ veće od $|Q|$. Inače je nedostatak kohezije u metodima jednak nuli. Jednostavnije rečeno, nedostatak kohezije u metodima iznosi broj praznih preseka umanjen za broj nepraznih preseka. Mera se zasniva na činjenici da su dva metoda slična ukoliko imaju jedan zajednički primerak promenljive. Što je veći broj sličnih metoda; klasa je kohezivnija. Prema tome, mera LCOM pokazuje stepen relativne različitosti metoda neke klase.

Reaktivnost klase je skup metoda koje bi mogle da se izvrše kao reakcija na poruku koju neki objekt te klase primi. Ako je moguće pozvati veliki broj metoda po prijemu jedne poruke, daleko je teže izvesti testiranje i otklanjanje grešaka. Zato bi projektanti trebalo da se trude da reaktivnost klase bude mala, jer visoke vrednosti reaktivnosti ukazuju na potrebu dodatnog ispitivanja klase i njene implementacije.

Chidamber-Kemererove mere možemo primeniti i na dizajn programa. Na slici 6.34 prikazan je deo dizajna programa za servisnu stanicu Royal. Iz tabele na slici možemo videti da se spregnutost između objekata za klasu *faktura* povećala sa 2 na 3, ali da su ostale mere za klase *faktura*, *kupovina* i *pisma upozorenja* ostale iste. Promene u merama možemo koristiti kao indikatore za identifikaciju mogućih problema u kodu. Na primer, svaka mera nagoveštava neku smernicu u pogledu verovatnoće pojave grešaka u kodu:

- što je veći vrednosni metod po klasi, veća je verovatnoća greške u kodu (zato što je sistem složeniji, pa prema tome teži za testiranje);
- što je veći broj naslednika, veća je verovatnoća da će u kodu biti više grešaka;



Metrika	Faktura	Kupovina	Pisma upozorenja	Dugme U redu	Prozor	Okvir za tekst
WMC	4	2	0	0	1	0
NOC	0	0	0	0	0	0
DIT	0	0	1	0	1	0
RFC	-	-	-	-	-	-
CBO	3	4	2	1	3	1
LCOM	-	-	-	-	-	-

SLIKA 6.34 Chidamber-Kemererove mere primenjene na dizajn programa servisne stanice Royal

- što je veća dubina nasleđivanja, veća je verovatnoća da će u kodu biti više grešaka;
- što je veća reaktivnost klase, veća je verovatnoća da će u kodu biti više grešaka.

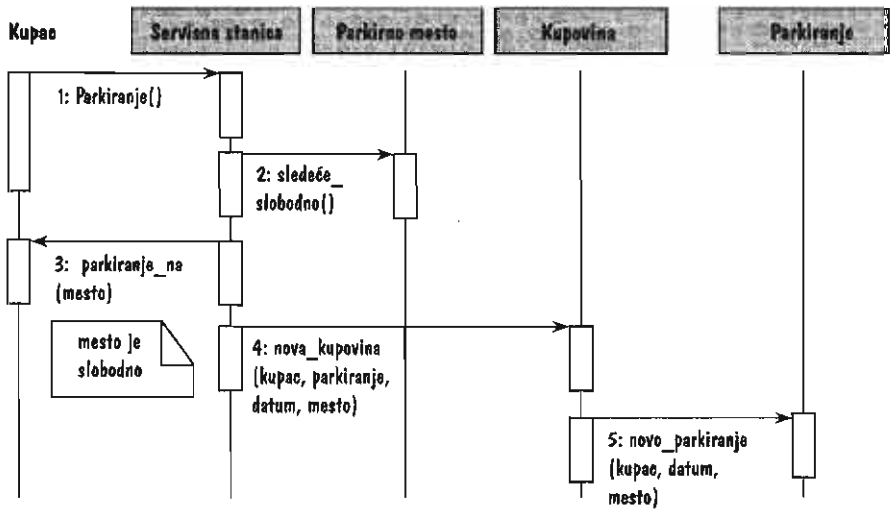
Ove smernice su izvedene iz empirijske procene C++ koda koje su obavili Basily, Briand i Melo (1995). Li i Henry (1993) takođe su razmatrali ove mere i prethodnom skupu koji su definisali Chedamber i Kemerer dodali još dve svoje:

- **spregnutost predavanjem poruka**: broj pozivanja metoda definisanih u klasi;
- **spregnutost apstrakcijom podataka**: broj apstraktnih tipova podataka koji se koriste u posmatranoj klasi, a definišani su u nekoj drugoj klasi tog sistema.

Li i Henry su pokazali kako se ove mere mogu upotrebiti za predviđanje veličine promena u klasama prilikom održavanja.

Travassos (1999) predlaže još neke mere za OO sisteme. Broj poruka koje šalje operacija može se upotrebiti za izračunavanje prosečne veličine operacija. Slično tome, moglo bi biti korisno da se zna prosečan broj parametara po operaciji.

Metrika	Min	Max
Prosečna veličina operacije	5	5
Prosečan broj parametara po operaciji	0	4



SLIKA 6.35 Merenja na osnovu dijagrama sekvence

Svaka pojedinačna operacija može se proceniti pomoću niza mera složenosti, koje su jednako primenljive i na tradicionalne metode projektovanja. Na slici 6.35 prikazana je primena tih mera na dijagram sekvence za servisnu stanicu Royal. Pošto je prikazana samo jedna operacija, a ona ima pet poruka, maksimalna i minimalna veličina operacije su jednake, i iznose pet. Vidimo da četvrti korak zahteva četiri parametara, a da za prvu operaciju nije potreban nijedan parametar. Prema tome, prosečan broj parametara po operaciji kreće se od nula do četiri.

Mere možemo da izvedemo i iz opisa klase. Na primer, pogledajmo šablon za klasu *punjenje gorivom*:

Ime klase: ***punjenje gorivom***

Kategorija: usluge

Spoljni dokumenti:

Kontrola izvoza: javna

Kardinalnost: n

Hijerarhija:

Natklase: usluge

Pridruživanja:

<bez imena uloge>:pridruženo gorivo<nema imena>

Ime operacije: cena

Javni član klase: punjenje gorivom

Dokumentacija:

//Izračunava konačnu cenu goriva

Preduslovi:

broj galona > 0

Dijagram objekta: (nije naveden)

Semantika:

konačna_cena = galoni * cena

Dijagram objekta: (nije naveden)

Konkurentnost: sekvencijalna

Javni interfejs:

Operacije:

cena

Privatni interfejs:

Atributi:

galoni

cena

Implementacija:

Atributi:

galoni

cena

Mašina stanja: ne

Konkurentnost: sekvencijalna

Trajnost: privremena

Iz ovog šablona možemo da utvrdimo veličinu klase po Lorenzu i Kiddu, koja se izračunava kao ukupan broj operacija plus broj atributa (uključujući nasleđena svojstva). Ovde je veličina klase četiri. Procenat javnih i zaštićenih klasa je nula, a javni pristup članovima podataka je takođe nula. Klasu takođe možemo da procenimo i pomoću tradicionalnijih mera kao što su **ulazni faktor** (broj klasa koje referenciraju posmatranu klasu) i **izlazni faktor** (broj klasa koje posmatrana klasa referencira). Takve mere mogu da nam pomognu za postavljanje osnovice i za praćenje, pa možemo da posmatramo rast sistema i procenjujemo predložene promene.

Gde treba vršiti OO merenja

Za merenje objektno orijentisanih sistema ima više načina, a još nije pronađen „najbolji" skup mera. Važno je imati na umu da su merenja korisna samo ako mogu da se upotrebe za bolje razumevanje, predviđanje ili kontrolu. Mere koje smo ovde predstavili ili neke druge iz literature ili neke koje sami smislite za svoje projekte mogu da budu korisne zavisno od toga koliko se lako prikupljaju i koliko su relevantne za problem koji pokušavate da rešite. Ispitaćemo mere predstavljene u ovom poglavlju da bismo videli koje se najbolje prikupljaju iz različitih OO dokumenata koje smo opisali.

Tabela 6.7 Gde treba vršiti OO merenja

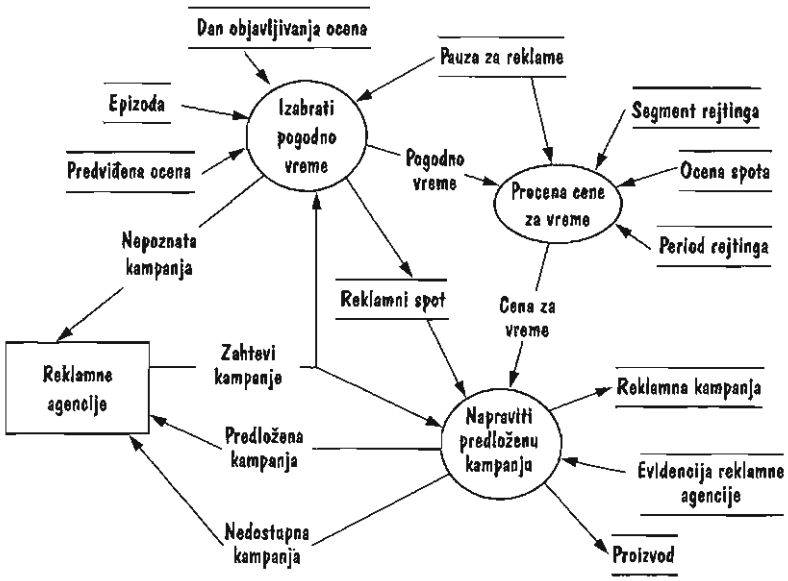
Metrika / Faza:	Slučajevi korišćenja	Dijagrami klasa	Dijagrami interakcije	Opisi klasa	Dijagrami stanja	Dijagrami paketa
Broj skriptova u scenariju	X					
Broj osnovnih klasa		X				
Broj klasa za podršku		X				
Prosečan broj klasa za podršku po osnovnoj klasi		X				
Broj podsistema						X
Veličina klase		X		X		
Broj operacija koje potklasa redefiniše		X				
Broj operacija koje potklasa dodaje		X				
Indeks specijalizacije		X				
Vrednosni metodi po klasi		X				
Dubina nasleđivanja		X				
Broj naslednika		X				
Spregnutost među objektima				X		
Reaktivnost na klase				X		
Nedostatak kohezije metoda			X			
Prosečna veličina operacije			X			
Prosečan broj parametara po operaciji				X		
Procenat javnih i zaštićenih elemenata				X		
Javni pristup podacima				X		
Broj osnovnih klasa		X				
Ulazni faktor/izlazni faktor		X				

Kao što vidite u tabeli 6.7, postoje mere koje se odnose na svih šest tipova dokumenata: slučajeve korišćenja, dijagrame klasa, dijagrame interakcije, opise klasa, dijagrame stanja i dijagrame paketa. Merenja podržavaju razvoj samim tim što problemi postaju vidljiviji. U primerima za servisnu stanicu Royal u celom ovom poglavlju videli smo kako svaka vrsta dijagrama otvara neko pitanje ili perspektivu koja nam omogućava bolje razumevanje problema ili rešenja. Na isti način jedna mera može da pomogne, ali kad koristimo više njih, dobijamo više uglova posmatranja, pa lakše možemo predvideti i rešavati sve izazove u fazi projektovanja i održavanja softvera.

6.8 PRIMER IZ INFORMACIONIH SISTEMA

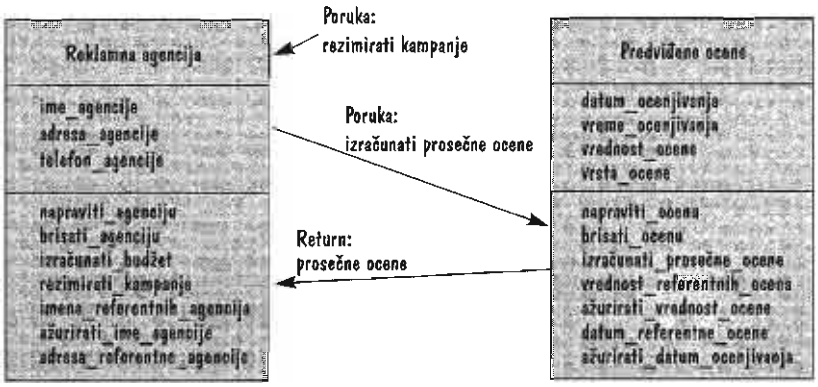
U poglavlju 4 smo videli da je za sistem Piccadilly predviđen i razvoj reklamnih kampanja. U tom poglavlju, za modelovane procesa koristili smo model događaj-odgovor, jedan rečnik podataka i jedan dijagram toka podataka za opisivanje postupka reklamne kam-

panje i definisanje zahteva za obradu kampanje. Dijagram toka podataka iz poglavlja 4, koji ovde prikazujemo kao sliku 6.36, poslužiće kao osnova za formulisanje slučajeva korišćenja u reklamnim kampanjama.



SLIKA 6.36 Dijagram toka podataka kao osnova za formulisanje slučajeva korišćenja u sistemu Piccadilly

Međutim, sa aspekta objektne orijentisanosti ovaj dijagram nije dovoljan da bi se napravili slučajevi korišćenja. Na primer, za reklamnu kampanju potrebne su predviđene ocene za konkretnu televizijsku emisiju ili skup emisija. Predviđena ocena uključena je u dijagram toka podataka; ona se dobija iz skladišta podataka. Ali, u stvari, sistem ta predviđanja mora da napravi na osnovu prethodnih evidencija. Na primer, predviđanje ocena omogućava agenciji da oceni prednosti i mane reklamiranja u raznim vremenima; reklama u jedanaest sati pre podne biće jeftinija od reklame u sedam sati uveče, ali će imati niži rejting i prema tome stići do manje ljudi. Treba da projektujemo klase koje će se baviti predviđanjem ocena i saopštavati ta predviđanja sistemu reklamne kampanje. Na slici 6.37 prikazan je prvi predlog za dve klase za predviđanje ocena. Klasa *reklamna agencija* prima poruku sa zahtevom da rezimira nekoliko predviđenih kampanja. Klasa zatim poziva klasu *predviđene ocene* i zahteva izračunavanje prosečnih ocena, koje se zatim vraćaju klasi *reklamna agencija*. U vežbi 6.4 analizira se ovaj dijagram i ove klase da bi se oni unapredili. Ali, čak i u ovom prvom pokušaju da se ova aktivnost definiše u smislu objekata, bolje se uočavaju problemi nego u dijagramu toka podataka i dijagramu raktivnosti na događaje iz poglavlja 4. Objektno orijentisani pristup nas primorava da detaljnije razmotrimo proces i da utvrdimo kako informacije teku kroz sistem u određeno vreme. Kada koristimo OO pristup, udaljavamo se od sekvencijalne prirode mnogih procesa, i umesto njih, za pokretanje niza aktivnosti koristimo zavisnost među podacima.



SLIKA 6.37 Dijagrami klasa za reklamnu kampanju

6.9 PRIMER IZ PRAKSE

Pristalice objektno orijentisanog razvoja često hvale potencijal višekratnog korišćenja objekata. Kao što smo videli, u sistemu Ariane-5 ponovo je upotrebljen SRI softver iz Ariane-4. Da je Ariana-5 implementirana objektno orijentisanim pristupom, ponovno korišćenje primenilo bi se kao kompozicija ili putem nasleđivanja. Da je upotrebljen kao kompozicija, SRI bi se posmatrao kao crna kutija, i pozivao bi se iz glavnog sistema i vraćao odgovarajuće vrednosti koje bi se koristile u drugim delovima sistema Ariane-5. Da je upotrebljen putem nasleđivanja, SRI struktura i ponašanja bili bi otvoreni za pristup i nasledili bi što je moguće više strukture i ponašanja iz roditeljskih klasa. Pristup sa kompozicijom verovatno ne bi sprečio nesreću do koje je došlo, pošto se problem prekoračenja ne bi video iz crne kutije. S druge strane, ponovno korišćenje putem nasleđivanja moglo je projektantima Ariane-5 da otkrije ranjivost dizajna SRI-ja, pa bi oni mogli da preduzmu preventivne mere.

6.10 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju je detaljno objašnjen proces objektno orijentisanog razvoja. Videli smo kako se slučajevi korišćenja formiraju tokom analize zahteva, prikazujući slike funkcionalnosti sistema sa različitih aspekata. Slučajevi korišćenja se zatim doteruju da bi se formirao model sistema sa visokim nivoom apstrakcije, na kome se vidi način na koji sistem rešava postojeći problem. Dizajn programa implementira nefunkcionalne zahteve, kao i ostale detalje potrebne da bi programeri izvršili implementaciju.

Prikazali smo osnovnu notaciju i pojmove iz UML-a, koji postaje de facto standard za opisivanje OO sistema. Istovremeno smo uveli nekoliko mera koje se obično koriste za merenje veličine i karakteristika dizajna OO proizvoda. Te mere nam ne pomažu samo za procenu alokacije resursa potrebnih za razvoj, testiranje i održavanje; one takođe služe kao upozorenje da sistem postaje složen i težak za razumevanje.

6.11 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

OO pristup pruža vašem timu zajednički jezik za komunikaciju o sistemu od njegovog nastanka, pa do isporuke. Zahtevi mogu da se izraze u formi objekata i metoda, što važi za dizajn i kôd. Ta konsistentnost terminologije omogućava da se predvidi veličina i struktura sistema mnogo pre završetka projektovanja. Na primer, ona omogućava lako praćenje proizvoda; vidite kako objekat u zahtevima postaje objekat u dizajnu, a zatim objekat u kodu.

UML podstiče konsistentnost notacije, pitanje koje je šire od konkretnog sistema koji gradite. Pogotovo ako gradite sistem koristeći tuđe OO komponente, konsistentna notacija olakšava vašem timu da shvati implikacije korišćenja konkretnog objekta ili klase. Štaviše, konsistentnost pomaže timovima koji rade na održavanju i testiranju da lakše naprave scenarije testova i lakše prate promene. Pošto se zahtevi, dizajn i kôd izražavaju na isti način, lakše možete da procenite efekte predložene izmene zahteva ili dizajna.

6.12 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Objektno orijentisani pristup bio je plodno polje za istraživanje u softverskom inženjerstvu. Tradicionalni načini za procenu sistema nisu adekvatni za objektni pristup. Predviđanja koja se zasnivaju na redovima koda zamenjuju se metodama predviđanja u kojima se koriste karakteristike klasa i hijerarhija. Konkretno, istraživanja predvidivosti grešaka proizvela su nove objektno orijentisane načine za procenu broja problema preostalih u dizajnu ili kodu.

Drugi istraživači procenjuju na koji bi se način efektno iskoristile najbolje karakteristike objektnog pristupa. Koliko treba da budu duboke hijerarhije? Kada je za višekratno korišćenje bolje komponovanje, a kada nasleđivanje? Kako je najbolje iskoristiti nasleđivanje i polimorfizam? Uz ova područja istraživanja vezan je neprekidan napor da se pronađu dobri načini za merenje OO procesa i njegovih proizvoda.

6.13 SEMINARSKI RAD

Pregledajte ponovo zahteve za sistem *Aranžer pozajmica*. Na koji način ih treba promeniti da bi se implementirali kao objektno orijentisani zahtevi? Na koji način bi te promene olakšale kasnije unapređenje i menjanje sistema? Nakon toga ponovo pregledajte sedam karakteristika objektno orijentisanih sistema. Da li bi neka od tih karakteristika bila od koristi za *Aranžer pozajmica*? Na kraju, razvijte objektno orijentisani dizajn za *Aranžer pozajmica* i uporedite ga sa prvobitnim dizajnom. Koji je od njih lakše testirati? A menjati?

6.14 KLJUČNE REFERENCE

Postoji nekoliko dobrih knjiga sa detaljnim uputstvima za razmišljanje o objektno orijentisanim sistemima. Gamma i dr. (1995) daju opšti pregled o tome kako razmišljati o sistemima u smislu šablona dizajna. Knjige kao što su Jacobson i dr. (1995), Rumbaugh i dr. (1991) i Booch (1994) nude detaljne informacije o tome kako da se odrede, projektuju i kodiraju OO sistemi. Binder (1999) opisuje sve što ste ikad poželeli da znate o

testiranju OO sistema; knjiga je puna dobrih primera i protivprimera. Larman (1998) i Fowler i Scott (1999) su dobri početni materijali za primenu UML-a.

Klasične reference za OO merenja su Lorenz i Kidd (1994), Li i Henry (1993) i Chidamber i Kemerer (1994). Travassos i Andrade (1999) povezuju ciljeve OO merenja sa nekim razumnim smernicama za OO razvoj. Pitanja vezana za zahteve koje smo predstavili u ovom poglavlju izvedena su iz čitanja zasnovanog na perspektivi koje su razvili Shull i dr. (2000).

6.15 VEŽBE

1. Mani, rukovodilac servisne stanice Royal, proširiće usluge perionicom automobila. To treba da bude automatizovan sistem. Kupac bira vrstu pranja i saopštava tip automobila. Sistem izračunava tarifu i prikazuje iznos dugovanja na kontrolnom panelu. Kupac zatim može da plati pranje automobila. Nakon plaćanja, ako je automat trenutno zauzet, sistem javlja da kupac mora da čeka. Inače, sistem javlja da kupac treba da doveze kola na rampu za pranje. Na osnovu opisa servisne stanice Royal iz ovog poglavlja, napravite skup slučajeva korišćenja perionice.
2. Na slici 6.23 dat je dijagram stanja za klasu *inventar* servisne stanice Royal. Na slici 6.25 prikazan je dijagram aktivnosti za tu istu klasu. Objasnite šta svaki od tih dijagrama nudi projektantima, programerima i grupi za testiranje. Zašto su nam potrebne dve vrste dijagrama?
3. Šta znači indeks specijalizacije koji predlažu Lorenz i Kidd? Koje su implikacije visokog indeksa specijalizacije? A niskog? Koje su implikacije značajne promene ovog indeksa u toku razvoja proizvoda?
4. Napišite ponovo dijagram toka podataka sa slike 6.36 u formi slučajeva korišćenja. Kako se klase sa slike 6.37 uklapaju u vaše scenarije? Dijagram na slici 6.37 je spoj nekoliko vrsta dijagrama opisanih u ovom poglavlju. Poboljšajte klase sa slike 6.37 pomoću pravilne UML notacije i opišite kako se predviđanje ocena uklapa u sveukupan proces reklamne kampanje. Na koji način pravilni UML dijagrami unapređuju dizajn?
5. Da li se objektno orijentisani pristup može koristiti za razvoj bilo kog sistema? Koje su prednosti objektne orijentacije? A mane? Navedite primer sistema za koji OO ne bi bila odgovarajuća strategija razvoja.

7 Pisanje programa



U ovom poglavlju obradivaćemo:
- standarde programiranja,
- smernice za višekratno korišćenje,
- dizajn kao okvir koda,
- unutrašnju i spoljnu dokumentaciju.

Veštine koje smo da sada savladali pomogle su nam da shvatimo problem kupaca i korisnika i da formulišemo njegovo opšte rešenje. Sada se moramo usredsrediti na implementiranje tog rešenja u obliku softvera, tj. moramo napisati programe koji implementiraju dizajn. Taj zadatak može izazvati zebnju iz nekoliko razloga. Prvo, projektanti možda nisu uzeli u obzir sve specifičnosti platforme i programskog okruženja; strukture i relacije koje je lako opisati grafikonima i tabelama nije uvek jednostavno preslikati u kôd . Drugo, kôd moramo pisati tako da bude razumljiv ne samo nama kada mu se vratimo radi testiranja, već takođe i drugima u toku evolucije sistema. Treće, moramo iskoristiti prednosti koje poseduju organizacija projekta, struktura podataka i programski sklopovi, a pri tome kreirati kôd koji se može jednostavno ponovno koristiti.

Jasno je da postoji više načina da se implementira isti dizajn, da postoje mnogi jezici i alati, i da u ovoj knjizi ne možemo sve da ih obradimo. U ovom poglavlju koristimo primere koji su zasnovani na nekoliko popularnih programskih jezika, ali su smernice u opštem slučaju primenljive u okviru proizvoljne implementacije. Namena ovog poglavlja nije da vas nauči da programirate, već da prikaže praktične elemente iz domena softverskog inženjerstva koje bi trebalo da imate na umu dok pišete kôd.

7.1 STANDARDI I PROCEDURE PROGRAMIRANJA

U toku svoje karijere verovatno ćete raditi na više različitih softverskih projekata i pisati kôd za različita područja primene, oslanjajući se na različite alate i tehnike. Jedan deo aktivnosti uključivaće i analizu postojećeg koda, bilo da ga treba u celosti zameniti ili

samo promeniti, ili ponovo upotrebiti u sklopu druge aplikacije. Takođe ćete učestvovati u formalnim i neformalnim pregledima kôda koji ste pisali vi ili neko drugi. Većina tih aktivnosti razlikuje se od programiranja koje ste upražnjavali na časovima. Na času radite samostalno da bi predavač procenio kvalitet koda i predložio poboljšanja. Međutim, van njih većina softvera se razvija timski, pri čemu je za izradu kvalitetnog proizvoda potrebno uključiti više različitih profesija. Čak i u samom pisanju koda obično učestvuje više ljudi i potreban je visok stepen saradnje i koordinacije. Zbog toga je veoma važno da i drugi mogu da shvate ne samo to što ste napisali, već i zašto ste to napisali i kako se to uklapa u njihov deo posla.

Zbog toga se morate upoznati sa standardima i procedurama koje važe u vašoj organizaciji pre nego što počnete da pišete kôd. Mnoge kompanije zahtevaju da kôd bude usklađen sa standardima vezanim za stil, format i sadržaj, da bi kôd i prateća dokumentacija bili jasni svakome ko ih čita. Na primer, u podsetniku 7.1 opisan je Microsoftov pristup standardima programiranja.

Standardi za vas

Standardi i procedure mogu da vam pomognu da sredite misli i izbegnete greške. Neke procedure odnose se na način dokumentovanja koda da bi on bio jasan i da bi mogao lako da se prati. Takva dokumentacija vam omogućava da prekinete rad i vratite mu se, a da pri tom ne izgubite nit. Standardizovana dokumentacija takođe pomaže pri pronalaženju grešaka i unošenju izmena, jer pojašnjava koji delovi programa izvršavaju koje funkcije.

PODSETNIK 7.1 STANDARDI PROGRAMIRANJA U MICROSOFTU

Cusumano i Selby (1995, 1997) su proučavali razvoj softvera u Microsoftu. Oni ističu da Microsoft pokušava da ugradi neke aspekte iz prakse softverskog inženjerstva u razvojni ciklus softvera, a da istovremeno očuva kreativnost i individualnost koja je obično svojstvena hakerima. Zato Microsoft mora da nađe načine „za strukturiranje i koordiniranje rada pojedinaca, a uz dovoljnu fleksibilnost da očuva njihovu kreativnost pri faznom razvoju detalja jednog proizvoda". Zbog pritisaka tržišta i promenljivih potreba, Microsoftovi timovi naizmenično rade na dizajniranju komponenti, njihovoj izgradnji i testiranju. Na primer, sa sve boljim poznavanjem funkcija određenog proizvoda, članovi tima revidiraju i tipove svojstava i detalje u vezi s njima.

Međutim, fleksibilnost ne sprečava primenu standarda. Skoro svi Microsoftovi timovi rade na jednoj fizičkoj lokaciji, koriste iste jezike za razvoj (obično C ili C++), iste stilove kodiranja i standardne razvojne alate. Ti standardi olakšavaju komunikaciju unutar tima, razmatranje prednosti i mana različitih varijanti dizajna, kao i rešavanje problema. Microsoft takođe zahteva da timovi redovno prikupljaju određene pokazatelje, uključujući informacije o pojavi otkaza i vremenu potrebnom za pronalaženje i ispravljanje grešaka koje ih prouzrokuju. Te mere se koriste prilikom donošenja odluke o tome kada produžiti razvoj, a kada samo popraviti proizvod.

Standardi i procedure takođe pomažu pri prevođenju dizajna u kôd. Strukturisanjem koda prema standardima, održavate usklađenost elemenata na nivou dizajna sa elementima na nivou koda. To za posledicu ima lakše implementiranje promena u kodu koje su posledica promena u dizajnu. Slično tome, izmene koda koje su posledica promena u specifikaciji hardvera ili interfejsa, jednostavno se sprovode i time se značajno smanjuje mogućnost pojave grešaka.

Standardi za druge

Kada dovršite svoj kôd, postoji verovatnoća da će ga drugi koristiti na različite načine. Na primer, kao što ćemo videti u kasnijim poglavljima, možda će drugi tim vršiti testiranja. Možda će druga grupa ljudi integrisati vaš softver sa drugim programima da bi izgradili i testirali podsisteme, a kasnije i ceo sistem. Čak i kad je sistem podignut i kada radi, može se javiti potreba za promenama, bilo zbog neke greške ili zato što kupac želi da promeni način na koji sistem obavlja funkcije. Vi možda nećete biti deo timova za održavanje ili testiranje i zato je bitno da organizujete, formatirate i dokumentujete svoj kôd tako da drugi mogu lako da razumeju šta on radi i kako to realizuje.

Na primer, pretpostavimo da svaki program proizveden u vašoj kompaniji počinje jednim odeljkom u kojem se opisuju funkcije programa i njegovi interfejsi sa ostalim programima. Početni odeljak mogao bi da izgleda ovako:

```
************************************************
*
* KOMPONENTA ZA PRONALAŽENJE PRESEKA DVE PRAVE
*
* IME KOMPONENTE: FINDPT
* PROGRAMER: E. ELLIS
* VERZIJA: 1.0 (2 FEBRUAR 2001)
*
* POZIVANJE PROCEDURE:
*  CALL FINDPT (A1, B1, C1, A2, B2, C2, XS, YS, INDIKATOR)
* ULAZNI PARAMETRI:
*  ULAZNE PRAVE DAJU SE U OBLIKU
*   A1*X + B1*Y + C1 = 0 I
*   A2*X + B2*Y + C2 = 0
*  PA SU ULAZI KOEFICIJENTI A1, B1, C1 I A2, B2, C2
* IZLAZNI PARAMETRI:
*  AKO SU PRAVE PARALELNE, INDIKATOR SE POSTAVLJA NA 1.
*  INAČE INDIKATOR = 0 I TAČKA PRESEKA JE (XS, YS)
*
************************************************
```

Ovaj blok komentara informiše čitaoca šta radi kôd koji sledi i daje pregled načina rešavanja. Nekome ko traži komponentu za ponovno korišćenje ovaj blok pruža dovoljno informacija za donošenje odluke da li je to ono što traži. Nekome ko traži uzrok otkaza ovaj blok daje dovoljno elemenata za procenu verovatnoće da je razmatrana komponenta direktni krivac ili možda saučesnik.

Čitanjem ovakvih blokova, programer koji radi na održavanju lakše će pronaći komponentu koja treba da se promeni. Kada se pronađe komponenta, ako su imena podataka jasna, a interfejsi dobro definisani, programer koji radi na održavanju može da bude siguran da zahtevana promena neće proizvesti neočekivane posledice po druge delove koda.

Postoje automatizovani alati za analizu kôda i utvrđivanje koje procedure komponenta poziva, a koje procedure nju pozivaju. Dokumentacija koju ti alati proizvedu ukazuje na komponente koje mogu pozvati razmatranu proceduru i komponente koje ona poziva. Uz takve informacije relativno je jednostavno vršiti promene u sistemu. Na kraju ovog poglavlja razmatramo primer standarda i procedura da bismo uočili načine na koje oni mogu da usmeravaju postupak programiranja.

Uparivanje dizajna sa implementacijom

Najkritičniji standard se tiče direktne usklađenosti komponenti iz dizajna programa sa komponentama iz programskog koda. Celokupan proces dizajna ništa ne vredi ako se modularnost prisutna u dizajnu ne prenese i na kôd. Karakteristike dizajna, kao što su slaba spregnutost, visoka kohezija i dobro definisani interfejsi trebalo bi da budu i karakteristike programa, da bi algoritmi, funkcije, interfejsi i strukture podataka mogli lako da se prate iz dizajna u kôd i obratno.

Imajte na umu da će osnovna namena sistema ostati ista tokom celokupnog životnog ciklusa softvera, ali da se vremenom njegova priroda može menjati kako kupci budu identifikovali unapređenja i izmene. Na primer, pretpostavimo da ste u timu koji projektuje računarski upravljane displeje za automobile. Sistem koji gradite verovatno će se uvek ugrađivati u automobile, ali meniji i uređaji za unos mogu da se promene, ili da se dodaju nove mogućnosti. Te promene se prvo vrše u sklopu dizajna visokog nivoa, a zatim prosleđuju, kroz niz detalja na nižem nivou apstrakcije, na kôd koji treba promeniti. Zbog toga je usklađenost između dizajna i koda važna. U kasnijim poglavljima videćemo da testiranje, održavanje i upravljanje konfiguracijom nisu mogući bez veza koje se ovim standardima ustanovljavaju.

7.2 SMERNICE ZA PROGRAMIRANJE

Programiranje zahteva mnogo kreativnosti. Prisetite se da dizajn u stvari predstavlja opis funkcije ili svrhe svake komponente, ali da programer ima veliku slobodu pri implementiranju dizajna u kodu. Dizajn ili specifikacija zahteva može da, bilo neposredno, kao zahtev projektanta ili kupaca, bilo posredno, zbog namere da se koriste gotovi delovi koda, sugeriše korišćenje određenog programskog jezika. Smernice specifične za određeni programski jezik ovde nisu predmet razmatranja, zato što postoje mnoge knjige koje tu temu obrađuju. Umesto toga, opisaćemo nekoliko smernica koje važe za programiranje uopšte, nezavisno od programskog jezika.

Bez obzira na programski jezik koji se koristi, svaka programska komponenta uključuje najmanje tri osnovna aspekta: kontrolne strukture, algoritme i strukture podataka. Razmotrićemo detaljnije svaku od njih.

Kontrolne strukture

Mnoge kontrolne strukture komponente predložene su na nivou arhitekture i dizajna, i potrebno ih je očuvati prilikom transformacije dizajna u kôd. Kod nekih arhitektura, kao što su implicitno pozivanje i objektno orijentisani dizajn, kontrola se zasniva na stanjima sistema i na promenama vrednosti promenljivih. U drugima, koje su više proceduralne, kontrola zavisi od strukture samog koda. Za svaku vrstu dizajna važno je da struktura programa odražava kontrolne strukture definisane u sklopu dizajna. Čitaoca ne bi trebalo primoravati da pravi velike skokove kroz kôd, obeležava delove na koje treba da se vrati i pita se da li je išao pravom putanjom. On bi trebalo da se usredsredi na to šta program radi, a ne na tok kontrole. Zato mnoga uputstva i standardi zahtevaju da se kôd piše tako da komponenta može lako da se pročita, odozgo nadole.

Pogledajmo jedan primer kako promena strukture može da poveća razumljivost. Analizirajmo sledeći program. Prenos kontrole u sklopu programa je skokovit i zato se teško prati.

```
        benefit = minimum;
        if (age < 75) goto A;
        benefit = maximum;
        goto C;
        if (age < 65) goto B;
        if (age < 55) goto C;
A:      if (age < 65) goto B;
        benefit = benefit * 1.5 + bonus;
        goto C;
B:      if (age < 55) goto C;
        benefit = benefit * 1.5;
C:      next statement
```

Isti rezultat možemo postići oslanjajući se na strukturu koja se lakše prati. Preuredićemo kôd na sledeći način:

```
if (age < 55) benefit = minimum;
elseif (age < 65) benefit = minimum + bonus;
elseif (age < 75) benefit = minimum * 1.5 + bonus;
else benefit = maximum;
```

Naravno, nije uvek moguće, niti praktično, da tok ide strogo odozgo nadole. Na primer, postizanjem uslova završetka petlje može se narušiti tok. Međutim, korisno je da, gde god je to moguće, zahtevana akcija bude neposredno nakon odluke koja je uslovljava.

U prethodnim poglavljima smo videli da je modularnost karakteristika dobrog dizajna. To isto važi i za kôd. Ako gradimo program od modularnih blokova, možemo na raznim nivoima da skrivamo implementacione detalje, čime sistem postaje razumljiviji i lakši za testiranje i održavanje. Programsku komponentu možemo modularno posmatrati, a zatim koristiti makroe, procedure, potprograme, metode i nasleđivanje kao sredstva za skrivanje detalja uz povećanje stepena razumljivosti. Štaviše, što je komponenta koda modularnija, lakše će se održavati i ponovo upotrebiti u drugim aplikacijama; izmene se ograničavaju na konkretan makro, potprogram ili drugi sastavni deo.

Zbog toga, kada pišete kôd, imajte na umu da je opštost vrlina; nemojte dopustiti da kôd bude specijalizovan više nego što je to neophodno. Na primer, komponenta koja u nizu od osamdeset znakova traži tačku, može da se napiše sa ulaznim parametrima: dužina stringa i znak koji se traži. Tako će komponenta moći da se upotrebi za pretraživanje niza znakova proizvoljne dužine na bilo koji znak. S druge strane, nemojte dopustiti da opštost komponente utiče na performansu ili mogućnost razumevanja.

Spregnutost i kohezija su još dve karakteristike dizajna koje se prenose u komponente koda. Kada pišete programe, vodite računa da upotrebite imena parametara i komentare koji će iskazati snagu veza među komponentama. Na primer, pretpostavite da pišete komponentu za procenu poreza na dohodak. Ona koristi iznos bruto prihoda i olakšice koje dobija od drugih komponenti. Umesto da svoj kôd opišete kao

```
Ponovo izračunati POREZ
```

bolje je da napišete

```
Ponovo izračunati POREZ na osnovu vrednosti BRUTO_PRIHOD i
  OLAKŠICE
```

Ovaj drugi komentar objašnjava kako se izračunavanje povezuje sa stavkama podataka iz drugih komponenti.

Vaš kôd mora da omogući čitaocu da prepozna parametre koji se prenose u vašu komponentu i iz nje (ako takvi postoje). U protivnom će testiranje i održavanje biti izuzetno teški. Drugim rečima, zavisnost među komponentama mora biti vidljiva. U isto vreme, kao što su komponente sistema projektovane da međusobno kriju informacije, potkomponente vašeg programa bi trebalo jedna od druge da kriju detalje u vezi sa izračunavanjima. Na primer, u prethodnom programu za pretraživanje niza znakova, komponenta koja pretražuje tekst mora da poseduje svest o načinu traženja navedenog znaka. Ali, komponenta koja je poziva ne mora da zna kako se znak pronalazi, već samo da je on pronađen i na kojem mestu se nalazi. To sakrivanje informacija omogućava promenu algoritma pretraživanja bez uticaja na ostatak koda.

Algoritmi

Dizajn programa često definiše klasu algoritama koju treba upotrebiti prilikom kodiranja komponente. Na primer, dizajn može da nalaže upotrebu algoritma Quicksort, ili može da navodi logičke korake tog algoritma. Međutim, postoji visok stepen fleksibilnosti u postupku konvertovanja algoritma u kôd, zavisno od ograničenja implementacionog programskog jezika i hardvera.

Jedna od oblasti u kojoj imate veliko diskreciono pravo jeste performansa ili efikasnost implementacije. Instinkt vas možda navodi da pravite kôd koji će se izvršavati najvećom mogućom brzinom. Međutim, ubrzavanje koda može da sadrži prikrivene troškove:

- trošak pisanja bržeg koda koji je možda složeniji i zato zahteva više vremena za pisanje;
- troškove vremena za testiranje koda čija složenost zahteva više primera za testiranje ili više test podataka;
- troškove vremena potrebnog da korisnici shvate kôd;
- troškove vremena za promenu koda ako se pojavi potreba.

Prema tome, vreme izvršavanja predstavlja samo mali deo u ukupnoj jednačini troškova. Pitanja vremena izvršavanja treba razmatrati zajedno sa kvalitetom dizajna, standardima i zahtevima kupca. Posebno je važno da ne žrtvujete jasnoću i ispravnost radi brzine.

Ako je brzina značajna za vašu implementaciju, morate da proučite na koji način prevodilac optimizuje kôd. U protivnom se može dogoditi da optimizacija uspori naizgled brži kôd. Da biste videli kako može doći do tog paradoksa, pretpostavite da pišete kôd za implementaciju trodimenzionalnog niza. U cilju povećanja efikasnosti, umesto trodimenzionalne, koristite jednodimenzionalnu matricu i sami izračunavate indeks. Na osnovu toga, u kodu se javljaju promenljive, kao na primer:

```
index = 3*i + 2*j + k;
```

za izračunavanje položaja elementa trodimenzionalnog niza. Međutim, možda prevodilac, da bi skratio vreme izvršavanja, za indeksiranje niza koristi registre. Ako se prevodilac oslanja na tehniku automatskog inkrementiranja registara a ne na izračunavanje svake pozicije množenjem i sabiranjem, korišćenje jednodimenzionalnog niza i izračunavanje indeksa produžavaju vreme izvršavanja!

Strukture podataka

Kada pišete programe, podatke bi trebalo da formatirate i čuvate na takav način da se njima jednostavno upravlja i manipuliše. Postoji nekoliko tehnika u kojima se organizacija programa prilagođava strukturi podataka.

Očuvanje jednostavnosti programa. Ponekad su u dizajnu programa definisane strukture podataka koje treba koristiti za implementaciju funkcija. Često se te strukture biraju zato što se uklapaju u sveukupnu šemu sakrivanja informacija i kontrole interfejsa komponenti. Manipulisanje podacima unutar komponente može da, na sličan način, utiče na izbor strukture podataka . Na primer, ponovno strukturisanje podataka može da uprosti izračunavanje u programu. Da bi to bilo jasnije, pretpostavimo da pišete program za izračunavanje visine saveznog poreza na dohodak. Na ulazu dobijate određeni iznos oporezivog dohotka i sledeća pravila:

1. Za prvih 10.000 dolara dohotka porez iznosi 10%.
2. Za sledećih 10.000 dolara dohotka iznad 10.000 dolara porez iznosi 12%.
3. Za sledećih 10.000 dolara dohotka iznad 20.000 dolara porez iznosi 15%.
4. Za sledećih 10.000 dolara dohotka iznad 30.000 dolara porez iznosi 18%.
5. Za svaki dohodak iznad 40.000 dolara porez iznosi 20%.

Znači da neko sa oporezivim dohotkom od 35.000 dolara plaća 10% od prvih 10.000 dolara (ili 1.000 dolara), 12% od sledećih 10.000 dolara (ili 1.200 dolara), 15% od sledećih 10.000 dolara (ili 1.500 dolara) i 18% od preostalih 5.000 dolara (ili 900 dolara) što ukupno iznosi 4.600 dolara. Za izračunavanje tog poreza možete u vašu komponentu da ubacite kôd koji učitava oporezivi dohodak i zatim primenjuje sledeći algoritam:

```
tax = 0.
if (taxable_income == 0) goto EXIT;
if (taxable_income > 10000) tax = tax + 1000;
```

```
else{
   tax = tax + .10*taxable_income;
   goto EXIT;
}
if (taxable_income > 20000) tax = tax + 1200;
else
   tax = tax + .12*(taxable_income-10000):
   goto EXIT;
if (taxable_income > 30000) tax = tax + 1500;
else
   tax = tax + .15*(taxable_income-20000);
   goto EXIT;
if (taxable_income > 40000)
   tax = tax + .18*(taxable_income-30000);
   goto EXIT;
else
   tax = tax + 1800. + .20*(taxable_income-40000);
EXIT: ;
```

Međutim, možemo da definišemo tabelu poreza za svaki interval oporezivog dohotka kao u tabeli 7.1, gde za svaki interval imamo osnovni iznos i procenat.

Pomoću te tabele možemo značajno da uprostimo algoritam:

```
for (int i-2; level=1; i <= 5; i++)
    if (taxable_income > bracket[i])
      level = level + 1;
tax = base[level]+percent[level]*(taxable_income-
bracket[level]);
```

Vidite kako je izračunavanje jednostavnije kad smo promenili način definisanja podataka. Zbog toga se program lakše shvata, testira i menja.

Tabela 7.1 Primer tabele za porez

Raspon	Osnova	Procenat
0	0	10
10.000	1.000	12
20.000	2.200	15
30.000	3.700	18
40.000	5.500	20

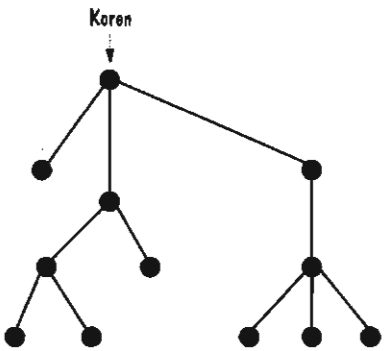
Korišćenje struktura podataka za utvrđivanje strukture programa. U primeru sa tabelom poreza, način definisanja podataka diktira način na koji se vrši izračunavanje. U opštem slučaju, strukture podataka mogu da utiču na organizaciju i tok programa. U

nekim slučajevima struktura podataka može da utiče i na izbor jezika. Na primer, LISP je projektovan za obradu listi i sadrži strukture koje ga čine primerenijim od drugih jezika kada treba obrađivati liste. Slično tome, Ada i Eiffel sadrže sklopove za obradu nedozvoljenih stanja, koja nazivamo izuzeci.

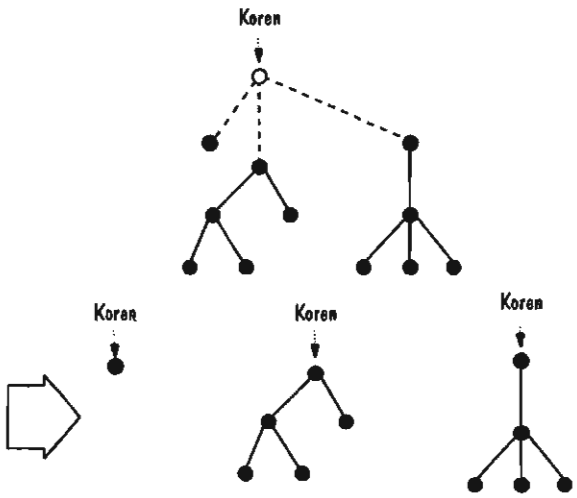
Za strukturu podataka se kaže da je **rekurzivna** ako se definiše pomoću jednog početnog elementa, a svi sledeći elementi se izvode kao funkcije prethodno definisanih. Na primer, **stablo sa korenom** je graf sastavljen od čvorova i grana i koji zadovoljava sledeće uslove:

1. Samo jedan čvor u stablu predstavlja koren.
2. Ako se izbrišu grane koje polaze iz korena, dobija se graf koji se sastoji od skupa grafova koji se ne seku i od kojih svaki predstavlja stablo sa korenom.

Na slici 7.1 prikazano je stablo sa korenom, a na slici 7.2 prikazano je kako uklanjanje korena proizvodi skup manjih stabala. Koren svakog manjeg stabla je čvor koji je prethodno bio povezan sa prvobitnim korenom većeg stabla. Prema tome, stablo sa korenom definiše se u smislu korena i podstabala: to je rekurzivna definicija.



SLIKA 7.1 Stablo sa korenom



SLIKA 7.2 Podstabla stabla sa korenom

Programski jezici kao što je Pascal dozvoljavaju da rekurzivne procedure obrađuju rekurzivne strukture podataka. Možda će vam za ovu vrstu podataka bolje odgovarati rekurzivne procedure pošto se njihovim korišćenjem teret upravljanja strukturom podataka prenosi sa programa na prevodioca. Tako će programiranje biti lakše i program će se lakše razumevati. Prema tome, u opštem slučaju bi trebalo pažljivo razmotriti strukture podataka pri odlučivanju na kom programskom jeziku će se implementirati dizajn.

Opšte smernice

Da bi se u vašem kodu očuvao kvalitet dizajna, korisno je da budete upoznati sa nekoliko opštih strategija.

Lokalizovanje ulaza i izlaza. Delovi programa koji učitavaju ulaz ili stvaraju izlaz veoma su specijalizovani i moraju da odražavaju karakteristike osnovnog hardvera i softvera. Zbog te zavisnosti, ponekad je teško testirati delove programa koji obavljaju ulazne i izlazne funkcije. U stvari, to su delovi koji će se najverovatnije menjati ako se promene hardver ili softver. Prema tome, poželjno je da u komponentama ti delovi budu odvojeni od ostatka koda.

Dodatna prednost lokalizacije je uopštavanje celog sistema. U specijalizovanu komponentu mogu da se uključe i druge opšte funkcije koje treba obaviti nad ulaznim podacima (kao što je formatiranje i provera tipa) tako da se ostale komponente rasterećuju i time eliminišu ponavljanja. Slično tome, stavljanjem izlaznih funkcija na jedno mesto dobija se sistem koji se lakše shvata i menja.

Uključivanje pseudokoda. Obično se u fazi dizajna postavlja okvir za svaku programsku komponentu. Zatim se, uz dodatak individualne kreativnost i znanja, piše kôd kojim se taj dizajn implementira. Na primer, dizajn ne mora biti strogo vezan za programski jezik i može da ostavi slobodu izbora jezičkih sklopova, načina njihove upotrebe, načina predstavljanja podataka itd. Pošto dizajn postavlja konture onoga što programska komponenta mora da uradi, korisno je da se od dizajna ka kodu ide u fazama, a ne da se dizajn odmah prevodi u kôd.

Za prilagođavanje dizajna izabranom programskom jeziku može se upotrebiti pseudokôd. Adaptiranjem sklopova i reprezentacija podataka ne ulazeći u specifičnosti svake komande, možete da eksperimentišete i posmatrate koja bi implementacija bila najpogodnija. Na taj način, kôd može da se preuređuje i da mu se menja struktura uz minimum dopisivanja. Na primer, pretpostavimo da ovako izgleda dizajn jedne komponente sistema za obradu teksta:

```
COMPONENT PARSE_LINE:
 Read next eighty characters.
     IF this is a continuation of the previous line,
        Call CONTINUE
    ELSE determine command type.
    ENDIF
 CASE of COMMAND_TYPE
    COMMAND_TYPE is paragraph: Call PARAGRAPH.
    COMMAND_TYPE is indent: Call INDENT.
    COMMAND_TYPE is skip line: Call SKIP_LINE.
```

```
    COMMAND_TYPE is margin: Call MARGIN.
    COMMAND_TYPE is new page: Call PAGE.
    COMMAND_TYPE is double space: Call DOUBLE_SPACE.
    COMMAND_TYPE is single space: Call SINGLE_SPACE.
    COMMAND_TYPE is break: Call BREAK.
    COMMAND_TYPE is anything else: Call ERROR.
  ENDCASE
```

Zavisno od izabranog jezika i vaših programerskih sklonosti, taj dizajn može da se implementira na mnoge načine. U prvoj fazi prevođenja dizajna u kôd možda ćete ispitati svaku pojedinačnu vrstu komande da biste utvrdili šta je sve potrebno za njeno kodiranje. Na sledeći način biste mogli, opet pomoću pseudokoda, da opišete ono što želite da vaš kôd postigne:

```
PARAGRAPH:
  Break line, flush line buffer. Advance one line between
  paragraphs. If fewer than 2 lines left on page, eject. Set
  line pointer to paragraph indent.
INDENT:
  Break line, flush line buffer. Get indent parameter. Set
  line pointer to indent parameter, set left margin to indent.
SKIP_LINE:
  Break line, flush line buffer. Get line parameter. Advance
  (parameter) lines or eject if not enough space left on
  current page.
MARGIN:
  Break line, flush line buffer. Get margin parameter. Set
  line pointer to left margin. Set right margin to margin.
PAGE:
  Break line, flush line buffer. Eject page. Set line
  pointer to left margin.
DOUBLE_SPACE:
  Set interline space to 2.
SINGLE_SPACE:
  Set interline space to 1.
BREAK:
  Break line, flush line buffer. Set line pointer to left
  margin.
```

Međutim, dodatnim pregledom ovog pseudokoda možda ćete uočiti da se pojedini koraci mogu grupisati tako da neke zajedničke funkcije budu jedna uz drugu:

```
FIRST:
  PARAGRAPH, INDENT, SKIP_LINE, MARGIN, BREAK, PAGE:
    Break line, flush line buffer.
  DOUBLE_SPACE, SINGLE_SPACE:
    No break line, no flush line buffer.
SECOND:
  INDENT, SKIP_LINE, MARGIN:
```

```
      Get parameter.
    PARAGRAPH, BREAK, PAGE, DOUBLE_SPACE, SINGLE_SPACE:
      No parameter needed.
THIRD:
    PARAGRAPH, INDENT, SKIP_LINE, MARGIN, BREAK, PAGE:
      Set new line pointer.
    DOUBLE_SPACE, SINGLE_SPACE:
      New line pointer unchanged.
FOURTH:
      Individual actions taken.
```

Pošto ste na ovaj način opisali komande, možda ćete primetiti da se grupa radnji FIRST i THIRD odnosi na istu vrstu komandi. Osim toga, primećujete da pokazivač reda zavisi od leve margine u svim slučajevima, osim ako je komanda PARAGRAPH. Koristeći ta saznanja možete detaljnije da napišete pseudokôd:

```
INITIAL:
  Get parameter for indent, skip_line, margin.
  Set left margin to parameter for indent.
  Set temporary line pointer to left margin for all but
    paragraph; for paragraph, set it to paragraph
    indent.
LINE_BREAKS:
  If not (DOUBLE_SPACE or SINGLE_SPACE), break line, flush
    line buffer and set line pointer to temporary line
    pointer.
  If 0 lines left on page, eject page and print page header.
INDIVIDUAL CASES:
  INDENT, BREAK: do nothing.
  SKIP_LINE: skip parameter lines or eject.
  PARAGRAPH: advance 1 line; if 2 lines on page, eject.
  MARGIN: right_margin = parameter.
  DOUBLE_SPACE: interline_space = 2.
  SINGLE_SPACE: interline_space = 1.
  PAGE: eject page, print page header.
```

Na kraju, spremni ste da napišete kôd za implementaciju dizajna:

```
//početak: učitati parametre
if ((command_type==INDENT)||(command_type==LINE_SKIP)
    ||(command_type==MARGIN))
  parm_value=get_parm(input_line);
if (command_type = INDENT)
  left_margin:=parm_value;
if (command_type==PARAGRAPH) temp_line_pointer=paragraph_
indent;
  else temp_line_pointer=LEFT_MARGIN;
```

```
//prekinuti trenutni red, započeti novi red
if (not((command_type==DBL_SPC)||(command_type==SNGL_SPC))){
   break_and_flush_line( ) ;
   if (lines_left==0)
     begin_new_page();
   line_pointer=temp_line_pointer ;
}
//aktivnosti za pojedinačne komande
switch(command_type){
  case LINE_SKIP:
   if (lines_left parm_value)
   for (i=1; i6parm_value; i++)
     advance_line();
   else begin_new_page();
  case PARAGRAPH:
   advance_line();
   if (lines_left 2)
     begin_new_page( ) ;
  case MARGIN: right_margin=parm_value;
  case DBL_SPC: interline_space=2;
  case SNGL_SPC: interline_space=1;
  case PAGE: begin_new_page
}//end switch
```

Prema tome, pseudokôd je poslužio kao radni okvir za konstruisanje kôda. Obratite pažnju na to da se organizacija dizajna nekoliko puta promenila prilikom izvođenja kôda iz dizajna na opisani način. Takve promene treba prijaviti projektantima i od njih dobiti saglasnost, da bi se dokumentovale i održavale veze između zahteva, dizajna i koda.

Revidirati i ponovo pisati, ne krpiti. Kada pišete kôd, kao i kad pripremate seminarski rad ili pravite umetničko delo, često prvo pravite grubu skicu, zatim je pažljivo revidirate i ponovo pišete sve dok ne budete zadovoljni rezultatom. Ako vam se čini da je tok kontrole zamršen ili da je postupak odlučivanja teško razumeti ili je teško ukloniti bezuslovna grananja, možda je vreme da se vratite na dizajn. Pregledajte ga ponovo da biste utvrdili da li problemi na koje nailazite potiču iz dizajna ili iz vašeg prevoda u kôd. Pogledajte ponovo kako su podaci predstavljeni i strukturisani, kako su izabrani algoritmi i kakvo je razlaganje.

Višekratna upotrebljivost. Postoje dve vrste višekratne upotrebljivosti: **na strani proizvođača,** kada pravimo komponente koje su planirane za upotrebu u kasnijim aplikacijama; i **na strani potrošača,** kada koristimo komponente koje su prvobitno razvijene za druge projekte (Barner i Bollinger 1991). U nekim kompanijama postoje programi koji se mogu ponovo koristiti na nivou odeljenja ili cele kompanije, kao i standardi za ocenjivanje i menjanje komponenti; u podsetniku 7.2 opisan je takav program iz firme Lucent. Uspeh ovih programa nagoveštava neke smernice za ponovno korišćenje. Ako ste vi potrošač u projektu koji gradite, pre nego što upotrebite postojeću komponentu, proverite četiri ključne karakteristike:

PODSETNIK 7.2 BIRANJE VIŠEKRATNO UPOTREBLJIVIH KOMPONENTI U LUCENTU

Kompanija Lucent Technologies započela je akciju višekratne upotrebljivosti softverskih komponenti koja važi za celu kompaniju (McClure 1997). Zato je odeljenje koje razvija softver za radne stanice oformilo savet za višekratnu upotrebljivost, sa zadatkom da postavi strategiju po kojoj će se birati komponente koje su kandidati za čuvanje u skladištu višekratno upotrebljivih komponenti. Savet se sastojao od sedmoro ljudi koji su predstavljali sve grupe u odeljenju. Savet je napravio inventar komponenti i formirao matricu sa svojstvima svih završenih i planiranih projekata. Zatim se za svako svojstvo ocenjivalo da li je implementirano i još uvek potrebno, da li je implementirano a više nije potrebno, ili nije implementirano ali je još uvek potrebno. Svojstva koja su bila potrebna, a postojala su u više od jednog projekta, označena su kao kandidati za ponovno korišćenje. U stvari, neka od njih su preradena kako bi bila pogodnija za ponovno korišćenje.

Savet se svake nedelje sastajao na dva sata i birao komponente, pregledao dizajnersku dokumentaciju tražeći komponente koje su već postojale u skladištu i nadgledao stepen ponovnog korišćenja u projektima njihovog odeljenja.

1. Da li komponenta izvršava funkciju ili pribavlja podatke koji su vam potrebni?
2. Ako je potrebna mala izmena, da li je manje potrebno za promenu ili za izgradnju komponente od nule?
3. Da li je komponenta dobro dokumentovana tako da se može potpuno razumeti bez potrebe da se verifikuje celokupna implementacija?
4. Postoji li potpuna evidencija o testiranju i revizijama komponente tako da možete biti sigurni da u njoj nema grešaka?

Takođe, morate da procenite koliko koda treba napisati da bi vaš sistem mogao da sarađuje sa višekratno upotrebljivim komponentama.

S druge strane, ako ste proizvošač višekratno upotrebljivih komponenti, trebalo bi da imate na umu nekoliko stvari:

- neka komponente budu opšte, neka koriste parametre i pretpostavljaju postojanje uslova sličnih onima u kojima ih vaš sistem poziva;
- razdvojite delove prema zavisnostima tako da odvojite delove koji su podložni promenama, od onih koji će verovatno ostati nepromenjeni;
- neka interfejs komponente bude uopšten i dobro definisan;
- uključite informacije o pronađenim i ispravljenim greškama;
- koristite jasne konvencije za imenovanje;
- dokumentujte strukture podataka i algoritme;
- neka delovi za komunikaciju i za obradu grešaka budu razdvojeni i laki za menjanje.

7.3 DOKUMENTACIJA

Mnogi korporacijski standardi i procedure stavljaju akcenat na opise koji prate skup programa. **Programskom dokumentacijom** smatramo skup opisa u pisanoj formi koji objašnjavaju šta programi rade i kako to postižu. **Unutrašnja dokumentacija** predstavlja opise koji se direktno upisuju u kôd; sva ostala dokumentacija je **spoljašnja dokumentacija**.

Unutrašnja dokumentacija

Unutrašnja dokumentacija sadrži informacije namenjene nekome ko će čitati izvorni kôd programa. Zato se daje kratak pregled za identifikaciju programa i opis njegovih struktura podataka, algoritama i toka kontrole. Obično se te informacije stavljaju na početak svake komponente u skup komentara koji nazivamo **zaglavlje bloka komentara**.

Zaglavlje bloka komentara. Kao što dobar novinar vodi računa da u priči iznese „ko, šta, gde, kada, kako i zašto" i vi morate da u zaglavlje bloka komentara svake komponente unesete sledeće informacije:

1. naziv komponente;
2. autora komponente;
3. koje je mesto komponente u celokupnom dizajnu sistema;
4. kada je komponenta napisana i revidirana;
5. zašto ta komponenta postoji;
6. kako komponenta koristi svoju strukturu podataka, algoritme i tokove kontrole.

Možemo detaljnije da ispitamo svaku od ovih informacija.

Prvo, ime komponente mora da bude uočljivo u dokumentaciji. Zatim, u bloku treba navesti autora, njegov broj telefona i adresu e-pošte da bi timovi koji rade na održavanju i testiranju mogli da ga pronađu ako imaju pitanja i komentare.

Tokom životnog ciklusa sistema, komponente se često ažuriraju i revidiraju bilo zbog ispravljanja grešaka, bilo zbog promena ili proširenja zahteva. Kao što ćemo videti u poglavlju 11, važno je pratiti revizije, pa bi u programskoj dokumentaciji trebalo da postoji evidencija izvršenih promena i autora promena.

Pošto je komponenta deo većeg sistema, blok bi trebalo da ukaže na to kako se ona uklapa u hijerarhiju komponenti. Ta informacija se ponekad iskazuje dijagramom, a ponekad je dovoljan jednostavan opis. Zaglavlje bi takođe trebalo da objasni kako se komponenta poziva.

Za objašnjenje kako komponenta izvršava svoj cilj, potrebne su detaljnije informacije. U bloku bi trebalo navesti:

- ime, tip i svrhu svake veće strukture podataka i promenljive;
- kratak opis logičkog toka, algoritama i obrade grešaka;
- očekivani ulaz i mogući izlaz;
- pomoćna sredstva za testiranje i kako ih treba koristiti;
- očekivana proširivanja ili revizije.

Standardi organizacije obično definišu redosled i sadržaj zaglavlja bloka komentara. Evo kako bi moglo da izgleda uobičajeno zaglavlje bloka komentara:

```
PROGRAM SCAN: Program kojim se u jednom redu teksta traži dati
znak
PROGRAMER: Beatrice Clarman (718) 345-6789/bc@power.com
POZIVNI NIZ: CALL SCAN(DUŽINA,ZNAK,NTEKST)
  gde je DUŽINA dužina reda koji treba pretražiti;
  ZNAK je znak koji se traži. Red teksta se predaje
  kao matrica NTEKST.
VERZIJA 1: napisana 3 novembra 2000, autor B. Clarman
REVIZIJA 1.1: 5 decembra 2001, autor B. Clarman da se poboljša
algoritam
  traženja.
SVRHA: Modul za pretraživanje opšte svrhe, za pretraživanje
bilo kakvog
  reda teksta, bez obzira na dužinu. Jedan iz grupe pomoćnih
programa za
  tekst projektovanih za dodavanje znaka u red teksta,
  čitanje jednog znaka, promenu jednog znaka ili brisanje
znaka.
STRUKTURE PODATAKA: Promenljiva DUŽINA - integer
  Promenljiva ZNAK - character
  Matrica NTEKST - matrica znakova dužine DUŽINA
ALGORITAM: Čitati matricu NTEKST znak po znak; ako se pronađe
  ZNAK, mesto u matrici NTEKST vratiti u promenljivu
  DUŽINA; inače promenljiva DUŽINA dobija vrednost 0.
```

Ostali komentari u programu. Zaglavlje bloka komentara služi kao uvod u vaš program, kao što u knjizi uvod objašnjava njenu namenu. Dodatni komentari usmeravaju čitaoca dok čita vaš program, pomažu mu da shvati kako su u kodu implementirane namere koje su opisane u zaglavlju. Ako organizacija koda reflektuje dobro strukturisan dizajn, ako su naredbe jasno formatirane i ako su oznake, imena promenljivih i imena podataka opisna i lako prepoznatljiva, tada će broj dodatnih komentara biti mali. To jest, ako se poštuju jednostavne preporuke za format i strukturu koda, on će sam po sebi biti izvor informacija.

Komentari imaju svoje mesto čak i u jasno strukturisanom i dobro napisanom kodu. Mada jasnoća koda i struktura umanjuje potrebu za dodatnim komentarima, oni su korisni gde god se komponenti mogu dodati pomoćne informacije. Osim što objašnjavaju šta program radi red po red, komentari mogu i da podele kôd na faze koje predstavljaju glavne aktivnosti. Zatim se svaka aktivnost može podeliti na još manje korake od kojih svaki sadrži samo nekoliko redova. Pseudokôd iz programskog dizajna može da posluži u ovu svrhu i može da se ugradi u sam kôd. Takođe, kada se kôd revidira, programeri bi trebalo da ažuriraju i komentare u skladu sa unetim promenama. Na taj način, komentari formiraju evidenciju revizija u vremenu.

Bitno je da komentari reflektuju ponašanje samog koda. Osim toga, potrudite se da komentari pruže dodatne informacije, a ne da opisuju ono što je već očigledno iz vašeg dobrog korišćenja labela i imena promenljivih. Na primer, nepotrebno je pisati

```
// Povećati i3
i3 = i3 + 1;
```

kada možete dodati daleko važniju informaciju ako napišete

```
// Podesiti brojač za učitavanje sledećeg znaka
i3 = i3 + 1;
```

Idealno je kada ime promenljive objašnjava aktivnost:

```
brojač_znakova = brojač_znakova + 1;
```

Obično se kodiranje započinje prelaskom sa dizajna na pseudokôd, koji zatim obezbeđuje radni okvir za vaš konačni kôd i osnovu za komentare. Pišite komentare dok pišete kôd, a ne kasnije, da biste zabeležili i dizajn i vašu nameru. Čuvajte se koda koji je teško komentarisati; ta teškoća je često znak da treba uprostiti dizajn pre nego što završite kodiranje.

Razumljiva imena promenljivih i oznake naredbi. Za promenljive i naredbe birajte imena koja opisuju njihovu ulogu. Ako napišete

```
nedeljna plata = (cena sata * sati) + (.5)* (cena sata) * (sati
- 40.);
```

biće čitaocu jasnije nego

```
z = (a * b) + (.5) * (a) * (b - 40.);
```

U stvari, primer *nedeljna plata* verovatno neće morati uopšte da se komentariše i manja je verovatnoća da se u njemu pogreši.

Slično tome, tekstualne oznake naredbi bi trebalo da upute čitaoce u to šta označeni deo programa radi. Ako oznake moraju da budu numeričke, povedite računa o tome da budu u rastućem redosledu i grupisane prema svrsi.

Unapređenje razumljivosti pomoću formatiranja. Format komentara može da pomogne čitaocu da razume svrhu vašeg koda i način na koji se ta svrha ostvaruje. Uvlačenje i prored naredbi može da objasni i osnovnu strukturu kontrole. Obratite pažnju na to kako ovakav kôd bez uvlačenja:

```
if (xcoord < ycoord)
result = -1;
elseif (xcoord == ycoord)
if (slope1 > slope2)
result = 0;
else result = 1;
elseif (slope1 > slope2)
result = 2;
elseif (slope1 < slope2)
result = 3;
result = 4;
```

može da postane mnogo jasniji ako se upotrebi uvlačenje i preurede razmaci:

```
if (xcoord < ycoord) result = -1;
elseif (xcoord == ycoord)
     if (slope1 > slope2) result = 0;
                  else result = 1;
elseif (slope1 > slope2) result = 2;
elseif (slope1 < slope2) result = 3;
else         result = 4;
```

Osim što se pomoću formatiranja može prikazati struktura kontrole, Weinberg (1971) preporučuje formatiranje naredbi tako da komentari budu na jednom delu stranice, a naredbe na drugom. Na ovaj način možete da prekrijete komentare dok testirate program kako vas ne bi zavela nepravilna dokumentacija. Na primer, sledeći kôd (iz knjige Lee i Tepfenhart 1997) može se čitati bez komentara, ako se gleda samo levi deo stranice.

```
void free_store_empty()
{
   static int i = 0;
   if(i++ == 0)                          //sprečavanje da cerr
                                         //dodeli memoriju
      cerr << "Out of memory\n";         //obavestiti korisnika
   abort();the                           //odustati
}
```

Dokumentovanje podataka. Čitaocima programa najteže je da shvate strukturu podataka i njihovo korišćenje. Mapa podataka je veoma korisna za tumačenje postupaka u kodu, pogotovo kada sistem obrađuje više datoteka različitog tipa i namene i kada koristi veliki broj indikatora i parametara. Ta mapa bi trebalo da odgovara rečniku podataka iz spoljne dokumentacije, da bi čitalac mogao da prati manipulisanje podacima, počevši od zahteva, preko dizajna, pa sve do koda.

Objektno orijentisani dizajn svodi na minimum ili čak eliminiše neke od ovih problema, ali ponekad to sakrivanje informacija otežava čitaocu da tačno shvati kako se vrednost podataka menja. Zbog toga bi unutrašnja dokumentacija trebalo da sadrži opise strukture podataka i njenog korišćenja.

Spoljna dokumentacija

Dok je unutrašnja dokumentacija koncizna i piše se na nivou koji odgovara programeru, spoljna dokumentacija je namenjena i onima koji možda nikad neće čitati sam kôd. Na primer, projektanti mogu da pregledaju spoljnu dokumentaciju kada razmatraju izmene i unapređivanja. Osim toga, spoljna dokumentacija omogućava da se stvari opširnije objasne nego što je to moguće sa komentarima u programu. Ako posmatrate zaglavlje bloka komentara kao pregled ili rezime vašeg programa, onda je spoljna dokumentacija celokupan izveštaj. Ona se bavi istim pitanjima - ko, šta, zašto, kada, gde i kako - samo iz perspektive sistema, a ne iz perspektive komponente.

Pošto se softverski sistem gradi od međusobno povezanih komponenti, spoljna dokumentacija često sadrži opšti pregled komponenti sistema ili nekoliko grupa komponenti (komponente korisničkog interfejsa, komponente za upravljanje bazom podataka i komponente osnovnih izračunavanja). Dijagrami praćeni tekstom u kojem se opisuje svaka komponenta, pokazuju kako se podaci dele i koriste u jednoj komponenti ili u više njih; opšti pregled uglavnom opisuje kako se informacije prenose iz jedne komponente u drugu. Ovde se opisuju klase objekata i njihova hijerarhija nasleđivanja, kao i razlozi za definisanje posebnih tipova ili kategorija podataka.

Spoljna dokumentacija komponenti čini deo sveukupne dokumentacije sistema. U trenutku kada se komponenta piše, veći deo obrazloženja za strukturu i tok komponente već je detaljno opisan u dokumentima dizajna. Na neki način, dizajn je okosnica spoljne dokumentacije koja se dopunjava tekstualnim opisom pojedinosti koda komponente.

Opis problema. U prvom odeljku dokumentacije kôda trebalo bi da objasnite problem koji komponenta rešava. U ovom odeljku se postavlja osnovica za opisivanje razmatranih opcija rešenja i razloga zbog kojih je izabrano konkretno rešenje. Opis problema nije ponavljanje dokumentacije zahteva, već opšti opis okvira sa objašnjenjem kada se komponenta poziva i zašto je ona potrebna.

Opis algoritama. Pošto ste razjasnili zašto komponenta postoji, trebalo bi da obrazložite izbor algoritama. Trebalo bi da objasnite svaki algoritam koji se koristi u komponenti, uključujući formule, ograničenja ili posebne uslove, pa čak i njihovo izvođenje ili reference na knjigu ili časopis iz kojih su preuzeti.

Ako za algoritam postoje posebni slučajevi, opišite svaki od njih i objasnite kako se on obrađuje. Ako se neki slučajevi ne obrađuju zato što se ne očekuje njihovo pojavljivanje, iznesite obrazloženje i opišite odgovarajuću obradu grešaka u kodu. Na primer, u nekom algoritmu može postojati obrazac u kojem se jedna promenljiva deli drugom promenljivom. U dokumentaciji bi trebalo pomenuti slučajeve kada je imenilac jednak nuli, ističući kada može doći do te situacije i kako je taj slučaj obrađen u kodu.

Opis podataka. U spoljnoj dokumentaciji bi korisnici ili programeri trebalo da vide tok podataka na nivou komponente. Dijagrame toka podataka trebalo bi da prate relevantne reference na rečnike podataka. Kod objektno orijentisanih komponenti bi opšti pregled objekata i klasa trebalo da opiše glavne interakcije objekata.

7.4 PROCES PROGRAMIRANJA

U ovom poglavlju opisuju se smernice, standardi i dokumentacija. Kakvu vrstu postupka primenjuju programeri da bi obezbedili visok kvalitet svog koda? Do nedavno se proces programiranja nije mnogo proučavao. Kako ističu Whittaker i Atkin (2002), istraživači u domenu softverskog inženjerstva često polaze od pretpostavke da će na osnovu dobrog dizajna svaki programer napraviti solidan kôd. Ali, dobar kôd nije rezultat samo dobrog dizajna već takođe i veštine, iskustva i umeća koji potiču od maštovitosti i dobrog rešavanja problema. Da biste shvatili koji postupci podržavaju dobru praksu programiranja, prvo ćemo pogledati kako se rešavaju problemi.

Programiranje kao rešavanje problema

Polya (1957) je klasična knjiga o načinu rešavanja problema. On ističe da se pronalaženje dobrog rešenja sastoji od četiri odvojene faze:

1. Razumevanje problema.
2. Smišljanje plana.
3. Izvršavanje plana.
4. Provera.

Da bismo razumeli problem, analiziramo sve njegove elemente. Šta je poznato? Šta nije poznato? Šta se može saznati? Šta se ne može znati? Ova pitanja odnose se i na podatke i na relacije. Kao što dobar analitičar sistema prikazuje problem u smislu granica sistema ukazujući na to šta se nalazi unutar granice, a šta je izvan nje, dobar rešavač problema mora da razume uslove u kojima problem mora da se rešava. Zatim, pošto definišemo uslove, mi ispitujemo da li se uslovi mogu zadovoljiti.

Ponekad je problem lakše razumeti pomoću slike. Programeri to često postižu crtanjem dijagrama toka, koji pokazuje kako dinamički alocirana struktura podataka menja oblik dok se nad njom vrše različite operacije. Takvo prikazivanje nam pomaže da prepoznamo različite delove uslova, a možda čak i da razložimo problem na manje probleme koji se lakše savlađuju. Na neki način, preformulisanje radi boljeg razumevanja je aktivnost mikrodizajna: dizajn programa se pažljivo posmatra i reorganizuje u delove koji se lakše savlađuju nego prvobitni okvir dizajna.

Zatim se smišlja plan kako da se rešenje izvede od onog što znamo o postojećem problemu. Kakve su veze između podataka i nepoznatih? Ako veze nisu trenutno vidljive, da li problem na neki način liči na neke druge poznatije probleme? Ovde razmišljamo o trenutnom problemu u smislu šablona: postoji li šablon ili skup šablona koji možemo da prilagodimo našem problemu?

Da bismo pronašli pravilan plan, možemo da isprobamo nekoliko tehnika:

- *Prepoznavanje sličnog problema*: postoje li podaci, algoritmi, biblioteke ili metodi koji mogu da se upotrebe za rešavanje našeg problema?
- *Preformulisanje problema*: koje su ključne definicije? Može li problem da se uopšti ili da se konkretizuje da bi se lakše savladao? Mogu li se postaviti pretpostavke koje će uprostiti problem?
- *Razlaganje problema*: Koji su osnovni elementi problema? Mogu li se podaci grupisati u kategorije tako da se svaka kategorija obrađuje zasebno?

Perkins (2001) predlaže grupnu diskusiju za traženje rešenja, da bi se postigao efekat koji on naziva „eureka": prepoznavanje najboljeg rešenja. On predlaže posmatranje više različitih mogućnosti, proširivanje misli i razmatranje čak i nemogućih scenarija. Treba takođe tražiti suptilne znakove koji upozoravaju na anomalije - razloge zbog kojih neko od mogućih rešenja ne bi funkcionisalo. Osim preuređivanja problema, Perkins predlaže „promenu usmerenja" sa trenutnih fiksacija, ispitivanjem svake pretpostavke i stava. On savetuje da posmatramo kreativnost kao uzvišeni oblik dovođenja svega u pitanje; da

koristimo svoju kreativnost kako bismo doveli u pitanje predložena rešenja i pronađemo bolje rešenje za naš problem.

Pošto izaberemo plan napada, Polyà nalaže da ga sprovedemo. Moramo, korak po korak, da proverimo da li je rešenje (u našem slučaju rešenje je program) ispravno. Ako je potrebno, možemo za procenu svakog koraka da koristimo neke od tehnika predstavljenih u poglavljima 8 i 9 i proverimo da li svaki logički iskaz proističe iz prethodnih koraka. Kada je pronađeno rešenje, vraćamo se i ispitujemo ga i ponovo pregledamo svaki deo obrazloženja. U isto vreme procenjujemo rešenje u smislu primenljivosti na druge situacije. Da li je rešenje višekratno upotrebljivo? Može li se upotrebiti za rešavanje drugačijeg problema? Može li da postane osnova za šablon?

Svaki proces programiranja koji ide ovim putem može da poveća brzinu kojom nalazimo dobra rešenja. Biblioteke programa, dobra dokumentacija i automatizovane alatke podržavaju dobar proces. Kao što ćemo videti u kasnijim poglavljima, skladišta za ponovno korišćenje, upravljanje promenama i obdukcije takođe pomažu proces, pogotovo kroz analizu uspeha i neuspeha. Ako u programiranju koristimo zaključke svojih analiza, programiranje će biti svaki put sve lakše.

Ekstremno programiranje

U poglavlju 2 smo učili o brzim metodama i odgovarajućoj filozofiji programiranja koje nazivamo ekstremnim programiranjem. U ekstremnom programiranju imamo dve vrste učesnika: kupce i programere. Kupci predstavljaju korisnike budućeg sistema. U toj ulozi oni izvršavaju sledeće aktivnosti:

- definišu svojstva koja će programeri da implementiraju, koristeći formu scenarija da opišu način funkcionisanja sistema;
- opisuju detaljne testove koje će izvršavati kada softver bude spreman, kako bi proverili da li su scenariji pravilno implementirani;
- dodeljuju prioritete scenarijima i njihovom testiranju.

Zatim programeri pišu kôd kojim implementiraju scenarije i to redosledom kojim su kupci postavili prioritete. Da bi pomogli kupcima, programeri moraju da procene koliko će trajati kodiranje svakog scenarija kako bi kupci mogli da planiraju testove prihvatanja. Planiranje se vrši za periode od po dve nedelje i tako se gradi poverenje između kupaca i programera. Martin (2000) ističe da se ekstremno programiranje ne svodi samo na ovaj dijalog između programera i kupaca, već da postoji i dijalog između programera kada rade u parovima: sede zajedno za radnom stanicom i sarađuju na izgradnji jednog programa.

Programiranje u parovima

Koncept programiranja u parovima je jedna od radikalnijih tehnika brzih metoda razvoja i zato ćemo ga temeljnije razmotriti u ovom poglavlju. Kod programiranja u parovima, jedan par programera koristi istu tastaturu, monitor i miša. Svaki član u tom paru preuzima veoma specifičnu ulogu. Jedna osoba je vozač ili pilot koji kontroliše računar i fizički upisuje kôd. Druga osoba je navigator koji pregleda vozačev kôd i obezbeđuje povratne informacije. Dva člana takvog tima povremeno menjaju uloge.

O programiranju u parovima izneta su do sada mnoga različita mišljenja, pa tako neki tvrde da ono podiže produktivnost i kvalitet. Međutim, dokazi su tanki i često dvosmisleni. Na primer, Parrish i dr. (2004) izveštavaju o uticaju na produktivnost koji se dobija upotrebom parova dizajnera. Programeri iz te studije su imali u proseku po deset godina radnog iskustva. U studiji se saradnja programera merila kao stepen do kojeg su parovi programera u istom danu radili na istom modulu. Parrish i njegove kolege su utvrdili da su „parovi koji su imali veći stepen istovremenog rada bili značajno manje produktivni i pored toga što su bili iskusni i obučeni za tu metodologiju". Štaviše, studijom je utvrđeno da parovi nisu bili ništa produktivniji u radu na kasnijim modulima nego na početnim. Međutim, izveštaji Instituta za softversko inženjerstvo (SEI) (iz istraživanja o personalnom procesu izrade softvera) nagoveštavaju da programeri koji koriste tradicionalne metode postaju sve produktivniji i kvalitetniji kako stiču više prakse. Uz zaključak da parovi koji zajedno rade nisu po prirodi produktivniji, Parrish i njegove kolege napominju da postoje načini da se prevaziđe prirodno opadanje produktivnosti do kojeg dolazi u parovima.

Druge studije, kao što je Cockburn i Williams (2000), Nosek (1998), i Williams i dr. (2000), nalaze da se programiranjem u parovima kôd proizvodi brže i sa manjom gustinom grešaka nego u tradicionalnom individualnom programiranju. Međutim, Padberg i Müller (2003) su utvrdili da je odnos troškova i koristi za programiranje u parovima zavisio od toga kolike zaista bile prednosti. Konkretno, oni dokazuju da se programiranje u parovima isplati samo pod jakim pritiskom tržišta, gde rukovodioci podstiču programere na brzo implementiranje sledećeg skupa svojstava za sledeću reviziju. Drobka, Noftz i Raghu (2004) pokazuju različite rezultate: u jednoj studiji je tim koji je primenio ekstremno programiranje bio produktivniji za 51% od tima koji je koristio tradicionalne pristupe, ali u drugoj studiji je tim programera koji nije koristio ekstremno programiranje bio produktivniji za 64%. Mnoge od ovih studija rađene su na malim nerealističnim projektima; kod drugih se možda pojavio takozvani efekat Hawthorne (efekat „kalemljenja") u kojem ponašanje nije bolje zato što su postupci bolji, već zato što učesnike posmatraju. U poglavlju 14 videćemo kako da donesemo odluke na osnovu ovako različitih, i često kontradiktornih dokaza.

Ostale prednosti programiranja u parovima izgledaju jasne. Ako stariji iskusan programer daje savete početniku dok on piše kôd, to može biti značajno iskustvo u učenju. Ako još jedan par očiju pregleda kôd dok se on razvija, greške će se uočiti u ranijim fazama. Međutim, Skowronski (2004) ističe da programiranje u parovima može da omete bitne korake u rešavanju problema. On tvrdi da neophodna interakcija među članovima može da poremeti usredsređivanje na problem. „Brze metode mogu da potisnu ljude koji su odlični u rešavanju problema, a da istaknu one članove programskog tima koji umeju dobro da govore, ali možda nemaju analitičke sposobnosti potrebne za teške zadatke dizajna i kodiranja" (Skowronski 2004). Štaviše, Polyà (1957) i Perkins (2001) ističu da je dobrim rešavaocima problema potrebno vreme u kojem će sami pažljivo da razmisle, smisle svoje planove i analiziraju opcije. Takvo vreme nije predviđeno u procesu programiranja u parovima.

Kuda ide programiranje?

Kao i kod svih drugih stvari, odgovor bi mogao da bude „sve sa merom". Drobka, Noftz i Raghu (2004) opisuju svoje korišćenje ekstremnog programiranja na četiri kritična

projekta. Otkrili su da su bili potrebni dodatni koraci da bi se ekstremno programiranje prilagodilo realnosti obimnog kritičnog softvera. Na primer, bez arhitektonskih smernica svaki programer je imao vlastitu viziju sveukupne arhitekture i te su se vizije značajno razlikovale. Zato je dokument sa arhitekturom višeg nivoa dat projektantima kao pomoćno sredstvo za planiranje. Umesto priča sa prioritetima, istraživači su otkrili da je bitno definisati osnovnu arhitekturu sa dokumentovanim scenarijima za opisivanje ključnih apstrakcija i definisanje granica sistema. Na kraju, dokumentacija se koristila kao pomoć u komunikaciji među članovima tima. Strogo brze metode preporučuju što manje dokumentacije, ali su Drobke, Noftz i Raghu utvrdili da je ona bitna. Isto važi i za revizije dizajna, jer se njima obezbeđuje da će softver koji gradimo moći da se održava tokom svog procenjenog veka trajanja od 30 godina.

Na taj način, u budućnosti će programiranje biti mešavina dobre prakse koja se preporučivala desetinama godina unazad i fleksibilnih pristupa za koje se zalažu pristalice brzih metoda. Videćemo u poglavlju 14 da softverski inženjeri definišu korpus znanja o softverskom inženjerstvu u kojem se opisuju znanja i veštine koje bi dobar softverski inženjer trebalo da savlada da bi mogao da izrađuje veoma kvalitetne sisteme.

7.5 PRIMER IZ INFORMACIONIH SISTEMA

Setite se da smo u poglavlju 5 razmatrali deo dizajna za sistem Piccadilly. Jedan aspekt tog dizajna sastojao se u pronalaženju televizijskih programa konkurentskih kanala da bismo koristili informacije o njima u svojoj reklamnoj kampanji. Deo opisa dizajna u kojem sistem utvrđuje koji programi su po rasporedu u isto vreme kada i epizode programa Piccadilly, izgleda ovako:

Ulaz: *Raspored konkurencije*
For each *Ime televizijske kompanije*, napraviti *Konkurentska kompanija.*
For each *Raspored konkurencije*
Pronaći *Epizodu* gde je *Datum epizode=Konkurentski datum emitovanja*
I *Početno vreme epizode=Konkurentsko vreme emitovanja*
Napraviti primerak *Konkurentski program*
Napraviti relacije *Planiranje* i *Nadmetanje*
Izlaz: *Spisak konkurentskih programa*

Uz ovaj deo dizajna priloženi su dijagrami, kao i opisi rečnika podataka:

Raspored konkurencije = *Tok podataka *
Ime televizijske kompanije
+ {Konkurentski datum emitovanja
+ Konkurentsko vreme emitovanja + Ime konkurentskog programa
+ (Predviđena ocena konkurencije)}

Za pisanje koda kojim će se implementirati ovaj deo sistema Piccadilly, potrebne su mnoge odluke na nivou komponente. Istražićemo jednu od njih.

Pretpostavite da želimo da implementiramo sistem objektno orijentisanim jezikom kao što je C++. Napravićemo hijerarhiju klasa u kojoj će postojati klasa za *Raspored konkurencije*. Prema rečniku podataka, klasa *Raspored konkurencije* imaće atribute *Ime televizijske kompanije, Konkurentski datum emitovanja, Konkurentsko vreme emitovanja, Ime konkurentskog programa* i *Predviđena ocena konkurencije* (ovaj poslednji atribut nije obavezan). Hoćemo da napravimo metod za klasu *Raspored konkurencije* koji uparuje konkurentske programe sa *Epizodom* gde je *Datum epizode* jednak *Konkurentskom datumu emitovanja*, a *Početno vreme epizode* jednako *Konkurentskom vremenu emitovanja*. Jedna od odluka koju moramo da donesemo je način prenošenja informacija o *Epizodi* u komponentu koja je to zahtevala. Postoji nekoliko mehanizama za predavanje argumenta: predavanje argumenta pomoću vrednosti, pomoću reference, pomoću pokazivača ili kao matrice.

Kada predajemo argument pomoću vrednosti, sama vrednost *Epizode* se ne menja. Umesto toga se pravi kopija i ona se stavlja u lokalni stek. Kada se metod završi, lokalne vrednosti mu više nisu dostupne. Prednost ovog rešenja je to što pozivajuća komponenta ne mora da čuva i obnavlja vrednosti argumenta. Međutim, predavanje po vrednosti može da utroši dosta vremena i prostora ako je argument velik. Štaviše, ova tehnika nije korisna ako metod treba da promeni samu vrednost argumenta. Ako implementiramo metod predavanjem po vrednosti, to bi u jeziku C++ izgledalo ovako:

```
void Match:: calv(Episode episode_start_time)
{
first_advert = episode_start_time + increment;
// Sistem pravi kopiju Epizode
// a vaš program može direktno da koristi vrednosti.
}
```

Druga mogućnost jeste da se argument preda kao pokazivač. Argument se ne kopira; umesto toga, metod dobija pokazivač na primerak *Epizode*. U ovom slučaju moguće je promeniti stvarnu vrednost argumenta, a rutina može da poziva metode klase *Epizoda*. Odgovarajući kôd bi mogao da izgleda ovako:

```
void Match:: calp(Episode* episode)
{
episode->setStart (episode->getStart());
// U ovom primeru predaje se pokazivač na primerak Epizode.
// Zatim rutina može da poziva usluge (kao što je setStart
// i getStart) klase Epizoda pomoću operatora "->".
}
```

Na kraju, argument može da se preda kao referenca. I ovde se ne pravi kopija argumenta; umesto toga, metod prima adresu argumenta. I u ovom slučaju može da se menja stvarna vrednost argumenta. Odgovarajući kôd bi mogao da izgleda ovako:

```
void Match:: calr(Episode& episode)
{
episode.setStart (episode.getStart());
// U ovom primeru predaje se adresa Epizode.
```

```
// Zatim rutina može da poziva usluge (kao što je setStart
// i getStart) klase Epizoda pomoću operatora ".".
}
```

Pošto odlučite na koji način ćete predati argument, taj izbor bi trebalo da dokumentujete komentarima u kodu, i u spoljnoj dokumentaciji. Na taj način će drugi programer, koji će možda kasnije ponovo da koristi ili da ažurira vaš kôd, moći da shvati vaše odluke na niskom nivou dizajna i očuva konsistentnost novog koda sa starim.

7.6 PRIMER IZ PRAKSE

Videli smo da je glavni problem sa softverom za Ariane-5 bila potreba da se greške i otkazivanja obrade na odgovarajući način. Takve situacije mogu da utiču i na izbor jezika za implementaciju. Coleman i dr. (1994) opisuju nekoliko objektno orijentisanih jezika i njihove mogućnosti za obradu otkaza.

Jedan od popularnih načina za obradu grešaka je uključivanje izuzetka. **Izuzetak** je stanje koje kada se otkrije izaziva prenošenje kontrole sistema jednom posebnom delu koda koji nazivamo rutina za obradu izuzetka. Zatim, **rutina za obradu izuzetka** poziva kôd projektovan da ispravi osnovnu grešku, ili da barem prebaci sistem u stanje koje je prihvatljivije od stanja izuzetka. Dizajn po ugovoru često u ugovoru sadrži konkretan opis ponašanja za obradu izuzetaka.

Jezik Eiffel (Meyer 1992b) sadrži eksplicitne mehanizme za obradu izuzetaka. Ako nastupi izuzetak za vreme izvršavanja nekog metoda, za rešavanje problema se poziva poseban kôd koji nazivamo **spasilački kôd**. Potrebne su odluke na nivou dizajna o tome kako da se izvrši spasavanje. Ponekad se problem ispravlja i zatim se pokuša ponovno izvršavanje metoda. U drugim slučajevima spasilački kôd završava svoj posao, a zatim predaje kontrolu drugoj rutini za obradu izuzetka; ako problem ne može da se ispravi, sistem prelazi u stanje iz kojeg može bezbedno i elegantno da se prekine. U dizajnu po ugovoru, ugovor sadrži preduslove, tvrđenja i postuslove. Za obradu izuzetaka jezik Eiffel takođe sadrži druge postuslove koji objašnjavaju u kojem stanju treba sistem da se nađe ako tvrđenje nije tačno. Prema tome, da je kôd za Ariane-5 bio pisan u jeziku Eiffel, mogao je da sadrži postuslove koji bi zaustavili podsistem SRI pre nego što je on doveo do skretanja rakete sa putanje.

S druge strane, C++ prevodioci ne sadrže standardne rutine za obradu izuzetaka. Coleman i dr. (1994) ističu da je moguće implementirati kôd u stilu jezika Eiffel kao:

```
try
 {
 }
catch (...)
 {
 // pokušati ispravljanje stanja
 // zadovoljiti postuslov ili ponovo uključiti izuzetak
 }
```

Bez obzira na izabrani jezik i strategiju za obradu izuzetaka, važno je da politika bude ista u celom sistemu, a ne da se u svakoj komponenti koristi drugi pristup. Sa konsistentnim pristupom lakše se otkrivaju greške i pronalazi njihov osnovni uzrok. Iz

istog razloga je korisno čuvati što je moguće više informacija o stanjima da bi se mogli rekonstruisati uslovi koji su doveli do otkazivanja.

7.7 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju smo razmotrili nekoliko smernica za implementiranje programa. Videli smo da prilikom pisanja koda treba imati na umu sledeće:

- standarde i smernice koje postoje u organizaciji;
- ponovno korišćenje koda iz drugih projekata;
- pisanje koda tako da bude upotrebljiv za ponovno korišćenje u budućim projektima;
- korišćenje dizajna niskog nivoa kao početnog radnog okvira i prelaženje od dizajna ka kodu u nekoliko iteracija;
- pridržavanje iste strategije za obradu grešaka u celom sistemu;
- korišćenje dokumentacije unutar programa i spoljne dokumentacije da bi se objasnila organizacija, podaci, kontrola i funkcija koda, kao i odluke o dizajnu;
- očuvanje atributa kvalitetnog dizajna u kodu;
- odlučivanje o jeziku za implementaciju na osnovu aspekata dizajna.

Postoji više dobrih knjiga sa specijalizovanim savetima vezanim za implementaciju na bazi izbora konkretnih programskih jezika.

7.8 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Mada veliki deo kodiranja predstavlja individualni poduhvat, pri tom treba uvek imati na umu ceo tim. Sakrivanje informacija omogućava da se otkriju samo informacije bitne za razmatranu komponentu da bi vaše kolege mogle da je pozivaju i ponovno koriste. Korišćenjem standarda unapređuje se komunikacija među članovima tima. Ako koristite zajedničke tehnike i strategije dizajna, vaš kôd će se lakše testirati, održavati i ponovno koristiti.

7.9 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Postoje mnogi aspekti programiranja koji zaslužuju detaljnije istraživanje.

- Potrebno je više informacija o karakteristikama dobrih programera. Produktivnost može da varira čak deset puta i kvalitet je takođe veoma promenljiv. Ako budemo znali koje karakteristike pogoduju brzom razvoju kvalitetnog koda, lakše ćemo obučavati projektante da budu uspešniji i produktivniji.
- Teško je prepoznati komponente sa najvećim potencijalom za ponovno korišćenje. Potrebna su merenja i procene da bi se saznalo koji atributi komponenti daju najbolju prognozu o mogućnosti ponovnog korišćenja.
- I dalje su potrebna dodatna istraživanja o karakteristikama programskih jezika i njihovog uticaja na kvalitet proizvoda. Na primer, Hatton (1995) objašnjava da treba izbegavati neke nestandardne aspekte jezika C kako bi softver bio bezbedniji i pouzdaniji.

- Automatizovani alati su uvek korisni za automatsko generisanje koda, upravljanje skladištima koda, nametanje dizajna po ugovoru i obezbeđivanje šablona za standardne strukture kodiranja. Istraživači ne samo što i dalje grade nove alate, već takođe procenjuju postojeće u smislu njihove upotrebe u praksi na velikim projektima.

7.10 SEMINARSKI RAD

U prethodnim poglavljima ste ispitali zahteve za *Aranžer pozajmica* i ponudili nekoliko dizajna za implementiranje vašeg rešenja za opisani problem. Sada je vreme da sistem implementirate. Postoji nekoliko područja koja zaslužuju posebnu pažnju zato što su teža nego što to izgleda na osnovu njihovog opisa u zahtevima. Jedno je potreba da se dozvoli da sistem istovremeno koriste četiri analitičara za zajmove. Taj zahtev je trebalo da vam nagovesti potrebu da se *Aranžer pozajmica* projektuje tako da se neki zajmovi zaključaju tokom razmatranja njihovog uključivanja u paket. Koji je najbolji način da se to implementira? Kako se može izvršiti implementacija da bi se najlakše menjao broj analitičara koji mogu istovremeno da koriste sistem?

Još jedna teškoća u implementaciji javlja se kada implementirate algoritam za paketiranje pozajmica. Kako ćete pomoću koda pronalaziti najbolji paket koji zadovoljava kriterijume? Imajte na umu da korisniku morate brzo da odgovorite.

Na kraju, kako vaš izbor programskog jezika utiče na mogućnost ispunjavanja zahteva u pogledu performanse?

7.11 KLJUČNE REFERENCE

Postoji nekoliko dobrih knjiga sa smernicama za programiranje: Kernighan i Plauger (1976, 1978); Hughes, Pfleeger i Rose (1978); Barron i Bishop (1984); Bentley (1986, 1989) i McConnell (1993).

Skladišta višekratno upotrebljivih komponenti postoje po celom svetu. Neka od najpopularnijih su:

- ASSET (Asset Source for Software Engineering Technology);
- CARDS (Central Archive for Reusable Defense Software);
- COSMIC (Computer Software Management and Information Center);
- DSRS (Defense Software Repository System);
- Software Technology Support Center u vazduhoplovnoj bazi Hill Air Force.

Kontaktne informacije za ove i druge resurse višekratno upotrebljivih komponenti mogu se naći na pratećoj veb stranici uz ovu knjigu.

7.12 VEŽBE

1. U svakom jeziku se **izračunatim slučajem** naziva naredba kojom se vrši grananje na nekoliko mesta u programu, zavisno od vrednosti jedne promenljive. Opišite prednosti i mane takve naredbe. Konkretno, na koji način ona utiče na tok kontrole? Na mogućnost održavanja? Na mogućnost ponovnog korišćenja?

2. Ako je jedna osoba napisala komponentu ali su je druge prepravljale, ko je kriv ako komponenta loše radi? Kakve su zakonske i etičke implikacije ponovnog korišćenja tuđe komponente?
3. Lista je struktura podataka koja se može rekurzivno definisati. Napišite rekurzivnu definiciju liste. Ako znate neki programski jezik koji ima rekurzivne procedure (kao što su LISP ili PL/I), objasnite kako se u tom jeziku elementi dodaju na listu i brišu iz nje.
4. Navedite primer kako se obrada liste lakše shvata u jeziku projektovanom za rekurziju, nego u jeziku u kojem to nije predviđeno.
5. Imate zahtev da napišete program koji će štampati kalendar za godinu dana. Korisnik upisuje željenu godinu, a rezultat je kalendar za tu godinu. Opišite na koji način predstavljanje unutrašnjih podataka utiče na način na koji se program piše. Navedite nekoliko primera struktura podataka koje bi mogle da se upotrebe za ovakav problem. (*Savet*: Da li su strukture podataka kumulativne ili nisu? Kako se obrađuje prestupna godina?)
6. Za uobičajeni algoritam rešavanja kvadratne jednačine pomoću kvadratne formule potrebno je u kodu razmotriti nekoliko specijalnih slučajeva. Napišite odgovarajuće komentare za ovaj algoritam tako da oni olakšaju uočavanje različitih slučajeva i načina na koji su oni obrađeni. Napišite propratnu spoljnu dokumentaciju koja će objasniti algoritam.
7. Utvrdite koji je algoritam straničenja u računarskom operativnom sistemu sa kojim ste upoznati. Napišite spoljnu dokumentaciju za algoritam tako da korisniku objasnite kako se straniči.
8. Pregledajte program koji ste predali kao projekat na nekom drugom predavanju. Može li se on poboljšati pomoću saveta iz ovog poglavlja? Kako? Da li primenom ovih saveta vaš program postaje efikasniji ili manje efikasan?
9. Kakve su prednosti i mane korišćenja istog standardizovanog jezika ili alatki za sve aplikacije u vašoj organizaciji?
10. Kada se komponente koda generišu automatski pomoću nekog alata ili se koriste postojeće komponente iz nekog skladišta, na koji način treba primeniti standarde za kodiranje, dokumentaciju i dizajn?
11. Kako se može dokumentovati kontrola toka za objektno orijentisani program?

8 Testiranje programa



Pošto iskodirate svoje programske komponente, vreme je da ih testirate. Postoji više vrsta testiranja, a ovo poglavlje, kao i sledeće, predstaviće vam nekoliko pristupa testiranju koji obezbeđuju da vašim kupcima isporučite kvalitetne sisteme. Testiranje nije prva prilika za otkrivanje grešaka; videli smo kako inspekcije zahteva i dizajna pomažu da se problemi uoče u ranim fazama razvoja. Međutim, srž testiranja je otkrivanje grešaka, a postoje mnogi načini da ono bude uspešnije i delotvornije. U ovom poglavlju posmatramo pojedinačne komponente i njihovu integraciju da bismo testirali interfejse. U poglavlju 9 pažnju usmeravamo na tehnike za ocenjivanje sistema kao celine.

8.1 GREŠKE I SOFTVERSKI OTKAZI

Idealno bi bilo da kao programeri postanemo toliko dobri u svom poslu da svaki program koji proizvedemo pravilno radi kad god se pokrene. Nažalost, taj ideal je teško ostvariti. Za to postoji više razloga. Prvo, mnogi softverski sistemi obrađuju veliki broj stanja i složenih formula, aktivnosti i algoritama. Osim toga, za implementiranje kupčeve zamisli sistema koristimo alate koje imamo na raspolaganju, a ni kupac nije sasvim siguran šta mu treba. Konačno, složenost potiče i od veličine projekta i broja učesnika u njegovoj izradi. Prema tome, prisustvo grešaka ne potiče samo od softvera već i od očekivanja korisnika i kupaca.

Šta se smatra otkazom softvera? Obično je to situacija kada softver ne radi ono što je opisano u zahtevima. Na primer, specifikacija kaže da sistem treba da odgovori na neko konkretno pitanje samo u slučaju kada je korisnik ovlašćen da vidi taj podatak. Ako si-

stem odgovori neovlašćenoj osobi, kažemo da je sistem otkazao. Otkaz može da nastane kao rezultat više uzroka:

- Možda je specifikacija pogrešna ili u njoj nedostaje neki zahtev. U specifikaciji možda nije tačno navedeno šta je kupac tražio i šta mu je potrebno. U našem primeru, kupac možda u stvari želi nekoliko nivoa ovlašćenja, svako sa svojom vrstom pristupa ali tu potrebu nije eksplicitno iskazao.
- U specifikaciji možda postoji zahtev koji ne može da se implementira sa propisanim hardverom i softverom.
- Možda postoji greška u dizajnu sistema. Možda dizajn baze podataka i jezika za upite ne omogućavaju proveru autentičnosti korisnika.
- Možda postoji greška u programskom kodu. Možda je algoritam nepravilno ili nepotpuno implementiran.

Prema tome, otkaz je rezultat jedne greške ili više grešaka u nekom od aspekata sistema.

Bez obzira na to koliko smo vešti u pisanju programa, iz raznolikosti mogućih otkaza jasno je da treba proveriti da li su naše komponente ispravno kodirane. Mnogi programeri posmatraju testiranje kao dokaz da njihovi programi ispravno rade. Međutim, svrha testiranja je upravo suprotna. Mi programe testiramo da bismo dokazali postojanje grešaka. Pošto nam je cilj da otkrijemo greške, test smatramo uspešnim samo ako se greška otkrije ili ako dođe do otkaza u toku testiranja. **Prepoznavanje grešaka** je postupak utvrđivanja zbog koje greške, iligrešaka, je došlo do otkaza, a **ispravljanje grešaka** ili **otklanjanje grešaka** je postupak unošenja izmena u sistem u cilju otklanjanja grešaka.

Kada smo kodirali programske komponente i kada pristupamo njihovom testiranju, očekujemo da su specifikacije ispravne. Štaviše, ako smo koristili tehnike softverskog inženjerstva opisane u prethodnim poglavljima, trudili smo se da obezbedimo da dizajn sistema i pojedinačnih komponenti odražava zahteve i predstavlja osnovu za valjanu implementaciju. Međutim, faze u ciklusu razvoja softvera ne angažuju samo našu veštinu u radu sa računarom, već i našu veštinu da komuniciramo i sarađujemo. Sasvim je moguće da greška u softveru potiče od nesporazuma tokom neke od prethodnih faza razvoja.

Važno je imati na umu da se greške u sklopu softvera razlikuju od hardverskih grešaka. Mostovi, zgrade i drugi građevinski objekti mogu da propadnu zbog nekvalitetnog materijala, lošeg dizajna ili zato što su se njihove komponente pohabale. Ali petlje se neće pohabati ni posle 100 ponavljanja, a argumenti neće ispadati kada se prenose iz jedne komponente u drugu. Ako određeni deo koda ne funkcioniše ispravno, a izvor problema nije neki misteriozni otkaz hardvera, možemo biti prilično sigurni da postoji greška u kodu. Zbog toga mnogi softverski inženjeri ne vole da se greške softvera nazivaju „bubicama", jer bi to značilo da se greška došetala u kôd iz nekog spoljnog izvora nad kojim projektanti nemaju nikakvu kontrolu. Prilikom izrade softvera, u cilju kontrole kvaliteta napisanog koda, koristimo veštine iz domena softverskog inženjerstva.

U prethodnim poglavljima ispitali smo mnoge veštine koje pomažu da se broj grešaka u fazi izrade specifikacija i prilikom projektovanja svede na minimum. U ovom poglavlju razmatramo tehnike koje obezbeđuju smanjivanje broja grešaka u samom programskom kodu.

Vrste grešaka

Nakon kodiranja programskih komponenti, obično pregledamo kôd da bismo uočili greške i odmah ih otklonili. Kada nema očiglednih grešaka, program testiramo i pokušavamo da pronađemo još neku grešku, stvarajući uslove u kojima kôd ne reaguje na planirani način. Zato je važno da znamo koje vrste grešaka tražimo.

Algoritamska greška se javlja kada algoritam komponente, ili njegova logika, ne proizvode ispravan rezultat za dati ulaz, zato što nešto nije u redu sa postupkom obrade. Te greške se ponekad lako otkrivaju čitanjem programa (to nazivamo **proverom na stolu**), ili podnošenjem ulaznih podataka iz svih različitih klasa podataka za koje se očekuje da se mogu pojaviti pri normalnom radu programa. Uobičajene algoritamske greške mogu biti:

- grananje pre vremena;
- prekasno grananje;
- ispitivanje pogrešnog uslova;
- nedodeljivanje početne vrednosti promenljivim ili podešavanje invarijanti petlje;
- propust pri ispitivanju konkretnog uslova (na primer, kada može nastupiti deljenje nulom);
- poređenje promenljivih neodgovarajućeg tipa.

Kada proveravamo algoritamske greške, možemo takođe da tražimo i **sintaksne greške**. Tu proveravamo da li smo pravilno upotrebili iskaze programskog jezika. Ponekad naizgled trivijalna greška može da dovede do katastrofalnih rezultata. Na primer, Meyers (1976) ističe da prvi let SAD na Veneru nije uspeo zbog toga što je nedostajao zarez u jednoj fortranskoj petlji „do". Srećom, prevodioci otkrivaju mnoge sintaksne greške.

Greške izračunavanja i **greške preciznosti** nastupaju kada se formula pogrešno implementira ili kada ne izračunava rezultat sa zahtevanom preciznošću. Na primer, kada se u jednom izrazu kombinuju celobrojne promenljive sa promenljivima u predstavi sa pokretnim zarezom, može doći do neočekivanih rezultata. Neprikladno korišćenje podatka u predstavi sa pokretnim zarezom, neočekivano odsecanje ili neprikladan redosled operacija ponekad može proizvesti rezultat sa neprihvatljivom preciznošću.

Kada dokumentacija ne odgovara onome što program zaista radi, kažemo da imamo **grešku dokumentacije.** Često se dokumentacija pravi na osnovu dizajna programa i daje veoma jasan opis onoga što je programer želeo da program radi, ali implementacija tih funkcija nije ispravna. Takve greške mogu da dovedu do progresivnog širenja i indukovanja novih grešaka tokom životnog ciklusa programa, jer je većina sklonija da ima poverenje u dokumentaciju kada analizira kôd radi izmena.

Obično se u specifikaciji zahteva da se detaljno navodi broj korisnika u sistemu, kao i potrebe za njihovom komunikacijom. Na bazi tih informacija projektant često podešava svojstva sistema na opterećenje, koje ne prelazi ono koje je specifikovano kao maksimalno u sklopu zahteva. Ta svojstva se dalje prenose kroz dizajn programa kao ograničenja dužine redova čekanja, veličine bafera, dimenzija tabela i tako dalje. **Stresne greške** ili **greške preopterećenja** nastupaju kada se te strukture podataka popune preko navedenog kapaciteta.

Slično tome, **greške kapaciteta** ili **greške ograničenja** nastupaju kada performanse sistema postaju neprihvatljive kako se njegove aktivnosti približavaju specifikovanim granicama. Na primer, ako je u zahtevima navedeno da sistem mora da opslužI 32 uređaja, programi moraju da budu testirani u cilju nadgledanja performanse sistema u uslovima kada su svi oni aktivni. Štaviše, ako je takva konfiguracija moguća, sistem bi trebalo testirati kada je aktivno i više od 32 uređaja, da bi se utvrdilo šta se tada dešava. Testiranjem i dokumentovanjem reakcija sistema na opterećenja koja su iznad specifikovanih, tim za testiranje može pomoći timu za održavanje u prognoziranju posledica ako se kapacitet sistema poveća u budućnosti. Stanje kapaciteta bi trebalo ispitati i sa aspekta broja pristupa diskovima, broja prekida, broja zadataka koji se istovremeno izvršavaju i sličnih sistemskih mera.

Kada se razvija sistem koji radi u realnom vremenu, kritično je pitanje koordinacije više procesa koji se izvršavaju istovremeno, ili u precizno definisanom redosledu. **Greške vremenskog rasporeda** ili **greške koordinacije** nastaju kada je kôd koji služi za koordinaciju tih događaja neodgovarajući. Postoje dva razloga zbog kojih se ove greške teško identifikuju i ispravljaju. Prvo, projektantima i programerima nije lako da predvide sva moguća stanja sistema. Drugo, zbog velikog broja činilaca koji utiču na vremensko raspoređivanje i obradu, ponekad je, nakon pojave greške, nemoguće ponoviti iste uslove.

Greške propusnosti ili **greške performanse** nastupaju kada sistem ne omogućava performansu i brzinu specifikovanu u zahtevima. To su vremenski problemi druge vrste: vremenska ograničenja performanse sistema koja su postavljena na zahtev kupca, a ne zbog potreba koordinacije.

Kao što smo videli tokom projektovanja i programiranja, velika pažnja se poklanja pripremama sistema za oporavak nakon različitih otkaza. **Greške oporavka** mogu da nastanu kada nastupi greška, a ponašanje sistema ne odgovara željama projektanata niti zahtevima kupca. Na primer, ako dođe do nestanka napajanja u toku obrade, sistem bi trebalo da se oporavi na prihvatljiv način, na primer, da se vrati stanje datoteka neposredno pre otkaza. Kod nekih sistema oporavak može da podrazumeva nastavak rada punim kapacitetom uz oslonac na rezervni izvor napajanja; kod drugih, oporavak znači da sistem evidentira tok transakcija, što mu omogućava da nastavi rad kada se napajanje ponovo uspostavi.

Kod mnogih sistema je jedan deo hardvera i pripadajućeg sistemskog softvera propisan u sklopu zahteva a komponente se projektuju u skladu sa specifikacijama ponovno iskoristivih ili kupljenih programa. Na primer, ako se za komunikaciju koristi propisani modem, upravljački program modema generiše komande koje modem očekuje i čita komande dobijene od njega. **Hardverske greške i greške sistemskog softvera** mogu nastupiti kada isporučeni hardver i sistemski softver ne funkcionišu u skladu sa dokumentovanim operativnim uslovima i procedurama.

Konačno, potrebno je utvrditi da li su u kodu poštovani standardi i procedure propisane u okviru organizacije. **Greške standarda i procedura** ne utiču uvek na rad programa ali stvaraju uslove za nastanak grešaka prilikom testiranja i održavanja sistema. Ako se ne pridržava propisanih standarda, programer otežava drugima da razumeju logiku njegovog koda ili da pronađu podatke neophodne za rešavanje problema.

Ortogonalna klasifikacija kvarova

Korisno je da se vrste grešaka pronađenih ne samo u kodu već bilo gde u softverskom sistemu evidentiraju i kategorišu. Istorijske informacije mogu da pomognu u predviđanju vrsta grešaka koje bi naš kôd mogao da sadrži (što pomaže pri usmeravanju procesa testiranja), a nagomilavanje grešaka određenog tipa može predstavljati upozorenje da je vreme da se ponovo razmisli o dizajnu pa čak i o zahtevima. Mnoge organizacije vrše statističko modelovanje grešaka i analizu uzroka na bazi broja i raspodele tipova grešaka. Na primer, u IBM-ovom procesu za prevenciju kvarova (Mays i dr. 1990) istražuju se i dokumentuju osnovni uzroci svih problema koji se pojave; te informacije se koriste kao podloga za utvrđivanje očekivanih tipova grešaka u fazi testiranja, čime se smanjuje broj grešaka unetih u softver.

Chillarege i dr. (1992) u IBM-u razvili su pristup praćenju grešaka pod nazivom **ortogonalna klasifikacija grešaka**, gde se greške raspoređuju u kategorije koje zajedno daju sliku o tome na koje delove softvera treba da se obrati pažnja, jer se javljaju kao izvori više grešaka. Ta šema klasifikacije ne sme da zavisi od organizacije niti od projekta, i mora biti primenjiva na sve faze razvoja. U tabeli 8.1 navedene su vrste grešaka koje čine IBM-ovu klasifikaciju. Kada primenjuju ovu klasifikaciju, projektanti moraju da odrede tip greške, ali i da utvrde da li je reč o grešci činjenja ili o grešci nečinjenja. **Greška nečinjenja** je ona koja nastaje zato što je izostavljen neki ključni deo koda; na primer, greška može da nastupi zato što nekoj promenljivoj nije dodeljena početna vrednost. **Greška činjenja** je vezana za neispravnost; na primer, promenljivoj je dodeljena pogrešna početna vrednost.

Jedno od ključnih svojstava ortogonalne klasifikacije grešaka je njena ortogonalnost. To jest, klasifikaciona šema je **ortogonalna** ako svaka stavka koja se klasifikuje pripada tačno jednoj kategoriji. Drugim rečima, želimo da pratimo greške našeg sistema na nedvosmislen način, tako da zbirne informacije o broju grešaka u svakoj klasi budu od značaja. Značaj ovih merenja se gubi ako greška može da se razvrsta u više od jedne klase. Isto tako, klasifikacija grešaka mora da bude jasna, jer iz njene jasnoće treba da proizađe da bi bilo koja dva programera konkretnu grešku verovatno klasifikovala na isti način.

Tabela 8.1 IBM-ova ortogonalna klasifikacija grešaka

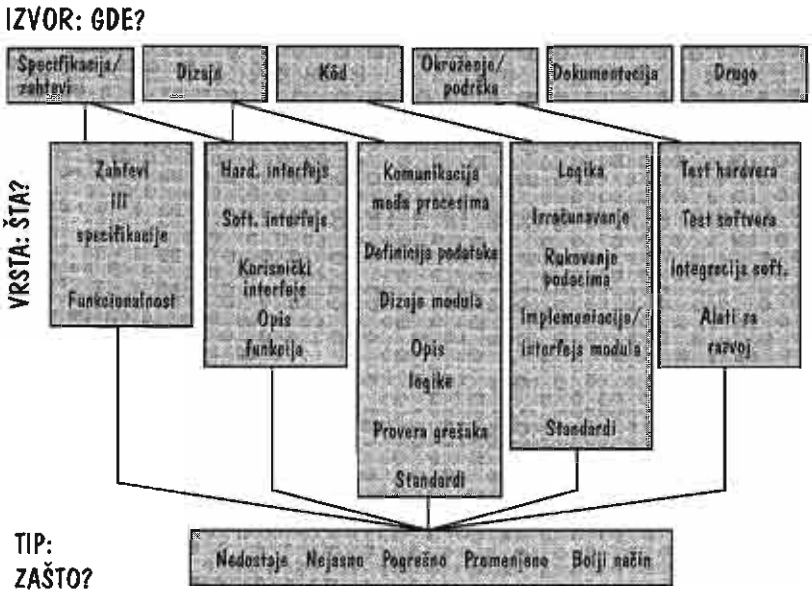
Tip greške	Značenje
Funkcija	Greška utiče na mogućnost, spregu sa korisnikom, sprege sa proizvodom, sprege sa hardverskom arhitekturom ili globalnom strukturom podataka.
Interfejs	Greška u međudejstvu sa drugim komponentama ili upravljačkim programima putem pozivanja, makroa, kontrolnih blokova ili listi parametara
Provera	Greška u logici programa koja ne uspeva pravilno da proveri valjanost podataka i njihove vrednosti pre njihove upotrebe.
Dodeljivanje	Greška u strukturi podataka ili u inicijalizaciji bloka koda
Vreme/serijalizacija	Greška koja utiče na vremenski raspored korišćenja deljenih resursa u i resursa realnog vremena
Gradnja/pakovanje/sjedinjavanje	Greška koja potiče od problema u skladištima, od promena upravljanja ili od kontrole verzija
Dokumentacija	Greška koja utiče na štampani materijal ili napomene vezane za održavanje
Algoritam	Greška koja se odnosi na efikasnost ili ispravnost algoritama ili strukture podataka ali ne i dizajna

PODSETNIK 8.1 KLASIFIKACIJA GREŠAKA FIRME HEWLETT-PACKARD

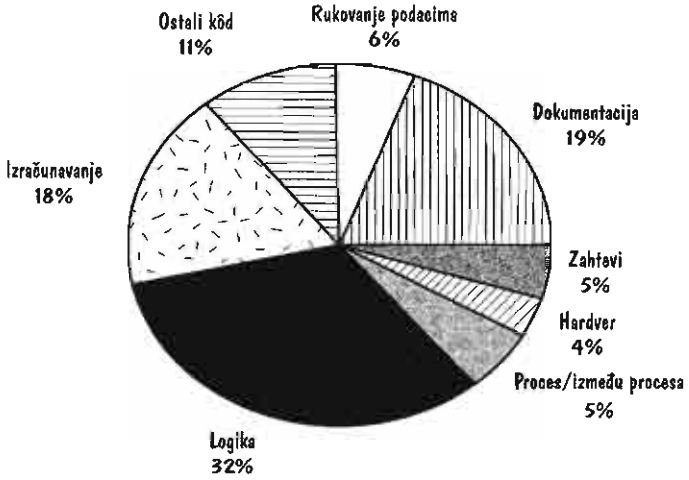
Grady (1997) opisuje princip klasifikacije grešaka u Hewlett-Packardu. Godine 1986. je savet za softversku metriku u Hewlett-Packardu utvrdio nekoliko kategorija za praćenje grešaka. Ta šema je rasla i sada izgleda kao što je prikazano na slici 8.1. Projektanti koriste taj model tako što za svaku otkrivenu grešku izaberu tri deskriptora: izvor greške (tj. gde je greška uneta u proizvod), vrstu greške, kao i tip (tj. da li je informacija nedostajala, da li je bila nejasna, pogrešna, promenjena ili se mogla uraditi na bolji način).

Svako odeljenje u Hewlett-Packardu zasebno prati svoje greške, a zbirni statistički podaci se predaju u vidu kružnih dijagrama, kao što je prikazano na slici 8.2. Različita odeljenja često imaju sasvim različite profile grešaka, a priroda profila pomaže projektantima u sledećim aktivnostima: formulisanje zahteva, dizajn, kodiranje i testiranje, sa posebnim naglaskom na vrsti grešaka koje se obično javljaju u tom odeljenju. Konačni efekat je bio da se vremenom smanjivao ukupan broj grešaka.

Klasifikacije grešaka kao što su ove IBM-ove i Hewlett-Packardove (podsetnik 8.1) pomažu unapređenju sveukupnog procesa razvoja tako što nam ukazuju na vrste grešaka koje su pronađene u pojedinačnim razvojnim aktivnostima. Na primer, za svaku tehniku prepoznavanja i proveravanja grešaka koja se koristi pri izgradnji jednog sistema možemo da napravimo profil vrsta pronađenih grešaka. Za očekivati je da će različiti metodi rezultirati različitim profilima. Na osnovu toga možemo izgraditi vlastitu strategiju za sprečavanje i otkrivanje grešaka, na osnovu vrsta grešaka koje možemo očekivati u našem sistemu i aktivnosti koje će ih ukloniti. Chillarege i dr. (1992) ilustruju IBM-ovo korišćenje tog koncepta iz kojeg se vidi da se profil grešaka nastao u fazi revizije dizajna značajno razlikuje od profila grešaka nastalih u fazi inspekcije koda.



SLIKA 8.1 Klasifikacija grešaka Hewlett-Packard (Grady 1997)



SLIKA 8.2 Greške za jedno odeljenje u okviru Hewlett-Packarda (Grady 1997)

8.2 PITANJA TESTIRANJA

Pre nego što kupcu predamo sistem za koji verujemo da ispravno funkcioniše, neophodno je obaviti više vrsta testiranja. Neki testovi zavise od onoga što se testira: komponente, grupe komponenti, podsistemi ili ceo sistem. Drugi testovi zavise od onoga što želimo da saznamo: da li sistem radi u skladu sa projektom? Zahtevima? Očekivanjima kupca? Razmotrimo neka od ovih pitanja.

Organizacija testiranja

Kada se razvija veliki sistem, testiranje obično zahteva više faza. Prvo se zasebno testira svaka programska komponenta, nezavisno od ostalih delova sistema. Takvim testiranjem, koje se naziva **testiranje modula** ili **jedinično testiranje**, proverava se da li pojedinačne komponente ispravno funkcionišu sa svim očekivanim tipovima ulaza, u skladu sa dizajnom komponente. Kad god je to moguće, jedinično testiranje se vrši u kontrolisanom okruženju tako da tim za testiranje može komponenti koja se testira da predaje unapred definisan skup podataka, i da posmatra izlazne akcije i rezultate. Osim toga, tim za testiranje proverava unutrašnju strukturu podataka, logiku i granične uslove za ulazne i izlazne podatke.

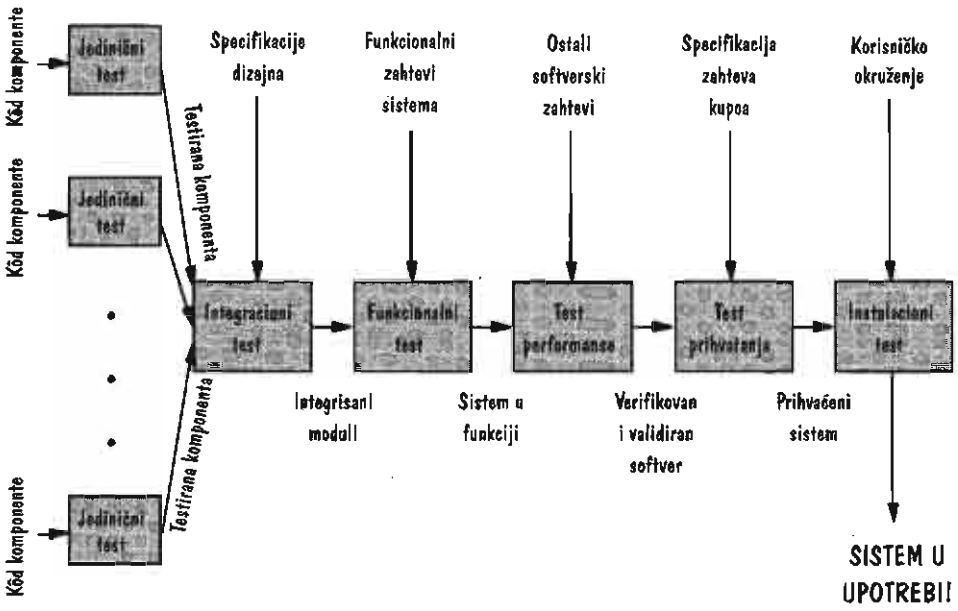
Kada se završi jedinično testiranje skupa komponenti, sledeći korak je proveravanje da li su interfejsi između komponenti pravilno definisani i realizovani. **Integraciono testiranje** je postupak proveravanja da li sistemske komponente sarađuju kao što je opisano u specifikacijama dizajna sistema i programa.

Pošto se uverimo da se između komponenti informacije prenose u skladu sa dizajnom, testiramo sistem da bismo proverili ima li on željenu funkcionalnost. **Funkcionalnim testiranjem** se proverava da li integrisani sistem zaista izvršava funkcije opisane u specifikaciji zahteva. Kao rezultat dobija se sistem koji funkcioniše.

Setite se da su zahtevi dokumentovani na dva načina: prvo u terminologiji kupca, a nakon toga u formi skupa softverskih i hardverskih zahteva namenjenih projektantima i programerima. Funkcionalni test predstavlja poređenje sistema koji se gradi sa funkcijama opisanim u specifikacijama zahteva kupca. Nakon toga se, u sklopu **testiranja performanse,** sistem poredi sa ostatkom hardverskih i softverskih zahteva. Kada se to testiranje uspešno obavi u stvarnom radnom okruženju kupca, dobija se **validiran sistem.**

Po okončanju testiranja performanse, projektanti i programeri su uvereni da sistem funkcioniše u skladu sa njihovim shvatanjem opisa sistema. Sledeći korak je sastanak sa kupcem na kojem proveravamo da li sistem funkcioniše u skladu sa očekivanjima kupca. Zajedno sa kupcem obavlja se **završni test prihvatanja**, u kojem se proverava usklađenost sistema sa opisom zahteva kupca. Kada se obavi završni test, prihvaćeni sistem se instalira u okruženje u kome će se koristiti; konačni **instalacioni test** potvrđuje da li sistem i dalje ispravno funkcioniše.

Na slici 8.3 ilustrovan je odnos između koraka testiranja. Bez obzira na veličinu sistema koji se testira, da bi se proverilo njegovo pravilno funkcionisanje, potrebne su sve opisane vrste testiranja. U ovom poglavlju se pre svega usredsređujemo na jedinično i na integraciono testiranje, u kojima se komponente testiraju same i zatim se spajaju u veći sistem koji radi. U devetom poglavlju razmatramo ostale korake u procesu testiranja, koji se zajedno zovu **testiranje sistema**. U tim kasnijim koracima sistem se posmatra i testira kao celina, a ne kao zasebni delovi.



SLIKA 8.3 Koraci testiranja

Odnos prema testiranju

Programeri početnici nisu navikli da posmatraju testiranje kao proces otkrivanja. Dok ste student, pišete programe prema specifikacijama koje dobijate od nastavnika. Po završenom dizajniranju programa, pišete kôd i prevodite ga da biste utvrdili postoje li neke sintaktičke greške. Kada predajete program na ocenjivanje, obično nastavniku predajete listing programa, podatke koje ste koristili kao probni ulaz i eventualni izlaz u kojem se vidi kako je vaš program obradio ulazne podatke. Kôd, ulaz i izlaz, zajedno, predstavljaju dokaz da vaš kôd pravilno radi, a obično birate ulazne podatke tako da ubedite predavača da kôd funkcioniše kao što je traženo u zadatku.

Svoj program ste verovatno posmatrali samo kao rešenje jednog problema: možda niste uzeli u obzir i sam problem. Ako je to slučaj, podatke za testiranje ste verovatno birali tako da potvrdite rezultate u specijalnim slučajevima, a ne da dokažete odsustvo grešaka. Tako napisan program služi kao dokaz vaše programerske veštine. Zbog toga kritiku programa psihološki doživljavate kao kritiku vaših sposobnosti. Testiranje kao dokaz ispravnosti programa služi da biste nastavniku pokazali svoje znanje.

Međutim, kada razvijate sistem za potrebe kupaca, njih ne zanima da li sistem pravilno funkcioniše u nekim situacijama. Njih daleko više zanima mogu li da budu sigurni da će sistem pravilno funkcionisati u svim situacijama. Zato bi vaš cilj kao projektanta-programera trebalo da bude da otkrijete što više grešaka bez obzira na to gde se greške nalaze i bez obzira na to ko ih je napravio. Za povređena osećanja i sujetu nema mesta kada se u procesu razvoja otkriju greške. Zbog toga su mnogi softverski inženjeri pristalice **bezličnog programiranja** gde se programi posmatraju kao komponente većeg sistema, a ne kao vlasništvo onih koji su ih napisali. Kada se otkrije greška ili kada dođe do otkaza, depersonalizovani razvojni tim se bavi ispravljanjem greške, a ne prebacivanjem krivice na konkretnog izvršioca.

Ko vrši testiranje?

Čak i kada se sistem razvija bezličnim pristupom, ponekad je teško potisnuti lična osećanja iz procesa testiranja. Zato se za testiranje sistema često koristi nezavisan tim za testiranje. Na taj način izbegavamo konflikt između vlastitog osećanja odgovornosti za greške i potrebe da se pronađe što više grešaka.

Osim toga, postoje dodatni argumenti u prilog nezavisnog tima. Prvo, moguće je da nenamerno unesemo greške prilikom tumačenja dizajna, određivanja programske logike, pisanja dokumentacije ili implementiranja algoritama. Jasno je da ne bismo dali program na testiranje kad bismo mislili da on ne funkcioniše u skladu sa specifikacijama. Ali možda nam je kôd previše blizak da bismo bili objektivni i prepoznali suptilnije greške.

Osim toga, nezavisni tim za testiranje može da učestvuje u pregledu komponente tokom celokupnog razvoja. Tim može da učestvuje u pregledima zahteva i dizajna, može pojedinačno da testira komponente koda a takođe i ceo sistem dok se on integriše i daje kupcima na prihvatanje. Na taj način, testiranje se može obavljati istovremeno sa kodiranjem; tim za testiranje može da testira i sklapa komponente čim se one završe, dok programeri za to vreme i dalje kodiraju preostale komponente.

Pogledi na predmet testiranja

Pre nego što pažljivo razmotrimo jedinično testiranje, pomenimo filozofiju na koju se testiranje oslanja. Kako testirate komponente, grupu komponenti, podsisteme ili sistem, vaš pogled na predmet testiranja (tj. komponentu, grupu, podsistem ili sistem) može da utiče na način izvođenja testa. Ako predmet testiranja posmatrate spolja kao **zatvorenu kutiju,** ili **crnu kutiju** sa nepoznatim sadržajem, vi prilikom testiranja ubacujete podatke u zatvorenu kutiju i beležite dobijene rezultate. U tom slučaju cilj testiranja jeste da se potvrdi da su prosleđeni svi tipovi ulaznih podataka i pri tome dobijeni očekivani rezultati.

Ova vrsta testiranja ima svoje prednosti i mane. Očigledna prednost je to što je zatvorena kutija oslobođena brige o ograničenjima koja nameće unutrašnja struktura i logika predmeta testiranja. Međutim, nije uvek moguće ceo test izvršiti na taj način. Na primer, pretpostavite da jedna jednostavna komponenta prihvata kao ulaz tri broja *a*, *b* i *c* i kao rezultat daje dva rešenja jednačine

$$ax^2 + bx + c = 0$$

ili poruku „nema realnih rešenja". Komponentu nije moguće testirati tako da joj se podnesu sve moguće uređene trojke (*a, b, c*). U tom slučaju će tim za testiranje možda izabrati reprezentativne podatke za testiranje da bi pokazao da se sve moguće kombinacije pravilno obrađuju. Na primer, mogu se izabrati podaci tako da imamo sve kombinacije pozitivnih, negativnih vrednosti i nule za *a*, *b* i *c* — dvadeset sedam mogućnosti. Ako znamo nešto o rešavanju kvadratnih jednačina, možda ćemo birati vrednosti takve da diskriminanta *b2 - 4ac* bude u svakoj od tri klase: pozitivna, negativna ili jednaka nuli. (U tom slučaju, mi u stvari nagađamo kako je komponenta implementirana.) Međutim, čak i kada test za sve razmatrane klase ne otkrije nijednu grešku, nemamo nikakvu garanciju da komponenta stvarno ne sadrži greške. Komponenta još uvek može u nekom konkretnom slučaju da dâ pogrešan rezultat, zbog finesa kao što su greške u zaokruživanju ili neusklađeni tipovi podataka.

Za neke predmete testiranja, testerski tim ne može da napravi skup reprezentativnih slučajeva koji dokazuju pravilno funkcionisanje u svim situacijama. Setite se iz poglavlja 7 one komponente koja je uzimala kao ulaz oporeziv bruto prihod, a kao izlaz davala iznos saveznog poreza na prihod. Mogli bismo da imamo tabelu poreza sa očekivanim rezultatima za određene date ulazne podatke, a da ne znamo kako se porez stvarno izračunava. Algoritam za izračunavanje poreza zavisi od poreskih kategorija, a granice kategorija i odgovarajući procenti spadaju u unutrašnju obradu komponente. Kad bismo tu komponentu posmatrali kao zatvorenu kutiju, ne bismo mogli izabrati reprezentativne testove, zbog nedovoljnog poznavanja načina obrade.

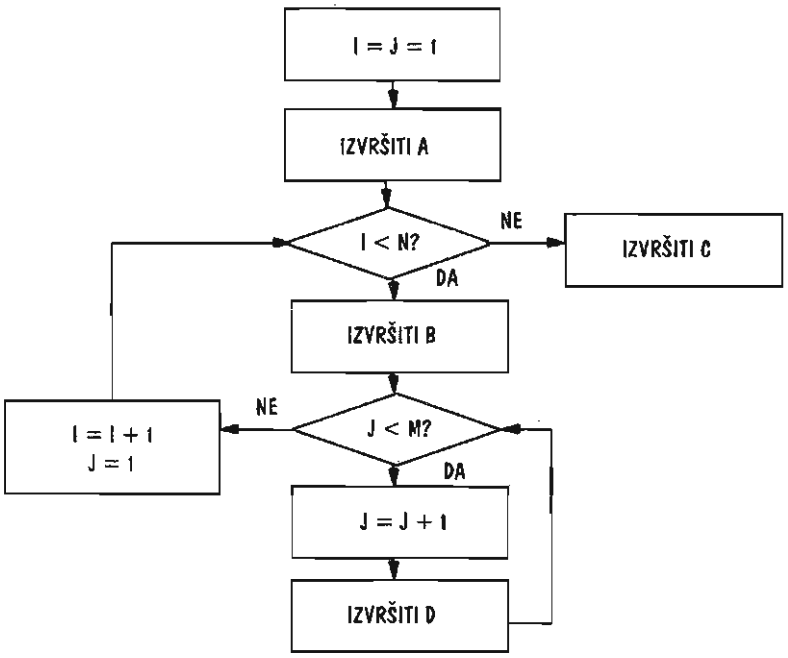
Za prevazilaženje ovog problema možemo posmatrati predmet testiranja kao **otvorenu kutiju** (koja se ponekad naziva i **belom kutijom**); tada možemo koristiti unutrašnju strukturu predmeta testiranja u cilju sprovođenja različitih testova. Na primer, možemo da smislimo testove tako da se izvrše sve naredbe ili svi tokovi kontrole unutar komponente, kako bismo se uverili da predmet testiranja ispravno radi. Međutim, kao što ćemo videti kasnije u ovom poglavlju, ta vrsta pristupa može da bude nepraktična.

Na primer, komponenta sa mnogo grananja i petlji sadrži veliki broj putanji koje treba proveriti. Čak i sa relativno jednostavnom logičkom strukturom, teško je temeljito testirati komponentu sa značajnim brojem iteracija ili rekurzija. Pretpostavite da je logika komponente tako postavljena da se petlja izvršava *nm* puta, kao što se vidi na slici 8.4.

Ako su i n i m svaki jednaki 100.000, primer koji se testira bi trebalo da se izvrši deset milijardi puta da bi prošao sve logičke putanje. Mogli bismo usvojiti strategiju testiranja u kojoj se petlja izvršava samo nekoliko puta i pri tome proveriti manji broj relevantnih slučajeva, koji će biti reprezentativni za ceo skup mogućnosti. U ovom primeru možemo izabrati jednu vrednost za I koja je manja od n, drugu vrednost koja je jednaka n i još jednu koja je veća od n; slično tome, možemo da ispitamo J koje je manje od m, jednako m i veće od m i njihove kombinacije sa kombinacijama vrednosti I. U opštem slučaju strategija može da se zasniva na podacima, strukturi, funkciji ili nekim drugim kriterijumima, kao što ćemo kasnije videti.

Prilikom donošenja odluke o načinu testiranja, ne moramo da biramo pristup otvorene ili zatvorene kutije. Testiranje po modelu zatvorene kutije možemo posmatrati kao jednu, a testiranje po modelu otvorene kutije kao drugu krajnost sveukupnog prostora testiranja. Sve filozofije testiranja padaju negde između te dve krajnosti. U podsetniku 8.2 vidimo kako se u nekim pristupima, kao što je pristup sa strukturom kutije, kombinuje više tačaka iz kontinuuma da bi se komponenta obradila iz više uglova. Uglavnom, izbor filozofije testiranja zavisi od više faktora, uključujući i:

- broj mogućih logičkih putanja;
- prirodu ulaznih podataka;
- količinu potrebnog izračunavanja;
- složenost algoritama.



SLIKA 8.4 Primer logičke strukture

PODSETNIK 8.2 STRUKTURE KUTIJA

Primena pristupa na bazi strukture kutija na slučaj informacionih sistema povezuje i proširuje pojmove otvorene i zatvorene kutije (Mills, Linger i Hevner 1987; Mills 1988). Ova tehnika počinje sa modelom crne kutije i kroz postupnu rafinaciju prelazi na model kutije stanja, a zatim na providne kutije.

Pogled po modelu *crne kutije* na neki objekat predstavlja u stvari opis njegovog spoljašnjeg ponašanja u svim mogućim uslovima. Objekat (komponenta, podsistem ili ceo sistem) se opisuje preko ulaznih dejstava i istorije ulaznih dejstava, to jest, evidencija o njegovim reakcijama na ranija ulazna dejstva. Na taj način se svaki odgovor može opisati prelazom:

(dejstvo, istorija_dejstava → odgovor)

Zatim se iz crne kutije, dodavanjem informacija o stanju, izvodi tzv. opis kutije-stanja. Svaki prelaz iz jednog u drugo stanje se opisuje kao:

(dejstvo, prethodno_stanje → odgovor, novo_stanje)

Na kraju model bele kutije uvodi opis procedure koja implementira kutiju-stanja, to jest, opisuje se kako se dejstvo i prethodno stanje transformišu u odgovor i novo stanje:

(dejstvo, prethodno_stanje → odgovor, novo_stanje) procedurom.

Procedura se opisuju uz oslonac na sekvence, izmene, iteracije i konkurencije. Postepeni prelazak od crne do providne kutije koristan je ne samo za testiranje već i za dizajn komponenti, jer pomaže da se opis višeg nivoa transformiše u mnogo pažljivije definisani opis dizajna nižeg nivoa.

8.3 JEDINIČNO TESTIRANJE

Kako treba da počnemo ako nam je cilj da pronađemo greške u komponentama? Postupak je sličan onome koji ste koristili kada ste na predavanjima testirali program. Prvo ispitate kôd tako što ga pročitate i tražite greške u algoritmu, podacima i sintaksi. Možete čak da uporedite kôd sa specifikacijama i dizajnom da biste se uverili da li ste uzeli u obzir sve relevantne slučajeve. Zatim prevodite kôd i uklanjate sve preostale sintaktičke greške. Na kraju, pravite slučajeve za testiranje da li se ulaz pravilno konvertuje u željeni izlaz. Jedinično testiranje ide tačno ovim putem, koji ćemo ispitati korak po korak.

Ispitivanje koda

Pošto vam opis dizajna pomaže da kodirate i dokumentujete svaku programsku komponentu, program odražava vaše tumačenje dizajna. Dokumentacija rečima i slikom opisu-

je šta bi programski kôd trebalo da radi. Zato je logično da se zatraži od objektivne grupe stručnjaka da uporede kôd sa dokumentacijom u cilju otkrivanja nesporazuma, nekonsistentnosti i drugih propusta. Ovakav **pregled koda,** sličan je postupcima pregledanja zahteva i dizajna, koji su opisani u prethodnim poglavljima. Obrazuje se tim koji se sastoji od programera-autora i tri ili četiri druga tehnička stručnjaka; tim analizira program i na organizovan način traži greške. Tehnički stručnjaci mogu da budu drugi programeri, dizajneri, tehnolozi ili kontrolori projekta. Dok je tim za pregled dizajna imao predstavnika kupca, tim za pregled koda ne sadrži člana iz organizacije kupca. Kupci iskazuju zahteve i odobravaju predloženi dizajn; njih implementacija zanima jedino pri demonstraciji usklađenosti funkcionisanja sistema kao celine sa njihovim formulacijama.

Prolasci kroz kôd. Postoje dve vrste pregleda koda: letimičan pregled i inspekcija. Kod **letimičnog pregleda**, programer prezentira kôd i prateću dokumentaciju timu za pregled, dok tim komentariše njihovu ispravnost. Tokom letimičnog pregleda programer vodi i kontroliše diskusiju. Atmosfera je nezvanična i rasprava usmerena na kôd, a ne na autora. Mada su nadzorni organi možda prisutni, letimičan pregled nema nikakvog uticaja na ocenu performanse autora, u skladu sa opštim ciljem testiranja: pronalaženje grešaka, ali, ne nužno, i njihovo ispravljanje.

Inspekcija koda. Inspekcija koda koju je prvobitno uveo Fagan (1976) u IBM-u slična je letimičnom pregledu, ali je zvaničnija. Prilikom **inspekcije,** tim za pregledanje proverava kôd i dokumentaciju u skladu sa prethodno pripremljenom listom pitanja. Na primer, tim može da ispita definicije i korišćenje tipova podataka i struktura kako bi proverio da li je njihovo korišćenje konsistentno sa dizajnom i sistemskim standardima, odnosno procedurama. Tim može da pregleda ispravnost i efikasnost algoritama i izračunavanja. Mogu se porediti komentari sa kodom da bi se proverilo da li su precizni i potpuni. Slično tome, može se proveravati ispravnost interfejsa komponenti. Tim čak može da proceni karakteristike performanse koda u smislu korišćenja memorije i brzine obrade i da priprema ocenu ispoštovanosti zahteva u pogledu performanse.

Inspekcija koda se obično vrši u nekoliko koraka. Prvo, tim može da se sastane kao grupa i obavi grub pregled koda i opiše ciljeve inspekcije. Zatim se članovi tima pojedinačno pripremaju za drugi sastanak grupe. Svaki učesnik u inspekciji proučava kôd i pripadajuću dokumentaciju i evidentira pronađene greške. Na kraju, na sastanku grupe, članovi tima izveštavaju o tome šta su pronašli, evidentirajući i greške koje se pronađu tokom rasprave o pojedinačnim nalazima. Ponekad se greška koju otkrije pojedinac tretira kao „lažni alarm": nešto što je izgledalo kao greška, ali što grupa ne smatra pravim problemom.

Članovi tima za inspekciju biraju se na osnovu ciljeva inspekcije, a ponekad jedan član tima može da obavlja više uloga. Na primer, ako je cilj inspekcije da se proveri da li su interfejsi ispravni, u tim bi trebalo uključiti projektante interfejsa. Pošto se inspekcija usredsređuje na cilj, sastanak ne vodi programer već moderator tima, pomoću skupa ključnih pitanja na koje treba odgovoriti. Kao i kod letimičnih pregleda, inspekcije se bave kodom, a ne autorom i rezultati nemaju uticaja na ocenu njegovih kvaliteta.

Uspeh pregleda koda. Možda vam nije prijatno da neki tim pregleda vaš kôd. Međutim, pregledi su se pokazali izuzetno uspešnim u otkrivanju grešaka i u mnogim organi-

zacijama spadaju u obavezne ili veoma poželjne postupke. Imajte na umu da što se ranije greška primeti, njeno ispravljanje će biti lakše i jeftinije. Bolje je da se problem otkrije na nivou komponente, nego tek kasnije u ciklusu testiranja kada je uzrok problema manje jasan. U stvari, zbog toga Gilb (1988) i Gilb i Graham (1993) predlažu inspekcije ne samo koda već i produkata ranih faza razvoja, kao što su specifikacije i dizajn.

Nekoliko istraživača ispitivalo je do koje mere su pregledi pronalazili greške. Fagan (1976) je izvršio eksperiment u kome je 67% od ukupnog broja grešaka pronađenih u sistemu nađeno pomoću inspekcija pre jediničnog testiranja. U Faganovoj studiji je druga grupa programera pisala slične programe i umesto inspekcija koristila letimične preglede. Kôd grupe sa inspekcijama pokazao je u toku prvih sedam meseci rada 38% manje grešaka nego kôd grupe koja je koristila letimične preglede. U drugom Faganovom eksperimentu, od ukupnog broja grešaka otkrivenih tokom razvoja sistema, 82% je pronađeno tokom inspekcija dizajna i koda. Rano otkrivanje grešaka dovelo je do uštede vremena projektanata i programera. I drugi istraživači izveštavaju o iskustvima u korišćenju inspekcija. Na primer, Ackerman, Buchwald i Lewski (1986) napominju da je u jednoj poslovnoj aplikaciji od 6.000 redova 93% svih grešaka pronađeno prilikom inspekcija.

Jones (1977) je opširno analizirao produktivnost programera, uključujući i prirodu grešaka i metode za njihovo pronalaženje i ispravljanje. Ispitivanjem istorije za 10 miliona redova koda, on je našao da su inspekcije koda uklonile čak 85% svih pronađenih grešaka. Nijedna druga tehnika koju je Jones proučavao nije bila toliko delotvorna; u stvari, nijedna nije mogla da ukloni čak ni polovinu pronađenih grešaka. U novijim istraživanjima Jones (1991) predlaže uobičajena vremena za pripremu i vremena za sastanke kao što je prikazano u tabeli 8.2.

Grady (1997) objašnjava da u Hewlett-Packardu planiranje inspekcije obično traje oko dva sata, nakon čega sledi polusatovni sastanak tima. Pojedinačne pripreme uključuju dva sata za pronalaženja grešaka i devedeset minuta za evidentiranje pojedinačnih nalaza. Tim provodi oko trideset minuta u živoj diskusiji o nalazima i preporučivanju aktivnosti koje treba da se preduzmu. Pošto se greške otklone, moderator provodi dodatnih pola sata na pisanju i dostavljanju zaključnog dokumenta. U podsetniku 8.3 opisuje se kako projektanti i programeri softvera u kompaniji Bull Information Systems ispituju načine za smanjenje resursa neophodnih za inspekciju, uz očuvanje efikasnosti inspekcije.

Tabela 8.2 Uobičajena vremena za pripremu inspekcije i sastanka (Jones 1991)

Proizvod	Trajanje pripreme	Trajanje sastanka
Dokument zahteva	25 strana na sat	12 strana na sat
Specifikacija funkcija	45 strana na sat	15 strana na sat
Specifikacija logike	50 strana na sat	20 strana na sat
Izvorni kôd	150 redova koda na sat	75 redova koda na sat
Korisnički dokumenti	35 strana na sat	20 strana na sat

PODSETNIK 8.3 OPTIMALNA VELIČINA TIMA ZA INSPEKCIJE

Weller (1993) je ispitao podatke za tri godine inspekcija u kompaniji Bull Information Systems. Merenja sa skoro 7 hiljada inspekcijskih sastanaka obuhvatala su informacije o 11.557 grešaka i 14.677 stranica dokumentacije dizajna. On je našao da troclani tim za inspekciju uz manju brzinu pripreme daje jednako dobre rezultate kao i četvoročlani tim sa većom brzinom rada; on smatra da efikasnost inspekcije zavisi od brzine pripreme, a ne od veličine tima. Našao je takođe da efikasnost i uspešnost tima zavise od toga koliko njegovi članovi poznaju proizvod: što su bolje upoznati, rezultati su im bolji.

S druge strane, Weller je našao da dobri rezultati inspekcije koda mogu da dovedu do pogrešnog osećaja sigurnosti. U jednom projektu sa 12 000 redova koda na jeziku C zahtevi i dizajn nisu revidirani; inspekcije su počele od koda. Međutim, zahtevi su evoluirali u fazi jediničnog i integracionog testiranja tako da je ceo kod skoro uništen. Poređenjem podataka iz faze inspekcije koda sa podacima iz faze testiranja, Weller je našao da su inspekcije koda otkrile uglavnom greške vezane za kodiranje ili dizajn niskog nivoa, dok su testiranjem otkrivene uglavnom greške u zahtevima i arhitekturi softvera. Otuda, pošto se inspekcija koda nije bavila stvarnim uzrocima menjanja sistema, njeni rezultati nisu predstavljali ocenu njegovog stvarnog kvaliteta.

Jones (1991) rezimira podatke iz svog obimnog skladišta informacija o projektima i daje drugačiju sliku o tome kako pregledi i inspekcije, u poređenju sa ostalim aktivnostima otkrivanja, pronalaze greške. Pošto se veličina proizvoda značajno razlikuje, u tabeli 8.3 prikazane su brzine otkrivanja grešaka u odnosu na broj hiljada redova koda u isporučenom proizvodu. Iz tabele je jasno da se inspekcijom koda pronalazi daleko više grešaka nego primenom većine ostalih tehnika. Međutim, istraživači i dalje ispituju da li neke vrste aktivnosti otkrivaju drugačije tipove grešaka. Na primer, inspekcije dobro otkrivaju greške u kodu dok izrada prototipa uspešnije pronalazi probleme iz domena zahteva.

Pošto je Fagan objavio svoje smernice za inspekciju koda u IBM-u, mnoge druge organizacije uključujući Hewlett-Packard (Grady i van Slack 1994), ITT i AT&T (Jones 1991) prihvatile su inspekcije kao preporučeni ili standardni postupak. U literaturi se i dalje mogu naći opisi uspešne primene inspekcija, od kojih su neki pomenuti i na veb lokaciji ove knjige.

Tabela 8.3 Greške pronađene tokom aktivnosti otkrivanja (Jones 1991)

Aktivnost otkrivanja	Broj otkrivenih grešaka na hiljadu linija koda
Pregled zahteva	2,5
Pregled dizajna	5,0
Inspekcija koda	10,0
Test integracije	3,0
Test prihvatanja	2,0

Dokazivanje ispravnosti kôda

Pretpostavite da je vaša komponenta kodirana, da ste je vi pregledali, a tim ispitao. Sledeći korak testiranja jeste da se kôd podvrgne analizi na strukturisan način kako bi se utvrdila njegova ispravnost. S aspekta jediničnog testiranja, program je **ispravan** ako implementira funkcije i podatke kao što je zahtevano u dizajnu i ako ima ispravan interfejs prema ostalim komponentama.

Jedan od načina da se ispita ispravnost programa jeste da se kôd posmatra kao iskaz logičkog toka. Ako možemo ponovo da napišemo program pomoću nekog formalnog logičkog sistema (kao što je niz iskaza i pretpostavki o podacima) tada možemo da testiramo ispravnost te nove predstave. Ispravnost tumačimo u smislu dizajna i zahtevamo da naši izrazi poštuju pravila matematičke logike. Na primer, ako možemo da formulišemo program kao skup aksioma i teorema, moći ćemo da pokažemo da dokaz teorema podrazumeva ispravnost koda.

Tehnike formalnog dokazivanja. Pogledajmo kako izgleda formalno dokazivanje. Kôd se kroz niz koraka konvertuje u svoj logički ekvivalent:

1. Prvo ispisujemo aksiome koji opisuju ulazne i izlazne uslove komponente. Ovi iskazi su kombinacija logičkih promenljivih (od kojih je svaka tačna ili netačna) povezanih logičkim simbolima za operacije koji su prikazani u tabeli 8.4. Na primer, pretpostavite da komponenta na ulazu prihvata niz T veličine N. Na izlazu, komponenta daje ekvivalentni niz T' koji se sastoji od elemenata niza T poređanih u rastućem redosledu. Ulazne uslove možemo da napišemo u obliku aksioma:

$$A_1: (T \text{ je niz}) \ \& \ (T \text{ je veličine } N)$$

Slično tome, možemo da napišemo izlaz kao aksiom:

$$A_{kraj}: (T' \text{ je niz}) \ \& \ (\forall\, i \text{ ako je } i < N \text{ tada je } (T'(i) \le T'(i+1))$$
$$\& \ (\forall\, i \text{ ako je } i \le N \text{ tada } \exists\, j(T'(i) = T(j)) \ \& \ (T' \text{ je veličine } N)$$

Tabela 8.4 Logičke operacije

Operacija	Primer	Značenje
Konjunkcija	$x \& y$	x i y
Disjunkcija	$x \wedge y$	x ili y
Negacija	$\neg x$	nije x
Implikacija	$x \rightarrow y$	ako je x tada je y
Ekvivalencija	$x = y$	x je jednako y
Univerzalni kvantifikator	$\forall\, x\, P(x)$	za svako x uslov $P(x)$ je tačan
Egzistencijalni kvantifikator	$\exists\, x\, P(x)$	postoji x za koje je uslov $P(x)$ tačan

2. Zatim, crtamo dijagram toka koji prikazuje logički sled u sklopu komponente. Na dijagramu obeležavamo tačke u kojima dolazi do transformacije.

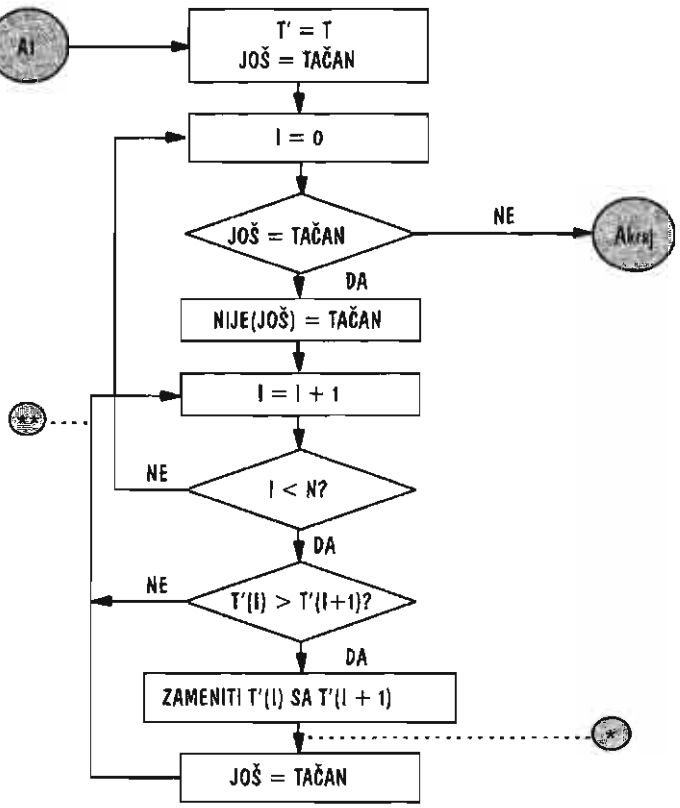
Na slici 8.5 prikazan je takav dijagram za komponentu gde se za sortiranje niza T po rastućem redosledu koristi sortiranje postupkom mehurića. Na slici su istaknute dve

tačke u kojima dolazi do transformacije. Tačka označena jednom zvezdicom može se izraziti u formi aksioma na sledeći način:

$$[(\text{nije(još)} = \text{tačan})) \ \& \ (i < N) \ \& \ (T'(i) > T'(i+1))]$$
$$\rightarrow [(T'(i)) \text{ zamenjuje se sa } T'(i+1)]$$

Slično tome, tačka označena sa dve zvezdice može se napisati kao:

$$[(\text{nije(još)} = \text{tačan})) \ \& \ (i \geq N)] \rightarrow [(T'(i)) \text{ je sortiran}]$$



SLIKA 8.5 Dijagram toka za sortiranje niza

3. Na osnovu aksioma formulišemo niz teorema koje treba dokazati. Počevši od prvog aksioma, idemo od jedne do druge transformacije i uveravamo se da ako je tačno prvo, onda je tačno i sledeće. Drugim rečima, ako je prva tvrdnja A_1 a prva tačka transformacije A_2, onda je naša prva teorema:

$$A_1 \rightarrow A_2$$

Ako je A_3 sledeća tačka transformacije, druga teorema će biti:

$$A_2 \rightarrow A_3$$

Na taj način izražavamo sve teoreme kao:

$$A_i \rightarrow A_j$$

gde su A_i i A_j susedne transformacije u dijagramu toka. Poslednja teorema tvrdi da, ako imamo stanje „tačan" u tački poslednje transformacije, znači da je tačna izlazna tvrdnja:

$$A_k \rightarrow A_{kraj}$$

U drugoj varijanti, možemo da pođemo od A_{kraj} i tražimo prethodnu tačku transformacije. Prvo dokazujemo da je:

$$A_k \rightarrow A_{kraj}$$

a zatim da je:

$$A_i \rightarrow A_j$$

i tako dalje sve do:

$$A_1 \rightarrow A_2$$

U svakom slučaju je rezultat isti.

4. Zatim, tražimo petlje u dijagramu toka i za svaku od njih odredimo tvrdnje „ako-onda".
5. Na taj način smo pronašli sve tvrdnje. Da bismo dokazali da je program ispravan, pronalazimo sve putanje koje počinju sa A_1 i završavaju se sa A_{kraj}. Po svakoj od ovih putanja, idemo putem kojim kôd dokazuje da istinita početna tvrdnja daje istinitu izlaznu tvrdnju.
6. Kada pronađemo sve putanje, moramo da proverimo istinitost svake od njih strogim dokazivanjem da ulazna tvrdnja podrazumeva izlaznu tvrdnju u skladu sa transformacijama na toj putanji.
7. Na kraju dokazujemo da se program završava.

Prednosti i mane dokazivanja ispravnosti. Kada automatski ili ručno gradimo dokaz koji je gore opisan, možemo otkriti greške u algoritmu ili kodu. Osim toga, tehnika dokazivanja obezbeđuje formalno razumevanje programa, pošto analiziramo njegovu logičku strukturu. Redovno korišćenje ovog metoda primorava nas da rigoroznije i preciznije definišemo podatke, strukture podataka i pravila algoritama.

Međutim, takva strogost ima svoju cenu. Za pripremanje i dokazivanje treba utrošiti mnogo rada. Na primer, kôd za komponentu dobijenu mehuranjem je mnogo kraći od njegovog logičkog opisa i dokaza. U mnogim slučajevima, za dokazivanje ispravnosti koda potrebno je više vremena nego za samo kodiranje. Štaviše, veće i složenije komponente mogu da proizvedu ogromne dijagrame toka, mnoge transformacije i proveravanje velikog broja putanja. Na primer, nenumerički programi se teže logički prikazuju od numeričkih. Paralelna obrada se teško obrađuje, a kompleksne strukture podataka ponekad proizvode veoma složene transformacione izraze.

Obratite pažnju na to da se tehnika dokazivanja zasniva samo na načinu na koji se prema pravilima logike ulazne tvrdnje transformišu u izlazne. Kada dokažemo da je program ispravan u logičkom smislu, to još uvek ne dokazuje odsustvo softverskih grešaka. Ova tehnika ne može da otkrije greške u dizajnu, interfejsima sa drugim komponentama, tumačenju specifikacija, sintaksi i semantici programskog jezika niti dokumentaciji.

Na kraju, moramo da priznamo da nisu svi dokazi ispravni. Više puta se u istoriji matematike dogodilo da se za dokaz koji je godinama bio prihvaćen, kasnije dokaže da je pogrešan. Uvek je moguće da neki posebno komplikovan aksiom u sklopu dokaza ne bude tačan.

Druge tehnike dokazivanja. Tehnika logičkog dokazivanja zanemaruje strukturu i sintaksu programskog jezika na kome je implementiran testirani program. Na neki način, ovom tehnikom dokazujemo da je korektan dizajn komponente, ali ne obavezno i njena implementacija. Postoje druge tehnike koje uzimaju u obzir karakteristike jezika.

Jedna takva tehnika, **simboličko izvršavanje**, uključuje simulirano izvršavanje koda pri čemu se umesto promenljivih koriste simboli. Testirani program se posmatra tako da mu je ulazno stanje određeno ulaznim podacima i uslovima. Kako se izvršava red po red koda, proverava se da li je došlo do promene stanja. Svaka promena stanja se čuva, a izvršavanje programa posmatra se kao niz promena stanja. Na taj način, svaka logička putanja u programu odgovara nizu uređenih promena stanja. Konačno stanje svake putanje trebalo bi da bude jedno izlazno stanje, a program je korektan ako svako moguće ulazno stanje proizvodi pravilno izlazno stanje.

Pogledaćemo jedan primer da bismo videli kako funkcioniše simboličko izvršavanje. Uzmimo da testiramo sledeće redove u programu:

```
a = b + c;
if (a > d) taskx(); // IZVRŠITI TASKX
else tasky;   // IZVRŠITI TASKY
```

Alat za simboličko izvršavanje će zabeležiti da uslov, $a > d$, može da bude ili tačan ili netačan. Dok bi uobičajeno izvršavanje koda uključilo konkretne vrednosti za a i d, alatka za simboličko izvršavanje beleži dva moguća stanja: $a > d$ je netačno i $a > d$ je tačno. Umesto da se testira veliki broj mogućih vrednosti za a i d, u simboličkom izvršavanju se posmatraju samo ta dva slučaja. Na ovaj način se veliki skupovi podataka dele na disjunktne ekvivalentne klase ili kategorije, a kôd može da se razmatra samo u odnosu na to kako reaguje na svaku kategoriju podataka. Pošto se razmatraju samo klase ekvivalentnih podataka, koje se predstavljaju kao simboli, značajno je smanjen broj slučajeva potrebnih za dokazivanje.

Međutim, ova tehnika poseduje većinu nedostataka karakterističnih i za logičko dokazivanje teorema. Razvijanje dokaza može da traje duže od pisanja koda, a dokaz ispravnosti ne znači i odsustvo grešaka. Štaviše, tehnika se oslanja na pažljivo praćenje promena stanja po putanjama programa. Mada ova tehnika može donekle da se automatizuje, obiman ili složen kôd ipak zahteva da se proveri veliki broj stanja i putanja, što traje dugo. Alati za automatizovano simboličko izvršavanje teško prate tok izvršavanja petlji. Osim toga, kad god se u kodu koriste indeksi i pokazivači, podela na ekvivalentne klase postaje teža.

Automatizovano dokazivanje teorema. Neki softverski inženjeri su pokušali da automatizuju postupak dokazivanja ispravnosti programa tako što su razvili alate koji kao ulaz prihvataju:

- ulazne podatke i uslove,
- izlazne podatke i uslove,
- redovi koda komponente koja se testira.

Izlaz iz automatizovanog alata je ili dokaz da je komponenta ispravna, ili kontraprimer sa skupom podataka za koje je komponenta nekorektno transformisala ulaz u izlaz. Automatizovani dokazivač teorema sadrži informacije o jeziku na kome je komponenta napisana pa su mu dostupna sintaktička i semantička pravila. Prateći korake programa, dokazivač teorema prepoznaje putanje na nekoliko načina. Ako su uobičajena inferencionа i deduktivna pravila previše zbunjujuća, ponekad se umesto njih koristi iskustveno rešenje.

Takav softver za dokazivanje teorema nije trivijalan za izradu. Na primer, alat mora da bude u stanju da proveri pravilnu upotrebu unarnih i binarnih operacija, (na primer, sabiranja, oduzimanja i negacije) kao i poređenja u kojima se javlja jednakost i nejednakost. Moraju se ugraditi i složenija pravila, kao što su komutativnost, distributivnost i asocijativnost. Veoma je teško izraziti programski jezik kao skup postulata od kojih se izvode teoreme.

Pretpostavimo da se te teškoće prevaziđu. Za sve, osim za najtrivijalnije komponente, izgradnja teorema metodom pokušaja i promašaja zahteva prevelik utrošak vremena. Zbog toga je za usmeravanje dokazivača teorema poželjna izvesna ljudska saradnja. Pomoću metoda koje se često koriste u izgradnji ekspertskih sistema, interaktivni dokazivač teorema može da sarađuje sa svojim korisnikom pri izboru tačaka transformacije i praćenju putanja. Na taj način, dokazivač teorema u stvari ne generiše dokaz; umesto toga, on proverava dokaz koji je skicirao korisnik. Razvijeni su alati za simboličko izvršavanje kojima se analizira kôd malih programa ali ne postoji automatizovan sistem opšte namene za simboličko izvršavanje koji ne zavisi od jezika.

Može li se izgraditi idealan dokazivač teorema, uz pretpostavku da postoji dovoljno brza mašina i implementacioni jezik koji može da obradi svu složenost problema? Idealan dokazivač teorema bi učitao bilo koji program i kao rezultat dao iskaz o ispravnosti kôda ili prijavu mesta na kojem se nalazi greška. Dokazivač teorema bi morao da utvrdi da li se neka naredba u kodu izvršava za neki ulazni podatak. Nažalost, takav dokazivač teorema nije moguće izgraditi. Moguće je dokazati (na primer kod Pfleegera i Straighta [1985]) da je izgradnja takvog programa ekvivalentna problemu zaustavljanja Tjuringove mašine. Problem zaustavljanja nije rešiv, a to znači ne samo da za taj problem nema rešenja već takođe da je rešenje nemoguće pronaći. Dokazivač teorema bi mogao da se reši samo za slučaj kada program ne sadrži nikakva grananja, ali zbog takvog ograničenja on bi bio primenljiv samo na veoma uskom podskupu svih programa. Prema tome, mada bi bio veoma poželjan, svaki automatizovani dokazivač teorema je samo aproksimacija idealnog.

Testiranje programskih komponenti

Softverski inženjeri imaju za cilj da dokažu kako je kôd ispravan; zbog toga je rađeno mnogo istraživanja sa ciljem razvoja metoda i automatizovanih alata. Međutim, u bliskoj

budućnosti će razvojni timovi verovatno biti zainteresovaniji za testiranje softvera nego za dokazivanje ispravnosti programa.

Poređenje testiranja i dokazivanja. Pri dokazivanju ispravnosti programa, tim za testiranje ili programer razmatraju jedino kôd i njegova ulazna i izlazna stanja. Program se posmatra samo u smislu klasa podataka i uslova opisanih u dizajnu. Zbog toga u dokazivanju nije obavezno izvršavanje koda, nego se samo razmatra šta se događa u programu.

Međutim, stanovište kupaca je drugačije. Da bismo njima dokazali da program ispravno radi, moramo pokazati kako se njegov rad manifestuje spolja. U tom smislu, testiranje se pretvara u niz eksperimenata na čijim se rezultatima zasniva ocena o ponašanju programa u datoj situaciji. Dok nam dokaz govori o radu programa u hipotetičkom okruženju opisanom u dizajnu i zahtevima, testiranje nam daje informaciju o tome kako program radi u njegovom stvarnom radnom okruženju.

Izvor slučajeva za testiranje. Za testiranje komponente biramo ulazne podatke i stanja, puštamo komponentu da manipuliše podacima i posmatramo rezultate. Ulaz biramo tako da bi nam izlaz manifestovao neko ponašanje definisano u kodu. **Tačka** ili **slučaj za testiranje** je konkretan izbor ulaznih podataka koji će se upotrebiti za testiranje programa. **Test** je konačna kolekcija slučajeva. Kako da izaberemo slučajeve i kako da definišemo testove da bismo se i mi i naši kupci uverili da program ispravno radi, ne samo za izabrane slučajeve već i za svaki ulaz?

Počinje se tako što se odrede ciljevi testa. Zatim se biraju slučajevi i definiše se test projektovan da ispuni konkretan cilj. Jedan cilj bi mogao da bude dokazivanje da se sve naredbe pravilno izvršavaju. Drugi cilj bi mogao da bude prikazivanje da se pravilno izvršava svaka funkcija koja se obavlja u kodu. Od ciljeva zavisi kako ćemo klasifikovati ulaz da bismo izabrali slučajeve.

Kôd možemo posmatrati bilo kao zatvorenu bilo kao otvorenu kutiju, zavisno od ciljeva testa. U slučaju zatvorene kutije, podnosimo joj sve vrste ulaza i poredimo izlaz sa onim što se očekuje u skladu sa zahtevima. Međutim, ako kôd posmatramo kao otvorenu kutiju, možemo da razmotrimo unutrašnju logiku koda i pažljivo primenimo strategiju testiranja.

Setite se primera komponente za izračunavanje korena kvadratne jednačine. Ako je cilj našeg testiranja da dokažemo kako kôd pravilno funkcioniše, mogli bismo birati slučajeve tako da koeficijenti a, b i c budu iz reprezentativnih kombinacija negativnih brojeva, pozitivnih brojeva i nule. Možemo takođe birati kombinacije na osnovu odnosa između koeficijenata:

- a je veće od b, a ono je veće od c
- b je veće od c, a ono je veće od a
- c je veće od b, a ono je veće od a

i tako dalje. Međutim, ako uzmemo u obzir unutrašnji rad koda, možemo da vidimo da logika zavisi od vrednosti diskriminante, $b2 - 4ac$. Tada bismo slučajeve birali tako da diskriminanta bude pozitivna, negativna i jednaka nuli. Možemo takođe da uključimo slučajeve nenumeričkih podataka. Na primer, možemo kao koeficijent da ponudimo slovo 'F' da bismo ispitali kako sistem reaguje na nenumeričku vrednost. Zajedno sa tri numerička slučaja, za test imamo četiri vrste ulaza koji se međusobno isključuju.

Na ovaj način, ciljeve testa koristimo za razdvajanje ulaza u ekvivalentne klase. To jest, klase podataka bi trebalo da zadovolje sledeće kriterijume:

1. Svaki mogući ulaz pripada jednoj od klasa. To jest, klase pokrivaju ceo skup ulaznih podataka.
2. Svaki ulazni podatak pripada samo jednoj klasi. To jest, klase su disjunktne.
3. Ako pri izvršavanju koda dođe do greške kada se kao ulaz koristi član konkretne klase, ista greška se mora javiti za bilo koji drugi element te klase. To jest, svaki element klase predstavlja sve ostale elemente te klase.

Ne može se uvek proceniti da li je moguće zadovoljiti treći uslov za klase. Treći uslov možemo da oslabimo tako da ako element podataka pripada jednoj klasi i on dovede do greške, velika je verovatnoća da će i svi ostali elementi te klase manifestovati istu grešku.

Kod testiranja metodom zatvorene kutije neizvesno je da li će biti izabran slučaj koji će otkriti konkretnu grešku. S druge strane, kod testiranja po modelu otvorene kutije uvek postoji opasnost preteranog vezivanja za unutrašnju obradu. Može se dogoditi da testiramo to što program radi, umesto da testiramo ono što bi on trebalo da radi.

Kada pripremamo test podatke, možemo da kombinujemo testiranje po modelu otvorene i zatvorene kutije. Prvo, posmatramo program kao zatvorenu kutiju i pravimo početne slučajeve na osnovu spoljnih specifikacija programa. Tu bi trebalo da obuhvatimo ne samo očekivane ulazne podatke, već i granične slučajeve za ulaz i izlaz kao i nekoliko primera neispravnih podataka. Na primer, ako je komponenta kodirana tako da očekuje pozitivne vrednosti na ulazu, mogli bismo za testiranje da izaberemo po jedno od sledećeg:

- jedan veoma veliki ceo pozitivan broj;
- jedan pozitivan ceo broj;
- jedan pozitivan decimalan broj sa fiksnim zarezom;
- jedan broj veći od nule ali manji od 1;
- nula;
- jedan negativan broj;
- jedan nenumerički znak.

Neki podaci su namerno uzeti iako su neispravni; testiramo da li kôd elegantno reaguje na nepravilne podatke.

Zatim, pregledamo unutrašnju strukturu programa i dodajemo druge slučajeve. Na primer, možemo da dodamo slučajeve kojima će se testirati sva grananja i da isprobamo što je moguće više putanja. Ako postoje petlje, mogli bismo da uključimo slučajeve za testiranje tako da se petlja izvrši jednom, više puta ili nijednom. Možemo takođe da ispitamo implementaciju algoritama. Na primer, ako program vrši trigonometrijska izračunavanja, možemo da uključimo primere koji testiraju granične vrednosti trigonometrijskih funkcija kao što su nula, 90, 180, 270 i 360 stepeni. Ili možemo da uključimo vrednosti zbog kojih će imenitelj razlomka dobiti vrednost nula.

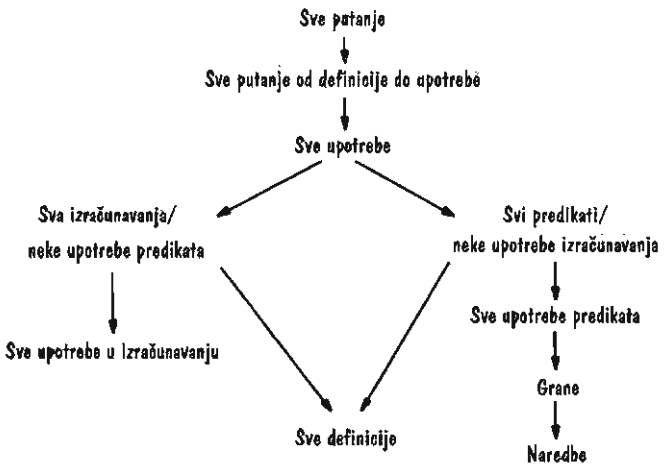
Ponekad sistem „pamti" stanja iz prethodnog primera i zato su potrebni skupovi test primera. Na primer, kada sistem implementira konačni automat, kôd mora da zadrži prethodno stanje sistema; prethodno stanje i trenutni ulaz određuju sledeće stanje. Slično tome, sistemi koji rade u realnom vremenu često se pokreću putem prekida; testiranjem treba isprobati grupe, a ne pojedinačne slučajeve.

Potpunost testiranja. Da bismo testirali, odlučujemo kako da na ubedljiv način pokažemo da test podaci prikazuju sva moguća ponašanja. Pogledajmo kakve su nam mogućnosti.

Da bismo potpuno testirali kôd, možemo da biramo slučajeve na jedan način ili na više načina koji su zasnovani na podacima koje kôd obrađuje:

- **Testiranje iskaza:** Svaki iskaz u komponenti izvršava se bar jednom u nekom testu.
- **Testiranje grananja:** Za svaku odluku koja se donosi u kodu, svaka grana se bira bar jednom u nekom testu.
- **Testiranje putanja:** Svaka zasebna putanja kroz kôd se izvršava bar jednom u nekom testu.
- **Testiranje putanje od definicije do upotrebe:** Nalaze se načini da se isproba svaka putanja od svake definicije svake promenljive, do svake upotrebe te promenljive.
- **Testiranje svih upotreba:** Skup za testiranje uključuje bar jednu putanju od svake definicije do svake upotrebe, u sklopu dometa definicije.
- **Testiranje svih upotreba predikata/nekih upotreba izračunavanja:** Za svaku promenljivu i svaku definiciju te promenljive test uključuje bar jednu putanju od definicije do svake upotrebe u izračunavanju; ako postoje definicije koje nisu pokrivene tim opisom, tada treba proširiti upotrebe u izračunavanju tako da se pokriju sve definicije.
- **Testiranje svih upotreba izračunavanja/nekih upotreba predikata:** Za svaku promenljivu i svaku definiciju te promenljive test uključuje bar jednu putanju od definicije do svake upotrebe u izračunavanju; ako postoje definicije koje nisu pokrivene tim opisom, tada treba uključiti upotrebe predikata tako da se pokriju sve definicije.

Postoje i druge slične vrste testiranja kao što su sve definicije, sve upotrebe predikata, sve upotrebe izračunavanja. Beizer (1990) opisuje odnose snaga ovih strategija testiranja, kao što se vidi na slici 8.6. Na primer, testiranje svih putanja je jače od testiranja svih putanja od definicije do upotrebe. U opštem slučaju, što je jača strategija, više će slučajeva biti uključeno; uvek moramo da uzmemo u obzir odnos između resursa dostupnih za testiranje i potpunosti izabrane strategije.



SLIKA 8.6 Odnosi snaga strategija testiranja (Beizer 1990).

Veća je verovatnoća da uspemo korišćenjem neke strategije, nego nasumičnim testiranjem. Na primer, Ntafos (1984) poredi nasumično testiranje sa testiranjem grananja i testiranjem svih upotreba, na sedam matematičkih programa sa poznatim greškama. On je utvrdio da nasumično testiranje otkriva 79,5% grešaka, testiranje grananja 85,5%, a testiranje svih upotreba 90% grešaka.

Da biste videli kako strategija utiče na broj slučajeva za testiranje, razmotrite primer sa slike 8.7, na kojoj je ilustrovan logički tok komponente koja treba da se testira. Svaki iskaz predstavljen rombom ili pravougaonikom nosi svoj broj. Testiranje iskaza zahteva slučajeve koji će izvršiti iskaze od 1 do 7. Ako izaberemo X veće od K tako da REZULTAT bude pozitivan, možemo da izvršimo iskaze:

1-2-3-4-5-6-7

tako da je dovoljan jedan slučaj.

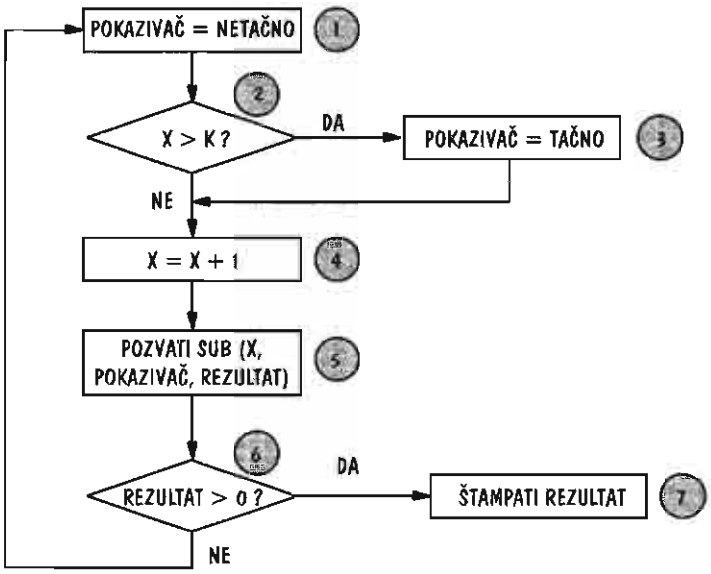
Za testiranje grananja, moramo da pronađemo sve tačke odlučivanja koje su na slici 8.7 prikazane kao rombovi. Postoje dve odluke: jedna ispituje odnos X i K a druga ispituje da li je REZULTAT pozitivan. Dva slučaja će izvršiti putanje:

1-2-3-4-5-6-7

i

1-2-4-5-6-1

pa će bar jednom proći kroz svako grananje. Prva putanja koristi granu *da* u prvoj odluci, a druga koristi granu *ne*. Isto tako, prva putanja koristi granu *da* u drugoj odluci, a druga koristi granu *ne*.



SLIKA 8.7 Logički tok

Ako želimo da isprobamo sve moguće putanje kroz program, biće nam potrebno više slučajeva. Putanje:

1-2-3-4-5-6-7
1-2-3-4-5-6-1
1-2-4-5-6-7
1-2-4-5-6-1

pokrivaju sve mogućnosti: dve tačke odlučivanja sa po dve mogućnosti u svakoj grani.

U našem primeru, testiranje iskaza je zahtevalo manje slučajeva od testiranja grananja, a ono opet zahteva manje slučajeva od testiranja putanja. Taj odnos važi u opštem slučaju. Što je program složeniji, to je potrebno više slučajeva za testiranje putanja. U četvrtom zadatku ispituje se odnos između strukture i redosleda tačaka odlučivanja prema broju putanja kroz kôd.

Postoji više različitih strategija testiranja koje mogu da se koriste prilikom jediničnog testiranja. Na primer, bezbedne aplikacije se često testiraju tako što se svaka moguća transakcija prati do kraja pomoću strategije koja se naziva **testiranje toka transakcije**. Potpunu diskusiju o strategijama testiranja pročitajte u knjizi koju je objavio Beizer (1990).

Poređenje tehnika

Jones (1991) je vršio poređenje nekoliko tipova metoda za otkrivanje grešaka pokušavajući da utvrdi za koju je verovatnije da će otkriti određene kategorije grešaka. U tabeli 8.5 prikazani su rezultati njegove ankete, uređene prema aktivnosti razvoja koja je dovela do greške. Na primer, ako se greška nalazi u kodu, ali je do nje došlo zbog problema sa specifikacijom zahteva, ona je stavljena u kolonu „zahtevi".

Tabela 8.5 Procenti pronađenih grešaka prema uzrocima grešaka (Jones 1991)

Tehnika otkrivanja	Zahtevi	Dizajn	Kodiranje	Dokumentacija
Prototipovi	40	35	35	15
Pregled zahteva	40	15	0	5
Pregled dizajna	15	55	0	15
Inspekcija koda	20	40	65	25
Jedinično testiranje	1	5	20	0

Tabela 8.6 Efikasnost tehnika za pronalaženje grešaka (Jones 1991)

	Greške u zahtevima	Greške u dizajnu	Greške u kodu	Greške u dokumentaciji
Pregledi	Dobro	Odlično	Odlično	Vrlo dobro
Prototipovi	Vrlo dobro	Dobro	Dobro	Nije primenljivo
Testiranje	Slabo	Slabo	Vrlo dobro	Dobro
Dokazivanje ispravnosti	Slabo	Slabo	Dobro	Dobro

PODSETNIK 8.4 EFIKASNOST PRONALAŽENJA GREŠAKA U CONTEL IPC-U

Olsen (1993) opisuje razvoj sistema od 184 hiljada redova koda na bazi C, Objective C, asemblera i skript jezika, u kompaniji koja je pružala automatizovanu podršku finansijerima. On je analizirao greške otkrivene tokom različitih aktivnosti i uočio razlike: 17,3% grešaka je otkriveno tokom inspekcija dizajna sistema, 19,1% tokom inspekcija dizajna komponenti, 15,1% tokom inspekcija koda, 29,4% tokom integracionog testiranja i 16,6% tokom testiranja sistema i regresionog testiranja. Samo je 0,1% grešaka otkriveno posle instaliranja sistema na licu mesta. Olsenov rad, dakle, pokazuje značaj korišćenja različitih tehnika da bi se različite vrste grešaka „iscačkale" tokom razvoja; nije dovoljno osloniti se samo na jedan metod za hvatanje svih vrsta problema.

Jones (1991) je takođe istraživao koju vrstu grešaka najbolje pronalazi određena vrsta tehnike otklanjanja. U tabeli 8.6 se vidi da pregledi i inspekcije najefikasnije pronalaze greške u dizajnu i kodu, ali da prototipovi najbolje pronalaze probleme koji su posledica zahteva. U podsetniku 8.4 objašnjeno je zašto je za pronalaženje grešaka dobro koristiti skup različitih tehnika.

8.4 INTEGRACIONO TESTIRANJE

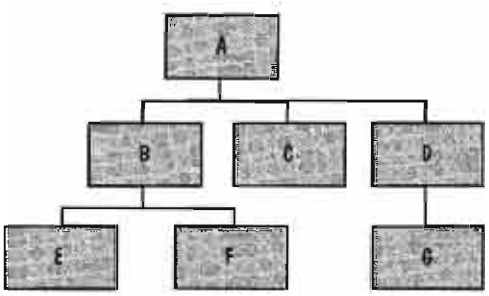
Kada se uverimo da pojedinačne komponente pravilno rade i zadovoljavaju naše ciljeve, mi ih ugrađujemo u sistem koji radi. Integracija se planira i koordinira na način da kada dođe do nepravilnosti unapred postoji orijentir u vezi sa uzrokom. Osim toga, redosled kojim se komponente testiraju utiče na izbor slučajeva i alata. Kod većih sistema su neke komponente možda još u fazi kodiranja, druge u fazi jediničnog testiranja, a neke grupe komponente se testiraju zajedno. Naša strategija testiranja objašnjava zašto i kako se komponente kombinuju radi testiranja sistema. Strategija utiče ne samo na vremenski raspored integracije i redosled kodiranja, već takođe i na troškove i sveobuhvatnost testiranja.

Sistem se ponovo posmatra kao hijerarhija komponenti, gde svaka komponenta pripada nekom sloju dizajna. Moguće je početi od vrha i napredovati nadole tokom testiranja, moguće je ići odozdo nagore ili kombinovati oba pristupa.

Integracija odozdo nagore

Jedan popularan pristup za spajanje komponenti za testiranje većeg sistema naziva se **testiranje odozdo nagore**. Kada se koristi ovaj metod, prvo se jedinično testira svaka komponenta na najnižem nivou hijerarhije sistema. Zatim se testiraju komponente koje pozivaju prethodno testirane komponente. Taj postupak se ponavlja sve dok testiranjem ne budu obuhvaćene sve komponente. Metod odozdo nagore je koristan kada većina komponenti na najnižem nivou predstavlja pomoćne rutine opšte namene koje se pozivaju iz drugih komponenti, kod objektno orijentisanog dizajna ili kada sistem integriše veći broj nezavisnih ponovno iskoristivih komponenti.

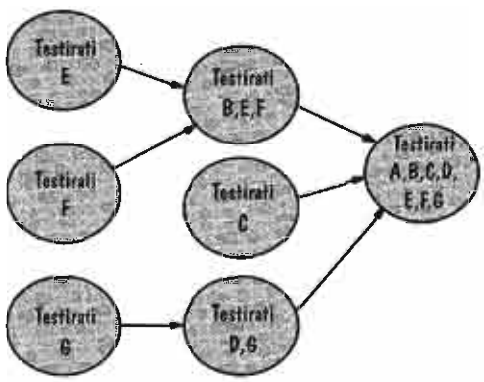
Na primer, uzmimo komponente i hijerarhiju na slici 8.8. Za testiranje ovog sistema odozdo nagore, prvo testiramo najniži nivo: E, F i G. Pošto komponente koje pozivaju programe najnižeg nivoa nisu spremne, pišemo poseban kôd kao pomoć za integraciju. **Rukovalac komponentom** je rutina koja poziva određenu komponentu i saopštava joj slučaj. Rukovalac se lako kodira, pošto retko zahteva složenu obradu. Međutim, potrebno je paziti da se pravilno definiše interfejs rukovaoca i komponente koja se testira. Ponekad se podaci za testiranje mogu automatski predati na jeziku specijalne namene koji olakšava definisanje podataka.



SLIKA 8.8 Primer hijerarhije komponenti

U našem primeru, potrebni su nam rukovaoci komponenti za E, F i G. Kada se uverimo da te tri komponente ispravno rade, prelazimo na sledeći viši nivo. Za razliku od komponenti najnižeg nivoa, komponente sledećeg nivoa se ne testiraju pojedinačno. One se kombinuju sa komponentama koje pozivaju (a koje su već testirane). U ovom slučaju, zajedno testiramo B, E i F. Ako se pojavi problem, znamo da je uzrok u B, ili u interfejsu između B i E, ili između B i F, pošto E i F pojedinačno ispravno rade. Da smo zajedno testirali B, E i F bez prethodnog jediničnog testiranja E i F, možda ne bismo mogli tako lako da otkrijemo uzrok problema.

Na sličan način testiramo D sa G. Pošto C ne poziva nijednu drugu komponentu, testiramo je izolovano. Na kraju testiramo sve komponente zajedno. Na slici 8.9 prikazan je redosled testiranja i njihove međuzavisnosti.



SLIKA 8.9 Testiranje odozdo nagore

Testiranje odozdo nagore se često kritikuje kod funkcionalno dekomponovanih sistema, jer se komponente najvišeg nivoa, koje su obično najvažnije, poslednje testiraju. Najviši nivo upravlja glavnim aktivnostima u sistemu, dok niži nivoi često obavljaju prizemnije poslove, kao što su ulazne i izlazne funkcije ili izračunavanja koja se ponavljaju. Zato neki projektanti smatraju da se time otkrivanje značajnijih grešaka odlaže do završetka testiranja, budući da se prvo testiraju niži nivoi. Štaviše, ponekad greške na visokom nivou ukazuju na greške u dizajnu; očigledno bi te greške trebalo ispraviti u što je moguće ranijim fazama razvoja, a ne čekati do samog kraja. Na kraju, komponente najvišeg nivoa često kontrolišu vreme i na njega utiču. Teško je testirati sistem odozdo nagore ako u velikom delu obrade u sistemu postoji vremenska zavisnost.

S druge strane, testiranje odozdo nagore je često najprimerenije za objektno orijentisane programe. Objekti se kombinuju jedan po jedan sa objektima ili kolekcijama prethodno testiranih objekata. Poruke se razmenjuju i testiranje obezbeđuje da objekti pravilno reaguju.

Integracija odozgo nadole

Mnogi projektanti više vole **pristup odozgo nadole** koji je na razne načine suprotan pristupu odozdo nagore. Na najvišem nivou obično imamo jednu kontrolnu komponentu i nju izolovano testiramo. Zatim se komponente koje ona poziva kombinuju i testiraju kao veća celina. Isto se ponavlja dok se ne uključe sve komponente.

Komponenta koja se testira možda poziva drugu koja još nije testirana i zato pišemo **lažnu rutinu (stab)**, program specijalne namene koji simulira aktivnost komponente koja nedostaje. Lažna rutina odgovara na poziv i vraća izlazne podatke tako da postupak testiranja može da se nastavi. Na primer, ako se poziva komponenta koja izračunava sledeću slobodnu adresu ali ta komponenta još nije testirana, tada lažna rutina može da vraća jednu fiksnu adresu samo da bi testiranje moglo da se nastavi. Kao što važi i za rukovaoca, lažne rutine ne moraju da budu složene niti logički potpune.

Na slici 8.10 prikazano je kako bi se naš primer sistema testirao odozgo nadole. Najviša komponenta A je jedina koja se testira izolovano, dok su za B, C i D potrebne lažne rutine. Pošto se ona istestira, kombinuje se sa sledećim nivoom pa se zajedno testiraju A, B, C i D. U ovoj fazi testiranja biće potrebne lažne rutine za E, F ili G. Na kraju se testira ceo sistem.

Ako komponente najnižeg nivoa vrše samo ulaz i izlaz, lažne rutine za njih bile bi skoro identične komponentama koje zamenjuju. U tom slučaju se može promeniti redosled integracije tako da se komponente ulaza i izlaza ugrade ranije.

Mnoge prednosti projektovanja i programiranja sistema odozgo nadole važe i za testiranje odozgo nadole. Kada se, primenom dizajna odozgo nadole, funkcije konkretnih komponenti lokalizuju, testiranje odozgo nadole omogućava da se isproba jedna po jedna funkcija, počevši od najvišeg nivoa kontrole, kroz odgovarajuće komponente, do dna. Tako se slučajevi mogu definisati prema funkcijama koje se ispituju. Uz to, bilo koja greška u dizajnu ili pitanje od značaja za fleksibilnost funkcija može da se postavi na početku, a ne na kraju procesa testiranja.

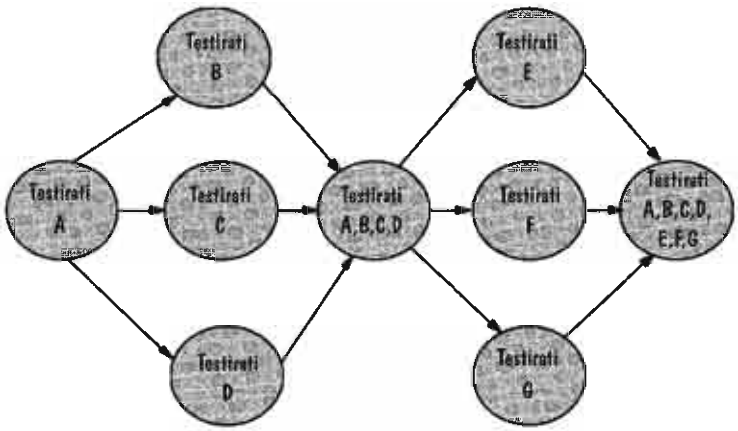
SLIKA 8.10 Testiranje odozgo nadole



Primetićete takođe da za testiranje odozgo nadole nisu potrebni programi rukovaoci. S druge strane, pisanje lažnih rutina može da bude teško, pošto one moraju da podrže testiranje svih mogućih uslova. Na primer, pretpostavimo da komponenta Z u sistemu za crtanje mapa obavlja izračunavanje na osnovu geografske širine i dužine koje dobija iz komponente Y. Specifikacija dizajna određuje da je izlaz iz Y uvek u severnoj hemisferi. Pošto Z poziva Y, u trenutku kada se Z uključuje u testiranje odozgo nadole, Y možda još nije kodiran. Ako napišemo lažnu rutinu koja će davati jedan broj između 0 i 180 da bi se omogućilo testiranje komponente Z, lažna rutina će morati da se promeni, ako se u dizajn uključi i južna hemisfera. To znači da je lažna rutina važan deo testiranja i da njena ispravnost može da ima uticaja na validnost celog testa.

Nedostatak testiranja odozgo nadole jeste potencijalna potreba za velikim brojem lažnih rutina. To se može dogoditi ako najniži nivo sistema sadrži veliki broj rutina opšte namene. Jedan od načina da se taj problem izbegne je mala promena strategije. Umesto da se istovremeno uključuje jedan ceo nivo, u modifikovanom pristupu odozgo nadole se pre objedinjavanja jedinično testiraju komponente svakog nivoa. Na primer, naš primer sistema može se testirati po modifikovanom pristupu tako što se prvo testira A, zatim B, C i D, i tek tada se te četiri komponente integrišu radi testiranja prvog i drugog nivoa. Zatim se E, F i G testiraju pojedinačno. Na kraju se ceo sistem spaja za testiranje kao na slici 8.11.

Jedinično testiranje komponenti svakog nivoa uvodi još jednu poteškoću. Za svaku komponentu su potrebne i lažne rutine i programi rukovaoci, što rezultira sa više kodiranja, a time i više potencijalnih problema.



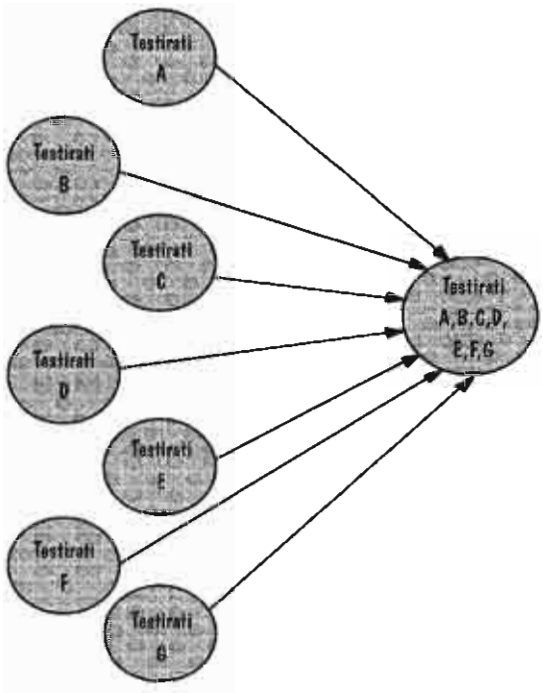
SLIKA 8.11 Modifikovano testiranje odozgo nadole

Integrisanje po principu „velikog praska"

Kada su sve komponente jedinično testirane, teško je odupreti se iskušenju da se sve one sastave u konačni sistem i isproba da li će sve odmah raditi. Myers (1979) to naziva **testiranjem „po principu velikog praska"**, a na slici 8.12 prikazana je njegova primena na naš primer sistema. Mnogi programeri koriste ovaj pristup kod malih sistema, dok njegova primena na velike sisteme nije praktična. U stvari, pošto testiranje „po principu velikog praska" ima niz nedostataka, ono se ne preporučuje ni za jedan sistem. Prvo,

za testiranje pojedinačnih komponenti potrebne su i lažne rutine i programi rukovaoci. Drugo, pošto se sve komponente integrišu odjednom, teško je pronaći uzrok bilo koje nepravilnosti. Na kraju, greške u interfejsima je teško razlučiti od ostalih vrsta grešaka.

SLIKA 8.12 Testiranje „po principu velikog praska"

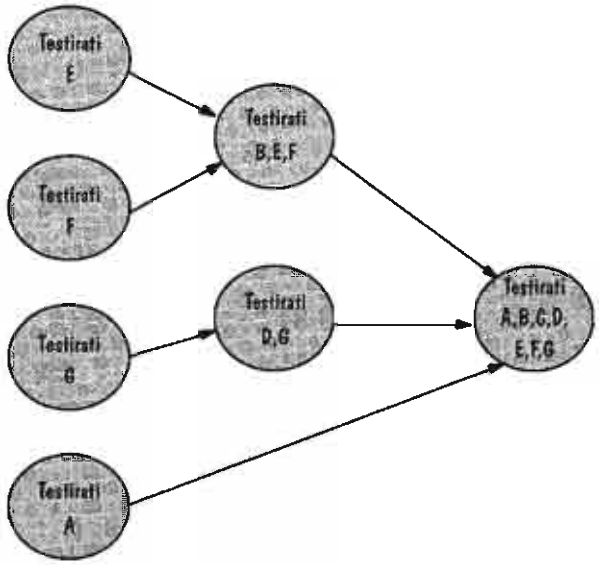


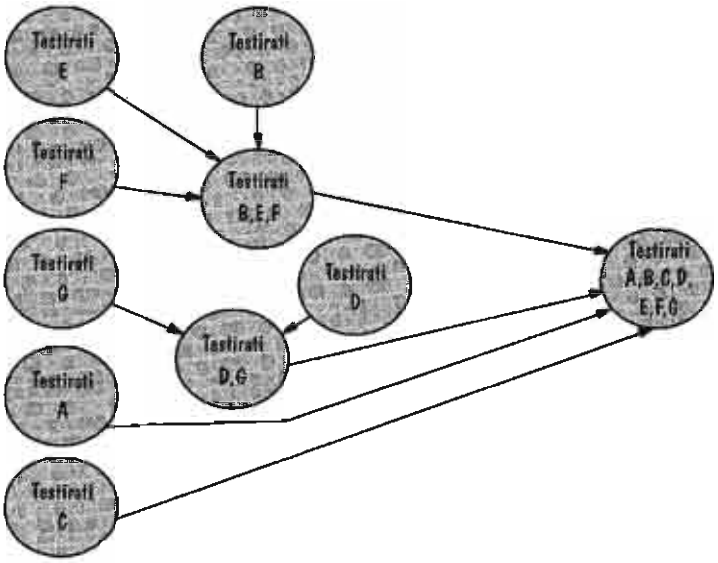
Sendvič integracija

Myers (1979) kombinuje strategiju odozgo nadole sa strategijom odozdo nagore i tako obrazuje tzv. **sendvič testiranje**. Sistem se posmatra u tri sloja, kao sendvič: ciljni sloj u sredini, sloj neposredno iznad i sloj neposredno ispod ciljnog. Za gornji sloj se koristi pristup odozgo nadole, a za donji sloj pristup odozdo nagore. Testiranje konvergira prema ciljnom sloju koji se bira na osnovu karakteristika sistema i strukturne hijerarhije komponenti. Na primer, ako sloj na dnu sadrži veliki broj pomoćnih programa opšte namene, ciljni sloj može da bude sloj iznad njega u kojem se nalaze komponente koje koriste pomoćne programe. Na ovaj način se na početku testiranja proverava ispravnost pomoćnih programa testiranjem odozdo nagore. Lažne rutine za pomoćne programe nisu potrebne, pošto se za to mogu upotrebiti sami pomoćni programi. Na slici 8.13 prikazan je jedan od mogućih redosleda za sendvič integraciju našeg primera hijerarhije komponenti, gde je ciljni sloj u sredini, tj. komponente B, C i D.

Sendvič testiranje omogućava da se integraciono testiranje obavi u ranim fazama. Ono takođe objedinjava prednosti pristupa odozgo nadole i pristupa odozdo nagore tako što se na početku testiraju kontrolne komponente i pomoćni programi. Međutim, ne vrši se sveobuhvatno testiranje pojedinačnih komponenti pre integracije. Jedna varijacija, modifikovano sendvič testiranje, predviđa testiranje komponenti višeg nivoa pre njihovog spajanja sa ostalima, kao što se vidi na slici 8.14.

SLIKA 8.13 Sendvič testiranje





SLIKA 8.14 Modifikovano sendvič testiranje

Poređenje različitih integracionih strategija

Izbor integracione strategije ne zavisi samo od karakteristika sistema, već i od očekivanja kupca. Na primer, kupac možda želi što pre da vidi radnu verziju, pa možemo da prihvatimo strategiju kojom se sistem koji funkcioniše gradi u ranim fazama procesa testiranja. Na taj način, neki programeri još kodiraju dok drugi testiraju, pa se faze testiranja i kodiranja preklapaju. Myers (1979) je sastavio matricu prikazanu u tabeli 8.7 u kojoj se porede različite strategije testiranja na osnovu nekoliko atributa sistema i potreba kupaca. U podsetniku 8.5 objašnjena je Microsoftova strategija, koju pokreću pritisci tržišta.

Tabela 8.7 Poređenje strategija integracije (Myers 1979)

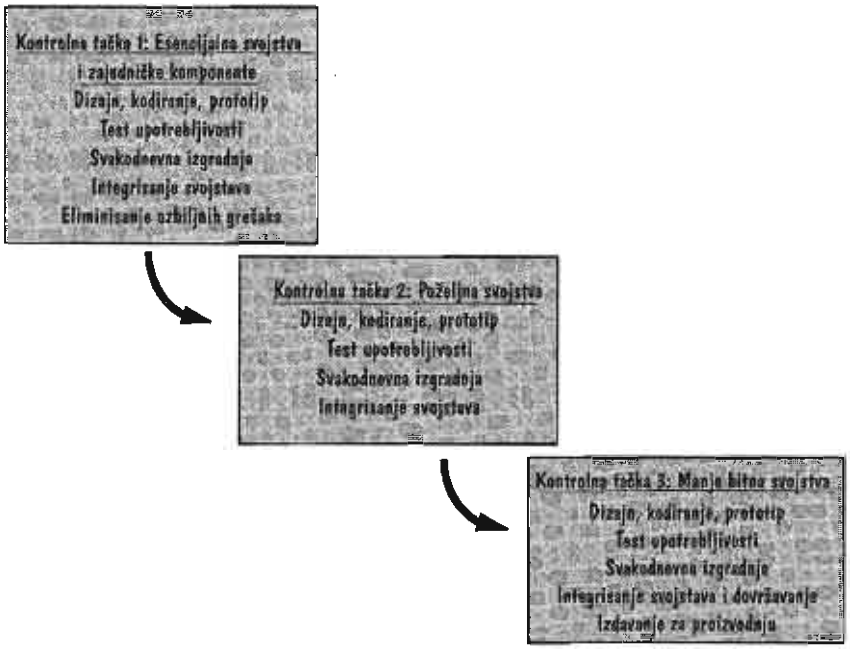
	Odozdo nagore	Odozgo nadole	Modifikovana odozgo nadole	Veliki prasak	Sendvič	Modifikovani sendvič
Integracija	Rano	Rano	Rano	Kasno	Rano	Rano
Vreme do osnovnog programa koji radi	Kasno	Rano	Rano	Kasno	Rano	Rano
Potrebni rukovaoci	Da	Ne	Da	Da	Da	Da
Potrebne lažne rutine	Ne	Da	Da	Da	Da	Da
Paralelan rad na početku	Srednje	Malo	Srednje	Veoma	Srednje	Veoma
Mogućnost testiranja konkretnih putanja	Lako	Teško	Lako	Lako	Srednje	Lako
Mogućnost planiranja i kontrole sekvence	Lako	Teško	Teško	Lako	Teško	Teško

PODSETNIK 8.5 IZGRADNJA U MICROSOFTU

Microsoftovu integracionu strategiju pokreće tržište, na osnovu potrebe da proizvod bude u funkciji što je moguće pre (Cusumano i Selby 1995 i 1997). Microsoft koristi više malih timova (od po tri do osam učesnika) koji rade paralelno i implementiraju tzv. pristup „sinhronizovati i stabilizovati". U postupku se naizmenično dizajniraju, grade i testiraju komponente, a kupci se uključuju u postupak testiranja. Svi delovi proizvoda često se integrišu da bi se utvrdilo šta radi ispravno, a šta ne.

Microsoftov pristup omogućava timu da menja specifikacije osobina kako se projektanti postepeno sve bolje upoznaju sa mogućnostima i potrebama proizvoda. Ponekad se skup osobina promeni i do 30%. Proizvod i projekat se podele na delove na osnovu osobina, a za njihovu realizaciju se zadužuju različiti timovi. Zatim se definišu kontrolne tačke, na osnovu klasifikacije osobina na esencijalne, poželjne i manje bitne. Timovi sinhronizuju rad tako što svakodnevno grade proizvod i pronalaze i ispravljaju greške, kao što je prikazano na slici 8.15. Tako se esencijalna svojstva prvo razvijaju i integrišu, a svaka kontrolna tačka ima i „rezervno vreme" za prevazilaženje neočekivanih komplikacija i kašnjenja. Ako vreme isporuke mora da se skrati, manje bitna svojstva se isključuju iz projekta.

Bez obzira na izabranu strategiju, svaka komponenta se u cilju testiranja samo jednom ugrađuje. Osim toga, komponenta ne bi nikada smela da se menja radi olakšavanja procesa testiranja. Lažne rutine i programi rukovaoci su zasebni novi programi, a ne privremene izmene postojećih programa.



SLIKA 8.15 Microsoftov pristup sinhronizovanja i stabilizovanja

8.5 TESTIRANJE OBJEKTNO ORIJENTISANIH SISTEMA

Mnoge opisane tehnike za testiranje sistema odnose se na sve tipove sistema, pa tako i na objektno orijentisane. Međutim, trebalo bi preduzeti nekoliko dodatnih koraka u cilju provere da li se karakteristike objektno orijentisanih programa pravilno tretiraju u sklopu odabranih tehnika za testiranje.

Testiranje koda

Rumbaugh i dr. (1991) predlažu da testiranje objektno orijentisanih sistema započne postavljanjem sledećih pitanja:

- Kada kôd očekuje jedinstvenu vrednost, da li postoji putanja koja tu vrednost generiše?
- Kada postoji više mogućih vrednosti, da li postoji način za izbor jedinstvenog rezultata?
- Postoje li korisni slučajevi koji nisu obrađeni?

Zatim, treba da proverite same objekte i klase tražeći viškove i manjkove: objekte koji nedostaju, nepotrebne klase, asocijacije koje nedostaju, ili su nepotrebne, ili pogrešno postavljene, ili isto takve atribute. Rumbaugh i dr. (1991) daju smernice za otkrivanje tih slučajeva u toku testiranja. Oni ističu da se može govoriti o nedostatku objekata ako:

- naiđete na asimetrične asocijacije i generalizacije;
- naiđete na neusklađene atribute i operacije nad klasom;

- jedna klasa ima dve ili više različitih uloga;
- operacija nema dobru ciljnu klasu;
- naiđete na dve asocijacije sa istim imenom i namenom.

Klasa može da bude nepotrebna ako nema atributa, operacija ili asocijacija. Slično tome, asocijacija može da bude nepotrebna ako sadrži redundantne informacije ili ako tu putanju ne koristi nijedna operacija. Ako su imena uloga preširoka ili preuska sa aspekta pozicije, asocijacija možda nije na pravom mestu. Ako se objektu pristupa preko vrednosti nekog njegovog atributa, lokacija atributa može biti pogrešna. Za svaku od tih situacija Rumbaugh i dr. (1991) predlažu da se nastala situacija prevaziđe promenama u dizajnu.

Smith i Robson (1992) smatraju da bi testiranje trebalo da obuhvati mnoge različite nivoe: funkcije, klase, klastere (grupe objekata koji sarađuju i međusobno utiču jedni na druge), i sistem kao celinu. Tradicionalni načini testiranja primenljivi su na funkcije, ali u nekim načinima testiranja se ne uzimaju u obzir stanja objekata potrebna da bi se testirala klasa. Trebalo bi bar razviti testove koji prate stanje objekta i promene tog stanja. Tokom testiranja, obratite pažnju na probleme u vezi sa konkurentnim radom i sinhronizacijom, i proverite da li su odgovarajući događaji potpuni i konsistentni.

Razlike između objektno orijentisanog i tradicionalnog testiranja

Perry i Kaiser (1990) pažljivo proučavaju testiranje objektno orijentisanih komponenti, pogotovo onih koje se ponovno koriste iz drugih aplikacija. Svojstva objektne orijentacije često otežavaju minimizovanje testova, ali to nije uvek slučaj. Na primer, komponente koje su zasebno razvijene izolovane su enkapsuliranjem. Mogao bi se izvesti zaključak da, ako se neke komponente ponovo upotrebe u neizmenjenom stanju a druge pre upotrebe promene, potrebno je ponovo testirati samo one koje su menjane. Međutim, „... program koji je adekvatno testiran izolovano možda nije adekvatno testiran u kombinaciji" (Perry i Kaiser 1990). U stvari, oni pokazuju da kada dodajemo novu potklasu ili menjamo postojeću potklasu, moramo ponovo da testiramo metode nasleđene od svake natklase.

Oni takođe ispituju adekvatnost slučajeva za testiranje. Kod proceduralnih jezika, za testiranje sistema možemo upotrebiti skup podataka; zatim, ako dođe do promene sistema možemo, oslanjajući se na postojeći skup testiranih podataka, proveriti da li je dodatna odnosno preostala funkcionalnost nepromenjena. Kada se nasleđeni metod u potklasi zamenjuje istoimenim lokalno definisanim metodom, potklasa u kojoj se metod nadjačava mora ponovo da se testira, ali za testiranje verovatno treba upotrebiti drugi skup podataka. Harrold i McGregor (1989) opisuju tehniku gde se istorija slučajeva za objektno orijentisani sistem koristi u svrhu svođenja dodatnog testiranja na najmanju moguću meru. Oni prvo testiraju osnovne klase koje nemaju roditelje; strategija testiranja je da se prvo pojedinačno testiraju sve funkcije, a zatim njihova uzajamna dejstva. Nakon toga oni daju algoritam za inkrementalno ažuriranje istorije testiranja roditeljske klase; testiraju se samo novi atributi i oni na koje mehanizam nasleđivanja utiče.

Graham (1996a) daje rekapitulaciju razlika između objektno orijentisanog i tradicionalnog testiranja na dva načina. Ona prvo evidentira aspekte objektne orijentacije koji olakšavaju testiranje i one koji ga otežavaju. Na primer, objekti su uglavnom manji, a složenost, koja je obično svojstvena strukturi komponente, često prelazi na interfejse između komponenti. Zbog tih razlika je jedinično testiranje jednostavnije, dok se integraciono testiranje mora šire posmatrati. Kao što smo videli, enkapsuliranje se često smatra

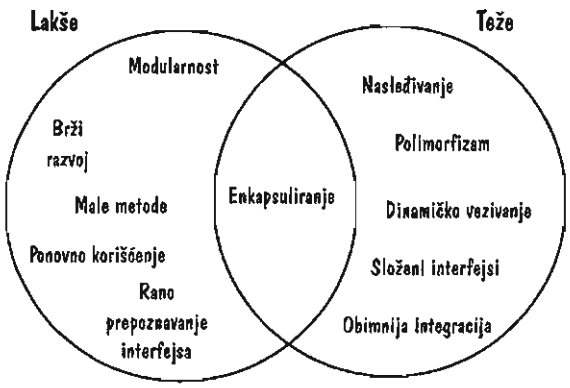
pozitivnom osobinom objektno orijentisanog dizajna, ali ono zahteva intenzivnije integraciono testiranje.

Slično tome, i nasleđivanje uvodi dodatna testiranja. Nasleđena funkcija zahteva dodatno testiranje ako:

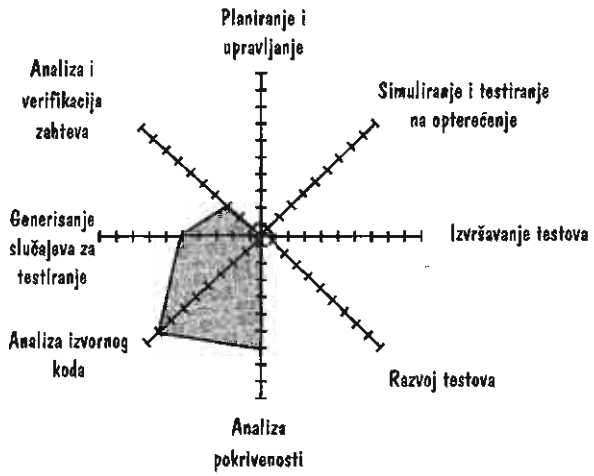
- je redefinisana;
- ima specifično ponašanje u izvedenoj klasi;
- ostale funkcije u toj klasi treba da budu konsistentne.

Na slici 8.16 prikazano je kako Graham posmatra ove razlike.

Graham takođe posmatra na koje korake u postupku testiranja utiče objektna orijentacija. Dijagram na slici 8.17 je Kivijatov, ili radarski grafikon, na kojem se porede razlike između objektno orijentisanog i tradicionalnog testiranja. Sivi mnogougao pokazuje da analiza i validacija zahteva, generisanje slučajeva, analiza izvornog koda i analiza pokrivenosti zahtevaju više pažnje. Što je siva linija udaljenija od centra dijagrama, veća je razlika između objektno orijentisanog i tradicionalnog testiranja.



SLIKA 8.16 Lakši i teži delovi testiranja objektno orijentisanih sistema (Graham 1996a)



SLIKA 8.17 Značajni aspekti domena testiranja u kojima je objektno orijentisano testiranje drugačije (Graham 1996a)

- Zahtevi se obično iznose u dokumentu zahteva, ali je malo alata koji podržavaju validaciju zahteva iskazanih u formi objekata i metoda.
- Isto tako, većina alata koji olakšavaju izradu slučajeva za testiranje ne može da prihvati modele zasnovane na objektima i metodama.
- Većina mera izvornog koda definisana je za proceduralni kôd, a ne za objekte i metode. Tradicionalne mere kao što su ciklomatski brojevi nisu od velike koristi za izračunavanje veličine i složenosti objektno orijentisanih sistema. Vremenom, kada istraživači predlože i istestiraju iskoristive objektno orijentisane mere, ova razlika će se smanjiti.
- Pošto je izvor složenosti objektno orijentisanih sistema interakcija između objekata, mere i alati zasnovani na analizi pokrivenosti imaju manji značaj nego kod testiranja tradicionalnih sistema.

8.6 PLANIRANJE TESTA

Kao što smo videli, testiranje komponenti i njihovo integrisanje u sistem nije jednostavno. Pažljivo planiranje testa pomaže da projektujemo i organizujemo proces testiranja tako da budemo sigurni da je ono obavljeno potpuno i na odgovarajući način.

Svaki korak u procesu testiranja mora da se planira. U stvari, proces testiranja ima vlastiti životni ciklus u sklopu razvoja i može da teče paralelno sa drugim aktivnostima. Pogotovo je potrebno da se planiraju sledeći koraci u procesu testiranja:

1. utvrđivanje ciljeva testiranja;
2. dizajn slučajeva;
3. pisanje slučajeva;
4. testiranje slučajeva;
5. izvršavanje testova;
6. ocenjivanje rezultata testiranja.

Cilj testiranja nas upućuje na vrste slučajeva za testiranje koje treba da izgradimo. Štaviše, dizajn slučajeva je od ključnog značaja za uspeh testiranja. Ako slučajevi nisu reprezentativni i ne izvršavaju u potpunosti funkcije koje dokazuju ispravnost i validnost sistema, ostalo testiranje postaje beskorisno.

Prema tome, izvršavanje testa počinje pregledom slučajeva u cilju provere njihove ispravnosti, izvodljivosti, postizanja željenog stepena pokrivanja i demonstracije željene funkcionalnosti. Kada se te provere obave, može se preći na samo izvršavanje testova.

Svrha plana

Plan testiranja koristimo za organizovanje aktivnosti testiranja. Plan testiranja uzima u obzir ciljeve testa i sadrži raspored diktiran strategijom testa ili rokovima isporuke projekta. Životni ciklus razvoja sistema zahteva nekoliko nivoa testiranja, počevši od jediničnih i integracionih testova, pa sve do dokazivanja pune funkcionalnosti sistema.

Plan testiranja opisuje način na koji ćemo kupcima pokazati da softver ispravno radi (tj. da softver ne sadrži greške i da obavlja zahtevane funkcije). Na taj način, plan testiranja ne sadrži samo jedinično i integraciono testiranje, već i testiranje sistema. Plan testiranja je putokaz za celokupnu aktivnost testiranja. On objašnjava ko će obaviti testiranje, zašto se vrše testovi, kako se testovi vode i kakav im je raspored.

Da bismo napravili plan testiranja, moramo da poznajemo zahteve, specifikacije funkcija i hijerarhiju modula u dizajnu sistema i koda. Kada razvijamo svaki od ovih elemenata sistema, možemo da upotrebimo svoje znanje pri izboru ciljeva testiranja, definisanju strategije testiranja i izradi slučajeva. Na taj način se plan testiranja razvija paralelno sa samim sistemom.

Sadržaj plana

Na početku plana testiranja navode se ciljevi, sa opisom svake vrste testiranja, od jediničnog, preko funkcionalnog i testa prihvatanja, sve do instalacionog testa. Na taj način je plan testiranja sistema u stvari skup planova testiranja, po jedan za svaku vrstu. Plan opisuje kako će se testovi izvršavati i koji kriterijumi će se primeniti za utvrđivanje da li je testiranje potpuno. Nije uvek lako odrediti da je test završen. Videli smo primere koda za koje je nemoguće ili nepraktično generisanje svih mogućih kombinacija ulaznih podataka i uslova. Biranjem podskupa mogućih podataka svesno povećavamo verovatnoću da izostavimo testiranje na neke vrste grešaka. To balansiranje između potpunosti i realnosti troškova i vremena dovodi do kompromisa u ciljevima. Kasnije u ovom poglavlju videćemo kako se izračunava broj preostalih grešaka u kodu, kao i kako se prepoznaje kôd koji je sklon greškama.

Kada je tim za testiranje u stanju da utvrdi da li je zadovoljen svaki od ciljeva testiranja, kažemo da su ciljevi **dobro definisani**. Tada odlučujemo kako da integrišemo komponente u operativni sistem. Na nivou komponente razmatramo stepen pokrivenosti iskaza, grananja i putanja, a na nivou integracije razmatramo strategije odozgo nadole, odozdo nagore i ostale. Dobijeni plan združivanja komponenti se ponekad naziva **plan integrisanja sistema**.

Za svaku fazu testiranja, plan testiranja detaljno opisuje metode za svaki test. Na primer, za jedinično testiranje mogu da se upotrebe nezvanični letimičan pregled ili zvanične inspekcije, nakon kojih sledi analiza strukture koda, a tek onda analiza same performanse koda. U planu se napominje svaka automatizovana podrška, uključujući uslove potrebne za upotrebu alata. Te informacije služe timu za testiranje prilikom planiranja aktivnosti i raspoređivanja testova.

Uz svaki metod ili tehniku testiranja, daje se detaljan spisak slučajeva. Plan takođe objašnjava kako se generišu podaci za testiranje i evidentiraju izlazni podaci ili informacije o stanjima. Ako se za praćenje testova, podataka i rezultata koristi baza podataka, treba opisati i bazu podataka i njeno korišćenje.

Tako ćemo, čitajući plan testiranja, imati potpunu sliku o tome kako i zašto se testiranje vrši. Ako pišemo plan testiranja paralelno sa dizajnom sistema, primorani smo da shvatimo sveukupne ciljeve sistema. U stvari, ponekad iz perspektive testiranja možemo da dovedemo u pitanje prirodu problema i celishodnost rešenja.

Mnogi kupci definišu sadržaj plana testiranja već u dokumentaciji zahteva. Na primer, kada se gradi sistem, Ministarstvo odbrane SAD daje izvršiocu automatizovanu standardnu dokumentaciju. U standardima stoji da je plan testiranja

> „sredstvo kojim se usmerava testiranje i sadrži raspored redosleda događaja i spisak materijala potrebnih za obavljanje sveobuhvatnog testa kompleta [automatizovanog sistema podataka]. Delovi dokumenta namenjeni osoblju biće dati jezikom koji nije tehnički, a u delovima dokumenta namenjenim operativnom osoblju biće upotrebljena odgovarajuća terminologija." (Ministarstvo odbrane 1977)

U poglavlju 9 razmatramo detalje ovog primera za plana testiranja.

8.7 AUTOMATIZOVANI ALATI ZA TESTIRANJE

Postoje mnogi automatizovani alati koji pomažu prilikom testiranja komponenti, a u ovom poglavlju smo pomenuli nekoliko njih, kao što su alati za automatizovano dokazivanje teorema i alati za simboličko izvršavanje. Ali uglavnom, postoji više mesta u procesu testiranja gde su alati korisni, ako ne i od suštinskog značaja.

Alati za analizu koda

Postoje dve kategorije alata za analizu koda. **Statička analiza** se vrši dok se program ne izvršava; **dinamička analiza** se vrši u fazi izvršavanja programa. Svaka vrsta alata izveštava o samom kodu i o slučaju koji se koristi.

Statička analiza. Neki alati mogu da analiziraju kôd pre nego što se on pokrene. Alati koji ispituju ispravnost programa ili skupa komponenti mogu se grupisati u četiri vrste:

1. **Analiza koda:** Komponente se automatski proveravaju u smislu sintakse. Iskazi mogu da se markiraju ako je sintaksa pogrešna, ako je konstrukcija sklona greškama ili ako stavka nije definisana.
2. **Provera strukture:** Ovaj alat generiše grafičku predstavu podnetih komponenti. Graf prikazuje logički tok i proverava da li ima logičkih propusta.
3. **Analiza podataka:** Alat pregleda strukture podataka, deklaracije podataka i interfejse komponenti a zatim izveštava o nepravilnom povezivanju komponenti, neusklađenim definicijama podataka i neispravnoj upotrebi podataka.
4. **Provera redosleda:** Alat proverava redosled događaja; ako su kodirani u pogrešnom redosledu, događaji će biti markirani.

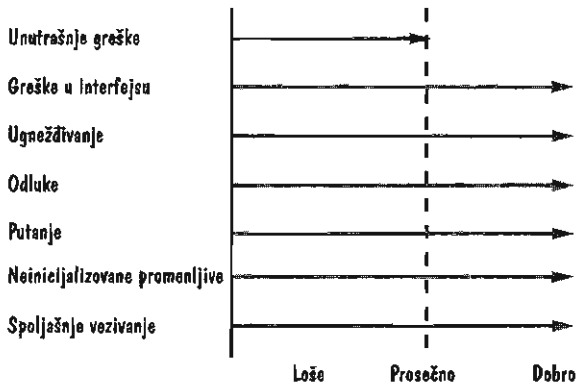
Na primer, pri analizi koda može se generisati tabela simbola u kojoj se evidentira gde se promenljiva definiše a gde se koristi, što pomaže za strategije testiranja kao što je testiranje putanje od definicije do upotrebe. Slično tome, provera strukture može da čita program i utvrdi lokacije svih petlji, označi iskaze koji se nikad ne izvršavaju, ukaže na grananja iz tela petlje i tako dalje. Analiza podataka može da nas obavesti kada imenilac može da postane nula; može takođe da proveri da li se argumenti pravilno prenose u potprograme. Ulazne i izlazne komponente mogu se podneti na proveru redosleda da bi se

utvrdilo da li su događaji kodirani u pravilnom redosledu. Na primer, provera redosleda može da obezbedi da svaka datoteka bude otvorena pre nego što se menja.

Rezultat većine alata za statičku analizu sadrži mere i strukturne karakteristike, tako da možemo bolje shvatiti svojstva programa. Na primer, uz dijagrame toka se često daju liste svih mogućih putanja kroz program, što pomaže pri planiranju slučajeva za testiranje putanja. Takođe dobijamo informacije vezane za broj ulaznih (fan-in) i izlaznih (fan-out) grana, broju operatora i operanada u programu, broju tačaka odlučivanja i nekoliko mera vezanih za složenost koda. Na slici 8.18 je dat primer izveštaja iz programa za statičku analizu, u kojem se nalazi za konkretan deo koda porede sa bazom istorijskih podataka. Poređenje se ne odnosi samo na mere kao što su dubina ugnežđivanja, snaga veza i broj odluka već takođe i na informacije o potencijalnim greškama i promenljivama koje nisu inicijalizovane. Ovakvi prikazi mogu da pomognu u oceni lakoće testiranja i ukažu na moguće greške koje možemo ispraviti pre formalnih testova.

Dinamička analiza. Često je sisteme teško testirati zato što se istovremeno obavljaju dve paralelne operacije. To posebno važi sa sisteme koji rade u realnom vremenu, jer je za njih teško predvideti uslove i napraviti reprezentativne slučajeve za testiranje. Automatizovani alati omogućavaju timu testera da uoči stanje događaja tokom izvršavanja programa tako što čuvaju „snimak" stanja. Ti alati se ponekad nazivaju **monitori programa** zato što posmatraju ponašanje programa i o tome izveštavaju.

Monitor može da izvesti o tome koliko puta je komponenta pozvana ili koliko puta je izvršen neki red koda. Takvi statistički podaci obaveštavaju testerski tim o stepenu pokrivenosti iskaza ili putanja odabranim slučajevima za testiranje. Isto tako, monitor može da izvesti o tome da li je u tački odluke bilo grananja na sve strane i informiše o pokrivenosti grananja. Takođe se daju i zbirni statistički podaci sa rekapitulacijom procenata pokrivanja za iskaze, putanje i grananja koja se odnose na ceo skup slučajeva. Ta informacija je važna kada su ciljevi testa iskazani u smislu pokrivenosti; na primer, za kontrolni sistem londonskog vazdušnog saobraćaja po ugovoru se tražilo 100% pokrivanje iskaza u testiranju (Pfleeger i Hatton 1997).



SLIKA 8.18 Izlaz iz statičke analize

РЕПУБЛИКА СРПСКА
Природно - математички
факултет Бања Лука
Ф И З И К А

Još neke informacije mogu da pomognu timu za testiranje u proceni performanse sistema. Moguće je napraviti statističke podatke o konkretnim promenljivama: na primer, njihova prva vrednost, poslednja vrednost, minimum i maksimum. Mogu se definisati tačke prekida u sistemu tako da alat za testiranje izvesti o tome da je određena promenljiva dostigla ili prešla zadatu vrednost. Neki alati mogu da se zaustave kada nastupi tačka prekida, tako da se može pregledati sadržaj memorije ili vrednosti određenih elemenata podataka; ponekad je moguće promeniti vrednosti u toku testiranja.

Za sisteme koji rade u realnom vremenu, evidentiranje što je moguće više informacija o određenom stanju ili uslovima tokom izvršavanja može se kasnije upotrebiti kao dodatna informacija o procesu testiranja. Tok kontrole može da se prati od tačke prekida unazad ili unapred, tako da testerski tim može da ispita prateće promene u podacima.

Alati za izvršavanje testa

Alati koje smo do sada opisivali bili su usredsređeni na kôd. Drugi alati mogu da se upotrebe za planiranje i izvršavanje samog testiranja. Posmatrajući veličinu i složenost većine današnjih sistema, alati za automatizovano izvršavanje testova su bitni jer omogućavaju veoma veliki broj slučajeva za testiranje, bez kojih sistem ne bi bio temeljito testiran.

Snimanje i ponavljanje. Kada se testovi planiraju, tim mora da, za svaki slučaj koji će se upotrebiti, navede ulaze i očekivani izlaz kao posledicu testirane aktivnosti. Alati za **snimanje i ponavljanje,** ili **snimanje i reprodukciju,** snimaju ulaz sa tastature i ostale ulaze, kao i odgovore u toku testiranja, i porede očekivane odgovore sa dobijenim. Tim dobija izveštaj o razlikama, a snimaljeni podaci pomažu da se uđe u trag razlozima odstupanja. Alati ove vrste su posebno korisni nakon otkrivanja i ispravljanja greške; oni mogu da se upotrebe za proveru da li je unetom izmenom greška ispravljena, i da li su pri tom u kôd unete nove greške.

Lažne rutine i rukovaoci. Ranije smo pomenuli značaj lažnih rutina i rukovalaca za integraciono testiranje. Na tržištu postoje alati koji pomažu u automatskoj izradi lažnih rutina i programa rukovalaca. Međutim, rukovaoci testiranja ponekad rade i više od jednostavnog pokretanja komponente. Rukovalac može da:

1. podesi vrednosti svih odgovarajućih promenljivih koje određuju stanje i pripremi testiranje datog slučaja, a zatim izvrši test;
2. pod određenim uslovima simulira ulaz sa tastature ili obezbedi druge reakcije koje se tiču potrebnih podataka;
3. poredi stvarne odgovore sa očekivanima i izvesti o razlikama;
4. prati koje putanje su upotrebljene prilikom izvršavanja;
5. vrati prethodne vrednosti promenljivih da bi se pripremio sledeći test;
6. sarađuje sa paketom za istraživanje grešaka tako da greške, ako se to želi, mogu da se pronađu i isprave u fazi testiranja.

Automatizovana okruženja za testiranje. Alati za izvršavanje testova mogu da se integrišu sa drugim alatima radi izgradnje okruženja za sveobuhvatno testiranje. Alati koje ovde opisujemo često se povezuju sa bazom podataka za testiranje, mernim alatima, alatima za analizu koda, editorima teksta i alatima za simulaciju i modelovanje da bi se automatizovao što je moguće veći deo procesa testiranja. Na primer, baze podataka

mogu da prate slučajeve koji se testiraju, čuvaju ulazne podatke za svaki od slučajeva, opisuju očekivane odgovore i evidentiraju stvarne odgovore. Međutim, nije isto utvrditi da greška postoji i pronaći samu grešku. Testiranje će uvek zahtevati ljudski trud u traženju izvora za svaki problem; automatizacija pomaže, ali ne može da zameni ovu čovekovu funkciju.

Generatori slučajeva za testiranje

Testiranje se oslanja na pažljivo i potpuno definisanje slučajeva za testiranje. Zato je korisno automatizovati deo postupka kojim se generišu slučajevi za testiranje, da bismo bili sigurni da oni pokrivaju sve moguće situacije. Za taj posao postoji nekoliko vrsta alata. **Strukturni generatori slučajeva za testiranje** zasnivaju proces generisanja na strukturi izvornog koda. Oni obuhvataju slučajeve za testiranje putanja, grananja ili iskaza, a često uključuju i stečeno znanje koje nam pomaže da postignemo bolju pokrivenost.

Drugi generatori slučajeva se zasnivaju na analizi toka podataka, funkcionalnom testiranju (tj. na izvršavanju svih mogućih stanja koja utiču na obavljanje date funkcije), ili na stanju svake promenljive iz domena ulaza. Postoje i drugi alati za generisanje slučajnih skupova podataka za testiranje, koji se uglavnom koriste kao podrška za modelovanje pouzdanosti (kao što ćemo videti u poglavlju 9).

8.8 KADA TREBA PRESTATI SA TESTIRANJEM

U prethodnim poglavljima je već pomenuto da kvalitet softvera može da se meri na različite načine. Jedan od načina da se proceni koliko je komponenta „dobra" jeste broj grešaka koje ona sadrži. Izgleda logično pretpostaviti da se najteže ispravljaju one softverske greške koje se najteže pronalaze. Takođe deluje logično verovanje da se greške koje se najlakše ispravljaju otkrivaju na početku, pri pregledanju koda, a da se teže greške pronalaze kasnije u toku testiranja. Međutim, Shooman i Bolsky (1975) su utvrdili da to nije slučaj. Ponekad je potrebno mnogo vremena da bi se pronašle trivijalne greške i mnogi takvi problemi se previde ili dolaze na videlo tek u kasnijim fazama testiranja. Štaviše, Myers (1970) izveštava da se sa povećavanjem broja otkrivenih grešaka, takođe povećava i verovatnoća da postoje još neotkrivene greške, kao što je prikazano na slici 8.19. Ako u jednoj komponenti ima mnogo grešaka, želimo da ih pronađemo u što ranijim fazama testiranja. Međutim, grafikon pokazuje da ako na početku pronađemo veliki broj grešaka, verovatno ima mnogo i onih koje još nismo otkrili.



SLIKA 8.19 Verovatnoća pronalaženja greške tokom razvoja

Osim što su suprotni našoj intuiciji, ovi rezultati takođe otežavaju donošenje odluke kada je u procesu testiranja potrebno prekinuti traženje grešaka. Moramo izračunati broj preostalih grešaka, ne samo da bismo znali kada da prestanemo da ih tražimo, već takođe da bismo stekli izvesno poverenje u kôd koji proizvodimo. Broj grešaka takođe ukazuje na rad koji treba predvideti za održavanje koda, ako greške ostanu i otkriju se nakon isporuke.

Sejanje grešaka

Mills (1972) je razvio tehniku poznatu kao **sejanje grešaka** koja služi za izračunavanje broja grešaka u programu. Polazi se od toga da jedan član testerskog tima namerno ubaci (tj. „poseje") poznat broj grešaka u program. Ostali članovi tima nakon toga pokušavaju da pronađu što je moguće više grešaka. Broj neotkrivenih posejanih grešaka služi kao indikator za ukupan broj grešaka (uključujući izvorne, neposejane greške) koje preostaju u programu. To jest, odnos otkrivenih posejanih grešaka prema ukupnom broju posejanih grešaka trebalo bi da bude isti kao odnos otkrivenih neposejanih grešaka prema ukupnom broju neposejanih grešaka:

$$\frac{\text{otkrivene posejane greške}}{\text{sve posejane greške}} = \frac{\text{otkrive neneposejane greške}}{\text{sve neposejane greške}}$$

Prema tome, ako se u program poseje 100 grešaka, a tim za testiranje pronađe samo 70, verovatno je da je 30% izvornih grešaka preostalo u kodu.

Ovaj odnos možemo da prikažemo i formalnije. Neka S bude broj grešaka posejanih u program, a N broj grešaka koji je već postojao u programu (neposejane greške). Ako je n broj neposejanih grešaka otkrivenih tokom testiranja, a s broj posejanih grešaka otkrivenih tokom testiranja, tada bi se ukupan broj izvornih grešaka izračunao kao:

$$N = Sn/s$$

Mada je jednostavna i korisna, ova metoda polazi od pretpostavke da su posejane greške iste vrste i složenosti kao i stvarne greške u programu. Međutim, mi ne znamo kakve su tipične greške dok ih ne pronađemo, pa je teško posejati greške koje će biti reprezentativne. Jedan od načina da se poveća verovatnoća reprezentativnosti je da se vrsta posejanih grešaka zasniva na istorijskim podacima o kodu iz ranijih sličnih projekata. Međutim, to je korisno jedino ako smo ranije gradili slične sisteme. A kako smo istakli u poglavlju 2, nešto što izgleda slično može u stvari da bude sasvim različito na način kojeg nismo uvek svesni.

Da bi se prevazišla ta prepreka, možemo da koristimo dve nezavisne grupe za testiranje koje testiraju isti program. Nazvaćemo ih Grupa 1 i Grupa 2. Neka x bude broj grešaka koje otkrije Grupa 1, a y broj grešaka koje otkrije Grupa 2. Neka q bude broj grešaka koje su otkrile obe grupe, pa će biti $q \leq x$ i $q \leq y$. Na kraju, neka je n ukupan broj svih grešaka u programu; potrebno je da izračunamo koliko je n.

Efikasnost testiranja svake grupe može se meriti izračunavanjem udela pronađenih grešaka u svakoj grupi. Tako se efikasnost $E1$ Grupe 1 može izraziti kao

$$E1 = x/n$$

a efikasnost Grupe 2 kao

$$E2 = y/n$$

Efikasnost grupe je mera njene sposobnosti da pronađe greške iz datog skupa postojećih grešaka. Otuda, ako je grupa u stanju da pronađe polovinu grešaka koje postoje u programu, njena efikasnost je 0,5. Razmotrimo sada greške koje su pronašle obe grupe. Ako pretpostavimo da Grupa 1 jednako efikasno pronalazi greške u svakom delu programa, možemo da razmotrimo odnos grešaka koje je pronašla Grupa 1 u odnosu na greške koje je pronašla Grupa 2. To jest, Grupa 1 je pronašla q grešaka koje je pronašla Grupa 2, pa je efikasnost Grupe 1 jednaka q/y. Drugim rečima,

$$E1 = x/n = q/y$$

Međutim, mi znamo da je $E2 = y/n$ pa možemo da izvedemo sledeću formulu za n

$$n = q/(E1 * E2)$$

Imamo poznatu vrednost za q, možemo da izračunamo q/y za $E1$, a q/x za $E2$ pa imamo dovoljno informacija da izračunamo n.

Da bismo pokazali kako to funkcioniše, uzmimo da program testiraju dve grupe. Grupa 1 pronalazi 25 grešaka. Grupa 2 pronalazi 30 grešaka, a 15 od njih je pronašla i Grupa 1. Prema tome, imamo:

$$x = 25$$
$$y = 30$$
$$q = 15$$

Izračunavanje efikasnosti $E1$ za Grupu 1 je q/y, odnosno 0,5, pošto je Grupa 1 pronašla 15 od 30 grešaka Grupe 2. Na sličan način dobijamo da je izračunata efikasnost $E2$ za Grupu 2 jednaka q/x, ili 0,6. Prema tome, n, odnosno ukupan broj grešaka u programu, iznosi 15/(0,5 * 0,6), odnosno 50 grešaka.

Strategija testiranja definisana u planu testiranja upućuje tim kako da odluči o prestanku testiranja. Ovaj način izračunavanja može da se upotrebi u strategiji odlučivanja, po završetku testiranja.

Poverenje u softver

Izračunate greške možemo da upotrebimo za izračunavanje stepena poverenja u softver koji testiramo. **Poverenje**, koje se obično izražava u procentima, je mera verovatnoće da u softveru nema grešaka. Kada kažemo da u programu nema grešaka sa stepenom poverenja od 95%, to znači da je verovatnoća da softver nema grešaka jednaka 0,95.

Pretpostavimo da smo posejali S grešaka u softver i tvrdimo da kôd sadrži samo N stvarnih grešaka. Testiramo program dok ne nađemo svih S posejanih grešaka. Ako je n, kao i pre, broj stvarnih grešaka koje smo otkrili tokom testiranja, tada se stepen poverenja može izračunati kao

$$C\begin{cases} = 1, & \text{ako je } n > N \\ = S/(S-N+1), & \text{ako je } n \le N \end{cases}$$

Na primer, pretpostavimo da tvrdimo da jedna komponenta nema grešaka, tj. da je N jednako nuli. Ako posejemo 10 grešaka u kôd i pronađemo svih 10 a ne nađemo nijednu izvornu grešku, tada možemo da kažemo da je stepen poverenja za $S = 10$, a $N = 0$. Prema tome, C je 10/11, a stepen poverenja je 91%. Ako je zahtev u ugovoru da stepen poverenja bude 98%, morali bismo da posejemo S grešaka, gde je $S/(S - 0 + 1) = 98/100$. Kada rešimo tu jednačinu, vidimo da moramo da upotrebimo 49 posejanih grešaka i nastavimo sa testiranjem dok ne pronađemo svih 49 (i nijednu izvornu grešku).

Ovaj obrazac otvara jedan veći problem: nemoguće je predvideti stepen poverenja dok se ne pronađu sve posejane greške. Richards (1974) predlaže jednu modifikaciju, u kojoj se stepen poverenja može izračunati na osnovu otkrivenih posejanih grešaka, bez obzira na to da li su sve pronađene. U ovom slučaju, C je

$$C\begin{cases} = 1, & \text{ako je } n > N \\ = \binom{S}{s-1} / \binom{S+N+1}{N+s} & \text{ako je } n \le N \end{cases}$$

U ovim izračunavanjima, pretpostavlja se da sve greške imaju podjednaku verovatnoću da budu pronađene, što najverovatnije nije tačno. Međutim, ti faktori se uzimaju u obzir i kod mnogih drugih izračunavanja. Takve tehnike ne pružaju samo ocenu poverenja koje možemo da imamo u naše programe već od njih ima i dodatnih koristi. Mnogi programeri imaju osećaj da je svaka greška koju pronađu poslednja. Ako izračunamo broj preostalih grešaka, ili ako znamo koliko grešaka moramo da pronađemo da bismo zadovoljili zahtevani stepen poverenja, imaćemo podsticaj da nastavimo sa traženjem.

Ove tehnike su takođe korisne za izračunavanje stepena poverenja u komponente koje se spremamo da ponovo koristimo. Možemo da pregledamo istoriju grešaka te komponente, pogotovo ako je korišćeno sejanje grešaka, i ovakvim tehnikama izračunamo koliko poverenja treba pokloniti komponenti ako je upotrebimo bez dodatnog testiranja. Možemo takođe da posejemo greške u komponentu i pomoću ovih tehnika utvrdimo osnovni stepen poverenja.

Ostali kriterijumi za završetak

Kriterijumi za završetak testiranja mogu da poteknu od same strategije testiranja. Na primer, kada testiramo iskaze, putanje ili grananja, možemo da pratimo koliko iskaza, putanja ili grananja treba izvršiti i da odredimo dokle smo stigli u testiranju po broju iskaza, putanja ili grananja koja su preostala za testiranje.

Mnogi automatizovani alati mogu da izračunaju vrednosti pokrivenosti. Pogledajte sledeći kôd iz knjige Lee i Tepfenhatr (1997) kojim je implementirana računarska igrica:

```
LISTING                                              GRANA BROJ
                                                      ISKAZA
void                                                          1
Collision::moveBall(Ball *ball)                               2
{                                                             3
        ball->change?position(final?loc(),upperLeft());       4
        int sf = 1; // faktor brzine                          5
          for(int i = 0; i<number?hit(); i++)          1-2    6
```

```
        {                                                    7
          Obstacle *hitptr=                                  8
            (Obstacle *) obj(i)->real_identity();            9
          sf *= hitptr->respond_to_being_hit(this);         10
        }                                                   11
      Point v = rebound(ball->get_velocity());              12
      if(v.X() == 0 )                          3-4          13
        v.X(1);                                             14
      if(v.Y() == 0 )                          5- 6         15
        v.Y(-1);                                            16
      ball->change_velocity(sf*v);                          17
}                                                           18
```

Kao što vidite, alatka može na listingu da označi gde se nalaze grananja. Prema tome, grana 1 je putanja kojom se ide ako je *i* unutar parametara petlje u iskazu 6, grana 2 je putanja kojom se ide ako *i* nije unutar parametara petlje, grana 3 je putanja kojom se ide kada je *v.X* jednako nuli u iskazu 13, grana 4 je putanja kojom se ide ako *v.X* nije nula, i tako dalje. Automatizovani alat može da izračuna sve putanje koje treba pokriti testiranjem; u ovom slučaju ima ih 2^3, odnosno 8. Zatim, kako testiranje napreduje, alat može da proizvede izveštaj prikazan u tabelama 8.8 i 8.9, te se može uočiti koliko putanja je ostalo da se pređe da bismo imali pokrivenost putanja ili grananja.

Otkrivanje koda sklonog greškama

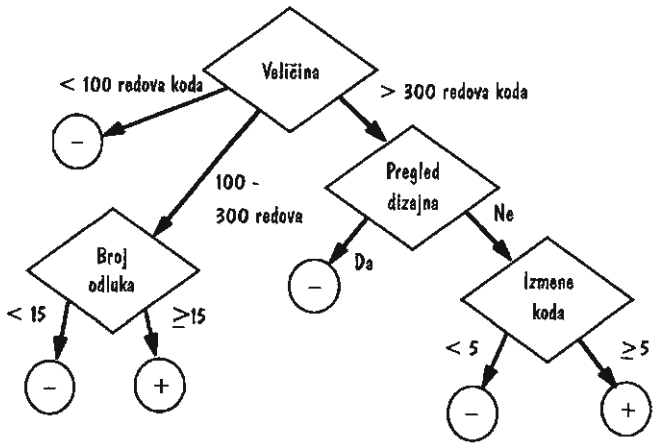
Mnoge tehnike mogu da se koriste za prepoznavanje koda sklonog greškama na osnovu istorije grešaka u sličnim aplikacijama. Na primer, neki istraživači prate broj grešaka pronađenih u svakoj komponenti tokom razvoja i održavanja. Oni takođe o svakoj komponenti prikupljaju mere kao što su veličina, broj odluka, broj operatora i operanada ili broj modifikacija. Zatim na osnovu toga izračunavaju kakve osobine imaju komponente koje su najviše sklone greškama. To može da posluži za određivanje redosleda kojim će se komponente testirati ili za izbor komponenti kojima treba pokloniti posebnu pažnju tokom pregleda ili testiranja.

Tabela 8.8 Rekapitulacija prolaska putanjama

		Ovaj test:			Kumulativno		
Slučaj	Broj putanja	Pozivanje	Pređene putanje	% pokrivenosti	Pozivanje	Pređene putanje	% pokrivenosti
6	8	1	4	50	5	6	75

Tabela 8.9 Putanje koje nisu izvršene

Slučaj	Propuštene putanje	Ukupno
6	1 2 3 4	4



SLIKA 8.20 Klasifikaciono stablo za prepoznavanje komponenti sklonih greškama

Porter i Selbi (1990) predlažu upotrebu klasifikacionog stabla za prepoznavanje komponenti sklonih greškama. Analiza pomoću klasifikacionog stabla je statistička tehnika kojom se uređuju veliki nizovi merenih informacija, čime se dobija stablo odlučivanja koje pokazuje koja merenja su najbolja za predviđanje konkretnog atributa. Na primer, pretpostavite da prikupljamo podatke o svakoj komponenti u našoj organizaciji. Evidentiramo veličinu (izraženu brojem redova koda), broj različitih putanja kroz kôd, broj operatora, dubinu ugnežđenosti, spregnutost i koheziju (na skali od 1 za najnižu do 5 za najvišu), vreme utrošeno na kodiranje komponente, broj grešaka pronađenih u komponenti i slično. Analizom stabla klasifikacije obradimo atribute komponenti koje su imale pet ili više grešaka, i poredimo ih sa onima koje su imale manje od pet grešaka. Rezultat bi moglo da bude stablo odlučivanja prikazano na slici 8.20.

Stablo koristimo da bismo izračunali koje komponente iz našeg trenutnog sistema verovatno imaju veliki broj grešaka. Prema ovom stablu, ako komponenta sadrži između 100 i 300 redova koda i bar 15 odluka, ona bi mogla da bude sklona greškama. Ili ako komponenta sadrži preko 300 redova koda i nije za nju održan pregled dizajna a menjana je bar pet puta, takođe može da bude sklona greškama. Ovu vrstu analize možemo da koristimo za usmeravanje testiranja kada imamo ograničene resurse za testiranje. Možemo takođe da zakažemo inspekcije za takve komponente da bismo pronašli probleme pre nego što testiranje počne.

8.9 PRIMER IZ INFORMACIONIH SISTEMA

Pretpostavite da pravimo slučajeve za testiranje komponenti sistema Piccadilly i da smo izabrali strategiju testiranja u kojoj se planira izvršavanje svih putanja u komponenti. Mogli bismo da napišemo skriptove testa sa opisom ulaza i očekivanog izlaza, a proces testiranja bi se sastojao od poređenja stvarnog izlaza sa očekivanim za svaki slučaj za testiranje. Ako stvarni izlaz bude jednak očekivanom, da li to znači da komponenta ne

sadrži greške? Ne sasvim. Možda smo u komponenti naišli na ono što Beizer (1990) naziva slučajnom ispravnošću komponente. Da bismo to razumeli, pogledajmo komponentu sa sledećom strukturom:

```
CASE 1: Y := X/3;
CASE 2: Y := 2X-25;
CASE 3: Y := X MOD 10;
ENDCASE;
```

Ako naš slučaj za testiranje kao ulaz za X koristi 15, očekivana vrednost Y kao izlaza je 5 i stvarno dobijena vrednost za Y je 5, pri čemu ne znamo koja grana je izvršena; svaki obrazac će za X jednako 15 dati Y jednako 5! Zbog toga mora svaki slučaj da se dopuni markerima koji će timu pomoći da prepozna kojom putanjom je kôd izvršen. U ovom primeru bi alat za analizu pokrivenosti putanja bio veoma koristan za praćenje iskaza koji su izvršeni prilikom testiranja svakog slučaja.

Pošto je sistem Piccadilly informacioni sistem, možda ćemo umesto strukturne strategije radije upotrebiti strategiju toka podataka. Pomoću rečnika podataka možemo lako da izdvojimo svaki podatak i zatim razmotrimo vrednost svakog od njih. Najbolje će odgovarati strategije kao što su definicija-upotreba; pratimo svaki podatak kroz komponentu i tražimo situacije u kojima može da mu se promeni vrednost i pri tom kontrolišemo da li se promena izvršila ispravno. Postoje mnogi automatizovani alati za podršku takvom testiranju: baze podataka, generatori slučajeva za testiranje i monitori izvršavanja testova koji beleže svaku promenu vrednosti podatka. U stvari, testerski tim će možda poželeti da poveže bazu podataka koja sadrži rečnik podataka sa drugim alatima.

8.10 PRIMER IZ PRAKSE

Sistem Ariane-5 podvrgnut je mnogim pregledima i testiranju. Lions i dr. (1996) izveštavaju da je sistem kontrole leta testiran na četiri načina:

1. testiranje opreme;
2. testiranje softvera za računar na letelici;
3. prikaz integracije;
4. testiranja valjanosti sistema.

Osnovna filozofija na kojoj je počivalo testiranje Ariane-5 bila je da se na svakom nivou proveri ono što na prethodnom nije postignuto. Projektanti su smatrali da će na taj način postići potpunu pokrivenost svakog podsistema i integrisanog sistema. Pogledajmo istragu nakon eksplozije, da bismo videli zašto testiranje prilikom provere softvera nije otkrilo probleme u SRI-ju pre samog leta. (U poglavlju 9 ćemo razmotriti testove integracije i validacije.)

Istražitelji su izvestili da „nije izvršen test koji bi proverio da će se SRI pravilno ponašati u vremenskimm rokovima predviđenim za odbrojavanje i let na putanji Ariane-5" (Lions i dr. 1996). U stvari, specifikacija za SRI softver nije u funkcionalnim zahtevima sadržala podatke o putanji Ariane-5. Drugim rečima, u dokumentima zahteva nije opisano na koji način će se putanja Ariane-5 razlikovati od putanje Ariane-4. Istražitelji su zabeležili da: „Tako iskazana ograničenja, koja bi trebalo da budu obavezna za svaki

uređaj od bitnog značaja za misiju, poslužila bi da se uoči neusklađenost sa putanjom Ariane-5".

Pošto je osnovni uzrok u zahtevima, on je mogao biti uočen u ranim fazama procesa razvoja. Zaista, pregledi su bili sastavni deo postupka dizajna i kodiranja, a istražitelji ističu da su oni „održavani na svim nivoima i uključivali sve glavne partnere u projektu (kao i stručnjake iz spoljne saradnje)". Zaključili su da

> „... je očigledno da ograničenja SRI softvera nisu potpuno proučena na pregledima i da nije shvaćeno da pokrivenost testiranjem nije dovoljna da bi se ta ograničenja otkrila. Takođe nisu shvaćene sve moguće posledice u slučaju da se dozvoli da softver za poravnavanje nastavi sa radom tokom leta. U tim aspektima, postupak pregleda je doprineo nesreći." (Lions i dr. 1996)

Znači, projektanti Ariane-5 su se oslanjali na nedovoljne preglede i pokrivenost testiranjem, i imali su neopravdano poverenje u softver.

Postoji više načina da se poveća verovatnoća da će pregledi i pokrivenost testiranjem biti potpuni. Jedan je da se u preglede uključi neko ko nije stručnjak. Takav učesnik će postaviti mnoga pitanja za koja ostali učesnici, često neopravdano, smatraju da se podrazumevaju. Drugi način je da se ispita potpunost slučajeva, bilo da se pozove saradnik spolja da ih pregleda, bilo pomoću neke formalne tehnike za izračunavanje stepena pokrivenosti. Kao što ćemo videti u poglavlju 9, projektanti Ariane-5 imali su još nekoliko mogućnosti da otkriju problem sa SRI-jem za vreme testiranja, ali se on provukao kroz njihovu bezbednosnu mrežu.

8.11 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju opisane su mnoge tehnike koje se mogu upotrebiti za jedinično testiranje kodiranih komponenti i testiranje njihove integracije sa komponentama koje su drugi izradili. Važno je shvatiti razliku između greške (problema sa zahtevima, dizajnom, kodom, dokumentacijom ili slučajevima za testiranje) i otkaza (problema u funkcionisanju sistema). Testiranjem se traže greške, ponekad tako što se kôd navede na otkaz, a zatim se traži razlog. Jedinično testiranje je razvojna aktivnost u kojoj se svaka komponenta izvršava zasebno; integraciono testiranje sastavlja komponente na organizovan način da bi se lakše pronašle greške kada se kombinovane komponente testiraju zajedno.

Cilj testiranja je da se pronađu greške, a ne da se dokaže ispravan rad. Istina je da odsustvo grešaka ne garantuje ispravnost. Postoje mnoge ručne i automatizovane tehnike koje pomažu da se pronađu greške u kodu, kao i alati za testiranje koji prikazuju deo koji je već testiran i kada se testiranje može prekinuti.

8.12 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Testiranje je i samostalna i grupna aktivnost. Pošto se komponenta napiše, ona može da se podvrgne inspekciji od strane celog razvojnog tima ili njegovog dela u cilju traženja grešaka koje nije uočio programer. Istraživačka literatura jasno pokazuje da su inspekcije veoma delotvorne u pronalaženju grešaka u ranim fazama razvoja. Takođe je jasno da druge tehnike za traženje grešaka pronalaze greške koje često promaknu inspekciji. Zato

je važno da se radi sa timom na bezličan način i upotrebi što više raspoloživih metoda u cilju pronalaženja grešaka u što je moguće ranijim fazama razvoja.

Integraciono testiranje je takođe timska aktivnost i zahteva izbor strategije integracije, planiranje testiranja, generisanje slučajeva za testiranje i izvršavanje testova u saradnji sa ostalim članovima. Automatizovani alati su korisni u ovim aktivnostima jer pomažu da pažljivo proučite rezultate testiranja i uočite probleme i njihove uzroke.

8.13 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Istraživači i dalje ispituju veliki broj važnih pitanja u vezi sa testiranjem:

- Inspekcije su delotvorne, ali bi se mogle učiniti još delotvornijima. Istraživači traže najbolje načine za izbor članova inspekcijskog tima, za pregledanje razvojnih proizvoda i za saradnju na grupnim sastancima kako bi se pronašlo što je moguće više grešaka. Neki istraživači porede korišćenje grupnih sastanaka sa radom bez sastanaka, da bi utvrdili njihov stvarni doprinos.
- Istraživači i dalje pokušavaju da shvate koje su tehnike najbolje za pronalaženje određenih vrsta grešaka.
- Sistemi koje gradimo daleko su složeniji i mnogo veći od sistema koji su građeni samo pre nekoliko godina. Zato je sve važnije da se imaju automatizovani alati koji podržavaju proces testiranja. Istraživači traže načine za definisanje slučajeva za testiranje i za izračunavanje potpunosti pokrivanja, kao i ulogu automatizacije u tim aktivnostima.
- Resursi za testiranje su obično ograničeni zbog rokova isporuke, pogotovo za proizvode namenjene tržištu. Istraživači i dalje traže načine za prepoznavanje komponenti koje su sklone greškama, da bi se testiranje prvo usmerilo na njih. Slično tome, za sisteme u kojima je bezbednost značajna, istraživači grade modele i alate kojima se obezbeđuje da najkritičnije komponente budu potpuno testirane.

8.14 SEMINARSKI RAD

Aranžer pozajmica zahteva intenzivno testiranje. Pošto je sistem značajan za „ekonomsko zdravlje" FCO, kupac želi da se softver isporuči što je moguće pre. Smislite strategiju testiranja za *Aranžer pozajmica* koja će angažovati što manje resursa. Obrazložite strategiju i objasnite kako ćete znati kada da prekinete testiranje i predate sistem kupcu.

8.15 KLJUČNE REFERENCE

Testiranje je bilo tema mnogih specijalnih izdanja žurnala i časopisa, uključujući *IEEE Software* za mart 1991. i *Communications of the ACM* za jun 1988. *Communications of the ACM* za septembar 1994. opisuje posebne aspekte testiranja objektno orijentisanih sistema. Dvobroj *IEEE Software* za januar/februar 2000. bavi se pitanjem zašto je testiranje tako teško. Osim toga, *IEEE Transactions on Software Engineering* često sadrži članke

u kojima se porede različite tehnike testiranja, zavisno od vrste grešaka koje najbolje otkrivaju.

Postoji više dobrih knjiga u kojima se detaljno opisuje testiranje. Myers (1979) je klasičan tekst, koji opisuje filozofiju testiranja, kao i nekoliko specifičnih tehnika. Beizer (1984) nudi dobar opšti pregled problema i tehnika testiranja, sa mnogim referencama na ključne referate iz te oblasti. Njegova knjiga iz 1995. posebno se usredsređuje na testiranje po modelu crne kutije. Hetzel (1984) je takođe korisna referenca, kao i Perry (1995), Kit (1995) i Kaner, Faulk i Nguyen (1993). Binder (2000) je sveobuhvatan vodič za testiranje objektno orijentisanih sistema.

Postoji više dobrih referata u kojima se opisuje upotreba inspekcija, uključujući Weller (1993 i 1994) i Grady i van Slack (1994). Knjiga Gilba i Grahamove (1993) o inspekcijama je dobar, sveobuhvatan i praktičan vodič. Istraživači i dalje doteruju tehnike inspekcije i načine da se one prošire na druge proizvode u procesu, recimo na zahteve. Primere iz te oblasti možete naći u knjigama koje su objavili Porter i dr. (1998) i Shull i dr. (2000).

Postoje mnogi automatizovani alati za testiranje koje možete da primenite na svoje programe. Software Quality Engineering iz Džeksonvila na Floridi svake godine objavljuje vodič *Testing Tools Reference Guide*. Informacije o konkretnim alatima mogu se naći na veb adresama prodavaca, kao što je Cigital ili Rational Software. Lokacija Cigital takođe sadrži bazu podataka sa resursima za testiranje. U knjizi Fewster i Graham (1999) opisuju se pitanja u vezi sa automatizacijom testiranja softvera.

Software Research razvija i prodaje alate za testiranje. Oni takođe organizuju konferenciju po imenu Quality Week svakog meseca maja u San Francisku, a svake jeseni u Evropi, na kojoj učesnici izveštavaju o iskustvima u oblasti testiranja.

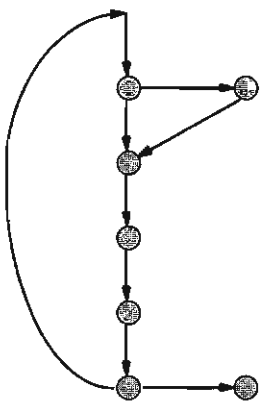
IEEE Computer Society i ACM sponzorišu još neke konferencije iz oblasti testiranja i o njima se možete informisati na njihovim veb adresama.

Osim toga, u Vašingtonu postoji međunarodna konferencija i izložba o testiranju računarskog softvera. Svake godine je tema neki aspekt testiranja, kao što je automatizacija testiranja ili testiranje pod pritiskom. Za više informacija možete se obratiti na ginger@fron-tech.com (G. Houston-Ludlam).

8.16 VEŽBE

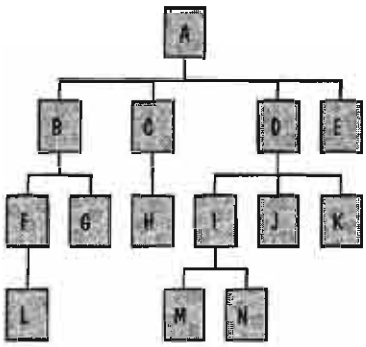
1. Pregledajte kategorije grešaka u klasifikacionoj šemi za Hewlett-Packard, prikazanoj na slici 8.1. Da li je ta klasifikacija ortogonalna? Ako nije, objasnite zašto i predložite kako bi mogla da postane ortogonalna.
2. Neka je P programska komponenta koja učitava listu od N zapisa i neka je dat uslov raspona za ključ zapisa. Prvih sedam znakova u zapisu čine ključ. P čita ključ i pravi izlaznu datoteku koja sadrži samo one zapise čiji ključ pripada propisanom rasponu. Na primer, ako je raspon od „JONES" do „SMITH", tada će se izlazna datoteka sastojati od onih zapisa koji se abecedno nalaze između „JONES" i „SMITH". Napišite ulazne i izlazne uslove u obliku tvrdnji koje će se upotrebiti za dokazivanje da je P ispravan. Napišite odgovarajući dijagram toka za P i označite tačke transformacije.
3. Dovršite proveru primera iz teksta koji ilustruje slika 8.5. Drugim rečima, napišite tvrdnje koje odgovaraju dijagramu toka. Zatim pronađite putanje od ulaznih uslova do izlaza. Dokažite da su te putanje teoreme.

4. Pretpostavite da program sadrži N tačaka odlučivanja od kojih svaka ima dve grane. Koliko je potrebno slučajeva za testiranje da bi se u takvom programu izvršilo testiranje putanja? Ako u svakoj tački odlučivanja postoji M mogućnosti, koliko je potrebno slučajeva za testiranje putanja? Može li se taj broj smanjiti promenom strukture programa? Navedite primer koji dokazuje vaš odgovor.
5. Razmotrite dijagram toka programa kao usmereni graf u kojem su rombovi i pravougaonici programa čvorovi, a strelice logičkog toka među njima usmerene grane. Na primer, program sa slike 8.7 može se prikazati grafom prikazanim na slici 8.21. Dokažite da je testiranje iskaza u programu ekvivalentno pronalaženju one putanje u grafu koja sadrži sve čvorove grafa. Dokažite da je testiranje grananja ekvivalentno pronalaženju skupa putanja čija unija pokriva sve grane grafa. Na kraju, dokažite da je testiranje putanja ekvivalentno pronalaženju svih mogućih putanja kroz graf.
6. Problem koji se može programirati: Napišite program koji kao ulaz prihvata čvorove i grane usmerenog grafa i kao rezultat štampa sve moguće putanje kroz graf. Koja su glavni problemi u vezi sa dizajnom ovog programa? Na koji način složenost grafa (u smislu broja grana i ciklusa) utiče na algoritam koji koristite?



SLIKA 8.21 Graf za program sa slike 8.7

7. Na slici 8.22 prikazana je hijerarhija komponenti u softverskom sistemu. Opišite redosled testiranja za integrisanje komponenti odozdo nagore, odozgo nadole, promenjenim redosledom odozgo nadole, pristupom pa kud puklo, pristupom sendvič i promenjenim sendvič pristupom.



SLIKA 8.22 Primer hijerarhije komponenti

8. Objasnite zašto se graf sa slike 8.19 može protumačiti kao savet da u slučaju da pronađete mnogo grešaka u kodu u vreme prevođenja, treba da bacite kôd i ponovo ga napišete.
9. Koja su neka od mogućih objašnjenja za ponašanje grafa sa slike 8.19?
10. U program je posejano 25 grešaka. Tokom testiranja otkriveno je 18 grešaka od kojih je 13 posejanih a 5 izvornih grešaka. Kolika je Millsova vrednost za broj izvornih grešaka koje još nisu otkrivene u programu?
11. Vi tvrdite da vaš program nema grešaka sa stepenom poverenja od 95%. Vaš plan testiranja zahteva da testirate dok ne pronađete sve posejane greške. Koliko grešaka morate da posejete u program pre testiranja da biste dokazali svoje tvrdnje? Ako iz nekog razloga ne nameravate da otkrijete sve posejane greške, koliko posejanih grešaka je potrebno za Richardsov obrazac?
12. Opišite razlike u testiranju sistema koji je značajan za poslovanje, sistema koji je značajan za bezbednost i sistema čije otkazivanje ne bi ozbiljno ugrozilo ničiji život, zdravlje, niti poslovanje.
13. Navedite primer objektno orijentisanog sistema u kojem problemi sinhronizacije zahtevaju pažljivo testiranje.
14. Ako je integraciono testiranje obavio nezavisan tim za testiranje, a u kodu je po završenom testiranju ostala ozbiljna greška, ko je zakonski, a ko moralno odgovoran za štetu nastalu zbog te greške?
15. Pretpostavite da gradite sistem za pripremu poreskih prijava koji se sastoji od tri komponente. Prva komponenta formira obrasce na ekranu u koje korisnik upisuje ime, adresu, poreski broj i finansijske informacije. Druga komponenta pomoću tabele za porez i ulaznih informacija izračunava iznos poreza za tekuću godinu. Treća komponenta koristi informacije o adresi za štampanje obrazaca za savezni, državni (ili oblasni) i gradski porez, uključujući i iznos zaduženja. Opišite strategiju koju biste upotrebili za testiranje ovog sistema i okvirno navedite slučajeve u planu testiranja.

9 Testiranje sistema



U ovom poglavlju obradivačemo:
- funkcionalno testiranje;
- testiranje performanse;
- završno testiranje prihvatljivosti;
- pouzdanost, dostupnost i mogućnost održavanja softvera;
- instalacioni test;
- dokumentovanost testiranja;
- testiranje bezbednosno kritičnih sistema.

Testiranje sistema se veoma razlikuje od jediničnog testiranja i integrativnog testiranja. Kada komponente testirate jedinično, vi imate potpunu kontrolu nad procesom testiranja. Sami pravite podatke za test, projektujete vlastite test slučajeve i sami obavljate testiranje. Kada integrišete komponente, ponekad radite sami, ali često sarađujete sa nekim članovima tima za testiranje ili tima za razvoj. Međutim, kada se testira sistem, vi radite sa celim razvojnim timom - koordinirate svoje postupke i nalazite se pod komandom rukovodioca tima. U ovom poglavlju razmotrićemo proces testiranja sistema: njegovu svrhu, korake, učesnike, tehnike i alate.

9.1 PRINCIPI TESTIRANJA SISTEMA

Cilj jediničnog i integrativnog testiranja bio je da se proveri da li kôd pravilno implementira dizajn, to jest, da li su programeri napisali kôd koji radi ono što žele projektanti. Kada je reč o testiranju sistema, cilj je sasvim drugačiji: proveravamo da li sistem radi ono što želi kupac. Da bismo razumeli kako da postignemo taj cilj, najpre moramo znati odakle potiču greške u sistemu.

Izvori softverskih grešaka

Setite se da softverska greška dovodi do otkaza samo ako se steknu odgovarajući uslovi. To jest, u kodu može da postoji greška, ali ako se on nikada ne izvršava ili ako se ne izvršava dovoljno dugo ili u odgovarajućim uslovima da bi doveo do problema, možda se nikada neće dogoditi da on otkaže. Pošto testiranjem ne možemo da ostvarimo sve moguće uslove, cilj treba da nam bude otkrivanje grešaka u nadi da ćemo tim putem

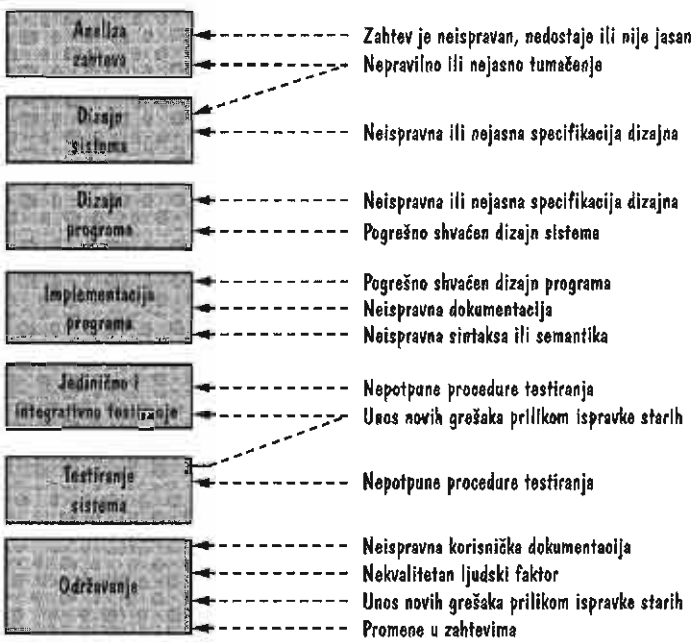
ukloniti sve greške koje bi mogle da dovedu do otkazivanja sistema prilikom njegove stvarne upotrebe.

Softverske greške mogu da nastanu u zahtevima, dizajnu ili kodu komponente ili u dokumentaciji u bilo koje vreme tokom razvoja ili održavanja. Na slici 9.1 prikazani su verovatni uzroci grešaka svake aktivnosti koju obuhvata razvoj sistema. Mada uvek želimo da pronađemo i ispravimo greške što je moguće pre, testiranjem sistema priznajemo da je prisustvo grešaka moguće i nakon integrativnog testiranja.

Greške mogu da se unesu u sistem u ranim fazama razvoja ili kasnije kada se ispravlja novootkrivena greška. Na primer, neispravan softver može da bude rezultat grešaka u zahtevima. Bio zahtev dvosmislen zato što kupac nije bio siguran šta mu treba ili zato što smo mi pogrešno shvatili šta je on mislio, rezultat je isti: sistem koji ne radi onako kako to kupac želi.

Do iste takve neuspele komunikacije može doći i tokom dizajniranja sistema. Možemo pogrešno da protumačimo neki zahtev pa napišemo pogrešnu specifikaciju dizajna. Ili da dobro razumemo zahtev, ali tako loše formulišemo specifikaciju da je pogrešno shvate oni koji kasnije čitaju i koriste taj dizajn. Slično tome, možda mi, za razliku od ostalih koji čitaju naš dizajn, podrazumevamo neke karakteristike ili relacije.

Na sličan način može doći i do grešaka u dizajnu programa. Do pogrešnog tumačenja često dolazi kada se dizajn sistema prevodi u opise nižeg nivoa za specifikacije programa. Programeri ne učestvuju u početnim razgovorima sa kupcima kada se raspravlja o ciljevima i funkcionalnosti sistema. Pošto su programeri zaduženi za jedno „drvo", a ne za „šumu", od njih se ne može očekivati da uoče greške u dizajnu koje su prenete iz prvih koraka ciklusa razvoja. Iz tog razloga su pregledi zahteva i dizajna bitni za osiguranje kvaliteta dobijenog sistema.



SLIKA 9.1 Uzroci grešaka tokom razvoja

Isto tako, programeri i projektanti u razvojnom timu možda ne koriste pravilnu sintaksu i semantiku za evidentiranje svog rada. Prevodilac ili asembler može da otkrije neke od tih grešaka pre nego što se program izvrši, ali ne može da otkrije greške ako je forma iskaza ispravna, ali ne odgovara namerama programera ili projektanta.

Kada počne testiranje programske komponente, greške mogu nenamerno da se dodaju prilikom otklanjanja drugih problema. Takve greške se često veoma teško otkrivaju jer se manifestuju samo kada se vrše određene funkcije ili samo pod određenim uslovima. Ako su te funkcije već testirane pre nego što je nenamerno uneta nova greška, ona će se primetiti tek mnogo kasnije, a tada njen uzrok neće biti jasan. Do te situacije može doći ako ponovo koristimo kôd iz drugih aplikacija, pa ga menjamo prema svojim trenutnim potrebama. Nijanse u dizajnu koda možda nisu sasvim jasne i naša izmena možda više škodi nego što koristi.

Na primer, pretpostavite da testirate komponente A, B i C. Prvo testirate svaku od njih zasebno. Kada testirate sve tri komponente zajedno, otkrivate da A na neispravan način predaje parametar komponenti C. Prilikom prepravljanja A vi sada vodite računa da predavanje parametra bude ispravno, ali pri tom dodate kôd u kojem pogrešno podešavate jedan pokazivač. Pošto se niste vratili da ponovo jedinično testirate A, vi nećete uočiti tu novu grešku sve do kasnijih faza testiranja, a tada neće biti jasno da je krivica do A.

Isto tako, nove greške mogu da se unesu prilikom održavanja. Unapređenja sistema ponekad zahtevaju da se promene zahtevi, arhitektura sistema, dizajn programa pa i sama implementacija. Zato može doći do raznovrsnih grešaka dok se unapređenje opisuje, dizajnira i kodira. Osim toga, sistem možda ne funkcioniše pravilno zato što korisnici ne shvataju kako je zamišljeno da sistem radi. Do greške može doći ako je dokumentacija nejasna ili neispravna. Ljudski faktor, uključujući i shvatanje korisnika, ima veliku ulogu u razumevanju sistema i tumačenju njegovih poruka i zahtevanog ulaza. Korisnici koji nisu navikli na sistem, možda pogrešno koriste njegove funkcije ili ih ne koriste u potpunosti.

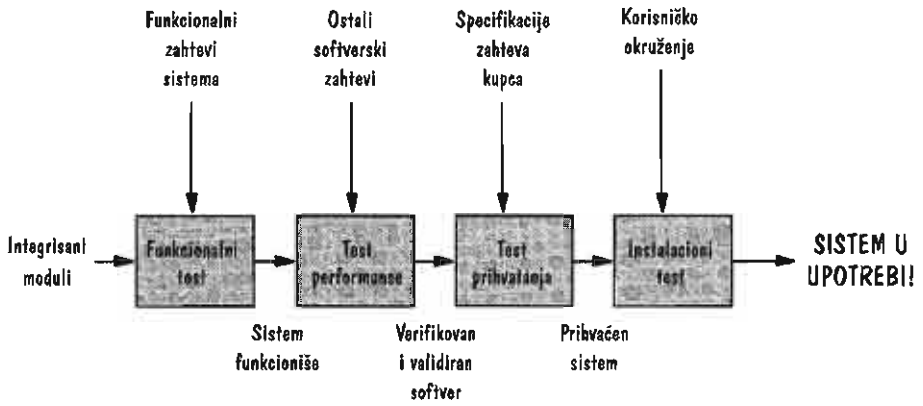
Procedura testiranja trebalo bi da bude dovoljno temeljita da izvrši sve sistemske funkcije na zadovoljstvo svih: korisnika, kupca i učesnika u razvoju. Ako testovi nisu potpuni, greške mogu da ostanu neotkrivene. Kao što smo videli, bolje je da što pre otkrijemo grešku; rano otkrivene greške se lakše i jeftinije otklanjaju. Zato će potpuno i pravovremeno testiranje pomoći ne samo da se greške brže otkriju, nego i da se lakše otkriju njihovi uzroci.

Na slici 9.1 prikazani su uzroci grešaka, a ne mesto na kojem su se manifestovale. Pošto je testiranje usmereno na pronalaženje što je moguće većeg broja grešaka, važno je znati gde se one nalaze. Ako znamo kako greške nastaju, bolje ćemo shvatiti gde da ih tražimo kada testiramo sistem.

Proces testiranja sistema

Testiranje sistema se sastoji od nekoliko koraka:

1. funkcionalno testiranje;
2. testiranje performanse;
3. testiranje prihvatanja;
4. instalaciono testiranje.



SLIKA 9.2 Koraci u postupku testiranja

Koraci su prikazani na slici 9.2. Svaki korak ima različit fokus, a njegov uspeh zavisi od postavljenog cilja. Zato je važno da se analizira svrha svakog koraka u testiranju sistema.

Ciljevi procesa. Na početku testiramo funkcije sistema. Počinjemo sa jednim skupom komponenti koje su prvo jedinično, a zatim zajedno testirane. **Funkcionalnim testom** se proverava da li integrisani sistem obavlja svoje funkcije onako kako je navedeno u zahtevima. Na primer, u funkcionalnom testu paketa za bankarsko poslovanje proverava se da li paket može pravilno da odobrava i zadužuje, evidentira podizanje, izračunava kamatu, štampa saldo itd.

Pošto se tim za testiranje uveri da funkcije rade na traženi način, **testom performanse** se integrisane komponente proveravaju prema nefunkcionalnim zahtevima sistema. Ovi zahtevi u koje spadaju bezbednost, preciznost, brzina i pouzdanost postavljaju okvire u kojima se moraju obavljati funkcije sistema. Na primer, test performanse za paket bankarskog poslovanja procenjuje brzinu kojom se vrši izračunavanje, preciznost izračunavanja, kakve su predostrožnosti u smislu bezbednosti i kakvo je vreme odziva na korisnički upit.

U ovoj fazi sistem radi onako kako su zamislili projektanti. Sada imamo **verifikovan sistem**; na način kako su projektanti protumačili specifikaciju zahteva. Nakon toga poredimo sistem sa očekivanjima kupca tako što pregledamo dokument specifikacije zahteva. Ako se uverimo da sistem koji smo izgradili zadovoljava zahteve, znači da imamo **validiran sistem**; to jest, dokazali smo da su zahtevi ispunjeni.

Do sada su sva testiranja obavljali učesnici u razvoju na osnovu njihovog shvatanja sistema i njegovih ciljeva. Kupci takođe testiraju sistem i proveravaju da li on zadovoljava njihovo shvatanje zahteva, koje može da se razlikuje od shvatanja učesnika u razvoju. Ovaj test se zove **test prihvatanja** i on uverava kupce da je sistem koji je izgrađen za njih zaista onaj koji su tražili. Test prihvatanja se ponekad vrši na samoj lokaciji, ali to nije uvek slučaj, pa se zato ponekad radi i konačni **instalacioni test** da bi korisnici isprobali funkcije sistema i dokumentovali dodatne probleme do kojih dolazi na samoj lokaciji. Na primer, sistem za plovidbu može da se projektuje, izgradi i testira u prostorijama isporu-

čioca konfigurisanim po uzoru na brod, ali ne na samom brodu. Pošto se završi testiranje u prostorijama isporučioca, može se izvršiti niz instalacionih testova na svakom tipu broda na kojem će se sistem koristiti.

Plan izgradnje ili integracije. Idealno je da se nakon testiranja programa kolekcija komponenti posmatra kao celina. Onda se u prvim koracima testiranja sistema integrisana kolekcija ocenjuje iz različitih perspektiva, kao što smo upravo opisali. Međutim, veliki sistemi su ponekad nezgrapni da bi se testirali kao jedna džinovska kolekcija komponenti. U stvari, takvi sistemi su često kandidati za razvoj u fazama, jednostavno zato što je mnogo lakše graditi i testirati manje celine. Zato ćete se možda odlučiti za testiranje sistema u fazama. U poglavlju 1 smo videli da se sistem može posmatrati kao ugnežđeni skup nivoa ili podsistema. Svaki nivo je zadužen za izvršavanje bar jedne funkcije iz podsistema koji ga sačinjavaju. Slično tome, možemo i test sistema da podelimo na ugnežđeni niz podsistema i da sistem testiramo na svakom podsistemu.

Definicije podsistema zasnivaju se na kriterijumima koji su prethodno određeni. Obično se podela vrši na osnovu funkcionalnosti. Na primer, videli smo u poglavlju 8 da Microsoft deli proizvod na tri podsistema, i to na podsistem esencijalnih funkcija, podsistem poželjnih funkcija i na podsistem manje potrebnih funkcija. Slično tome, telekomunikacioni sistem koji usmerava pozive može se podeliti na sledeće podsisteme:

1. usmeravanje poziva unutar iste centrale;
2. usmeravanje poziva u istom pozivnom broju;
3. usmeravanje poziva unutar države ili provincije;
4. usmeravanje poziva unutar zemlje;
5. usmeravanje međunarodnih poziva.

Svaki veći podsistem sadrži sve prethodne podsisteme. Testiranje sistema započinjemo testiranjem funkcija prvog podsistema. Kada se uspešno istestiraju sve funkcije unutar iste telefonske centrale, prelazimo na testiranje drugog podsistema. Na sličan način testiramo treći, četvrti i peti podsistem. Na kraju imamo uspešan test celog sistema, ali zbog postepenosti postupka mnogo su se lakše otkrivale i ispravljale greške nego da smo odmah prešli na najširi sistem. Na primer, problem otkriven tokom funkcionalnog testiranja poziva unutar države ili provincije verovatno se nalazi u kodu koji obrađuje državu, ali ne i isti pozivni broj niti istu centralu. Prema tome, možemo da tražimo uzrok greške u podsistemu 3, ili u delu koda podsistema 1 ili 2 na koje utiče podsistem 3. Da je ovaj problem otkriven tek kad su integrisani svi podsistemi, ne bismo tako lako odredili najverovatniji uzrok greške.

Postepeno testiranje zahteva pažljivo planiranje. Tim za testiranje mora da napravi **plan izgradnje** ili **integracije** u kome će definisati podsisteme koje treba testirati i opisati kako, gde, kada i ko će izvršiti testiranje. U planu izgradnje moraju se obuhvatiti mnoga pitanja koja smo pomenuli u integracionom testiranju, uključujući i redosled integracije kao i potrebu za lažnim rutinama i rukovaocima.

Ponekad se u planu izgradnje jedan nivo ili podsistem naziva **rotacija**. Rotacije se numerišu tako da se najniži nivo naziva **nulta rotacija**. Za veće sisteme nulta rotacija često predstavlja minimalni sistem; ponekad čak samo operativni sistem na glavnom računaru.

Tabela 9.1 Izgradnja plana za telekomunikacioni sistem

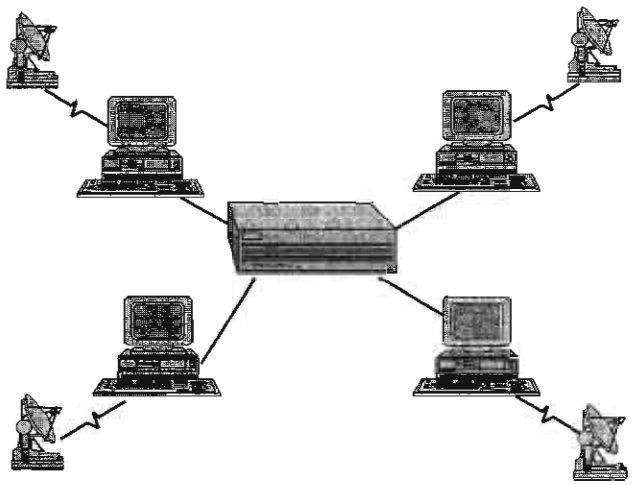
Rotacija	Funkcije	Početak testiranja	Kraj testiranja
0	Telefonska centrala	1. septembar	15. septembar
1	Pozivni broj	30. septembar	15. oktobar
2	Država/provincija	25. oktobar	5. novembar
3	Zemlja	10. novembar	20. novembar
4	Međunarodno	1. decembar	15. decembar

Na primer, plan izgradnje za telekomunikacioni sistem može da sadrži raspored sličan onome u tabeli 9.1. U planu izgradnje se svaka rotacija opisuje svojim brojem, funkcionalnim sadržajem i vremenom testiranja. Ako je testiranje rotacije n bilo uspešno, a problem se javi u rotaciji $n + 1$, tada se izvor problema najverovatnije nalazi u razlici između rotacija n i $n + 1$, naime, u dodatnoj funkcionalnosti između jedne rotacije i sledeće. Ako je razlika između dve susedne rotacije mala, imaćemo relativno malo mesta na kojima treba tražiti uzrok problema.

Broj rotacija i njihove definicije pre svega zavise od naših resursa i od resursa kupca. U te resurse ne spadaju samo hardver i softver već takođe i vreme i raspoloživost osoblja. Minimalni sistem se stavlja u prvu rotaciju, a sledeće rotacije se definišu tako da se što je moguće ranije integrišu u njih integrišu sledeće najvažnije ili najkritičnije funkcije. Na primer, analizirajmo zvezdastu mrežu prikazanu na slici 9.3. Centar zvezde je jedan računar koji prima poruke od više manjih računara, a svaki od njih prihvata podatke sa senzora i predaje ih na obradu. Prema tome, glavne funkcije centralnog računara jesu prevođenje i objedinjavanje poruka sa udaljenih računara. Pošto su te funkcije najbitnije za ceo sistem, njih treba uključiti u što raniju rotaciju. U stvari, rotacije možemo da definišemo na sledeći način:

- rotacija 0: testirati opšte funkcije centralnog računara;
- rotacija 1: testirati funkcije prevođenja u centralnom računaru;

SLIKA 9.3 Primer zvezdaste mreže



- rotacija 2: testirati funkcije objedinjavanja u centralnom računaru;
- rotacija 3: testirati svaki od isturenih računara u samostalnom radu;
- rotacija 4: testirati funkcije slanja poruka iz isturenih računara;
- rotacija 5: testirati funkcije primanja poruka u centralnom računaru

i tako dalje.

Definicije rotacija takođe zavise od mogućnosti komponenti sistema da rade u samostalnom režimu. Ponekad je teže simulirati deo sistema koji nedostaje nego ga samog uključiti u odgovarajuću rotaciju, pošto zavisnosti među delovima ponekad zahtevaju isto toliko koda za simulaciju koliko se nalazi i u stvarnom kodu. Imajte na umu da nam je cilj testiranje sistema. Vreme i rad koji su potrebni za izgradnju i upotrebu alata za testiranje možda je bolje iskoristiti za testiranje samog sistema. Ova dilema je slična onoj koja se javlja prilikom biranja filozofije testiranja kada radimo jedinično i integraciono testiranje: pravljenje mnogih lažnih rutina i rukovalaca ponekad u testiranju odnese isto toliko vremena kao i testiranje pravih komponenti koje oni treba da zamene.

Upravljanje konfiguracijom

Često se sistem testira po fazama ili delovima, po rotacijama (kao malopre) ili po podsistemima, funkcijama ili drugačijim podelama koje olakšavaju testiranje. (Takve strategije testiranja ćemo obraditi kasnije u ovom poglavlju.) Međutim, prilikom testiranja sistema moramo da imamo u vidu različite konfiguracije sistema koji pravimo. **Konfiguracija sistema** je kolekcija komponenti sistema koja se isporučuje konkretnom kupcu. Na primer, paket za matematička izračunavanja može da se prodaje u jednoj konfiguraciji za UNIX mašine, u drugoj za Windows mašine, a u trećoj za sisteme Solaris. Konfiguracije se takođe mogu razlikovati po tome što se izvršavaju na određenim vrstama čipova ili sa konkretnim raspoloživim uređajima. Obično razvijamo osnovni softver koji se izvršava svuda i primenjujemo principe opisane u poglavljima 5, 6 i 7 da bismo razlike među konfiguracijama izolovali u mali broj nezavisnih komponenti. Na primer, osnovna funkcionalnost može da se nalazi u komponentama A, B i C; pa će, konfiguraciju 1 da čine A, B, C i D, a konfiguraciju 2 komponente A, B, C i E.

Da bi se rizici i greške sveli na minimum, razvijanje i testiranje tih različitih konfiguracija zahteva **upravljanje konfiguracijom**, tj. kontrolu specifičnosti sistema. U prethodnim poglavljima smo videli kako tim za upravljanje konfiguracijom obezbeđuje da se promena u zahtevima, dizajnu ili kodu prikaže u dokumentaciji i u drugim komponentama na koje ta promena utiče. Tokom testiranja, upravljanje konfiguracijom je posebno važno da bi se koordinirale aktivnosti testiranja i razvoja.

Verzije i izdanja. Jedna konkretna konfiguracija sistema ponekad se naziva **verzijom**. Prema tome, početna isporuka softverskog paketa može da se sastoji od nekoliko verzija, od po jedne za svaku platformu ili situaciju u kojoj će se softver koristiti. Na primer, softver za avione može tako da se gradi da se verzija 1 izvršava na avionima mornarice, verzija 2 na avionima vazduhoplovstva, a verzija 3 na komercijalnim avionima.

Tokom testiranja i upotrebe softvera, otkrivaju se greške koje treba ispraviti ili se prvobitna funkcionalnost malo unapređuje. Novo **izdanje** softvera je poboljšani sistem koji treba da zameni stari. Softverski sistemi se često numerišu kao verzija *n*, izdanje *m*,

odnosno kao verzija *n.m*, i taj broj odražava stanje sistema tokom njegovog rasta i sazrevanja. Verzija *n* ponekad služi kao zamena za verziju *n* -1, a izdanje *m* zamenjuje izdanje *m* -1. (Reč „verzija" može da ima dva različita značenja: po jedna verzija za svaku vrstu platforme ili operativnog sistema, ili po jedna verzija u nizu razvojnih faza proizvoda. Značenje se obično shvata iz konteksta u kojem se koristi. Na primer, prodavac može da nudi verziju 3 svog proizvoda za UNIX platformu, a verziju 4 za Windows, iako obe sadrže istu funkcionalnost.)

Tim za upravljanje konfiguracijom zadužen je da obezbedi da svaka verzija (ili izdanje) bude ispravna i stabilna pre nego što se izda na korišćenje, a da se promene vrše precizno i brzo. Preciznost je presudna zato što želimo da izbegnemo unošenje novih grešaka kada ispravljamo postojeće. Slično tome, brzina je važna zato što se otkrivanje i ispravljanje grešaka obavlja u isto vreme dok tim za testiranje traži dodatne greške. Prema tome, oni koji pokušavaju da isprave sistemske greške treba da koriste komponente i dokumentaciju o trenutnom stanju sistema.

Praćenje i kontrola verzija su posebno važni kada se razvoj odvija u fazama. Kao što smo u prethodnim poglavljima napomenuli, **produkcioni sistem** je verzija koja je testirana i funkcioniše u skladu sa jednim podskupom zahteva kupca. Sledeća verzija, sa više svojstava, još se razvija dok korisnici upotrebljavaju produkcioni sistem. Ovaj **sistem u razvoju** se gradi i testira. Kada se testiranje završi, sistem u razvoju zamenjuje produkcioni sistem i postaje novi produkcioni sistem.

Na primer, pretpostavite da elektrana automatizuje funkcionisanje kontrolne sale. Radnici elektrane su navikli da sve rade ručno, zbunjuje ih rad sa računarom i zato odlučujemo da gradimo sistem u fazama. Prva faza je skoro identična ručnom sistemu, ali omogućava radnicima elektrane da koriste automatsko evidentiranje. U drugoj fazi se na prvu fazu dodaje nekoliko automatizovanih funkcija, ali se polovina funkcija u kontrolnoj sali i dalje obavlja ručno. Svaka sledeća faza se nadovezuje na prethodne i automatizuje deo po deo funkcija, dok sve ne bude automatizovano. Kada na ovaj način proširujemo automatizovani sistem, radnicima u elektrani omogućavamo da se postepeno privikavaju na novi sistem i steknu poverenje u njega.

U svim fazama faznog razvoja, radnici elektrane koriste potpuno testiran produkcioni sistem. Mi za to vreme radimo na sledećoj fazi i testiramo sistem u razvoju. Pošto se sistem u razvoju potpuno istestira i bude spreman za radnike elektrane, on postaje produkcioni sistem (tj. koriste ga radnici u elektrani) a mi prelazimo na sledeću fazu. Dok radimo na sistemu u razvoju, mi postojećem produkcionom ili operativnom sistemu dodajemo funkcije i tako oblikujemo novi sistem u razvoju.

Kada je sistem u produkciji, prijavljuju nam se svi problemi koji se mogu javiti. Na taj način sistem u razvoju često ima dve svrhe: dodavanje funkcionalnosti sledeće faze i ispravljanje problema otkrivenih u prethodnim verzijama. Sistem u razvoju, dakle, može da uključuje dodavanje novih komponenti, ali takođe i menjanje postojećih. Međutim, na taj način mogu da se provuku greške u komponente koje su već testirane. Kada pišemo plan izgradnje i planove testiranja, trebalo bi da uzmemo u obzir tu situaciju i potrebu da se kontrolišu izmene koje se implementiraju između jedne verzije (izdanja) i sledeće. Dodatno testiranje treba da obezbedi da sistem u razvoju funkcioniše bar na nivou trenutnog produkcionog sistema. Međutim, važno je da se precizno evidentiraju izmene koje su unete u kôd između dve verzije, da bi se omogućilo praćenje problema

prema njihovom izvoru. Na primer, ako korisnik produkcionog sistema prijavi neki problem, moramo da znamo koju verziju i izdanje je tada koristio. Kôd može drastično da se razlikuje između dve verzije. Ako radimo na pogrešnom listingu, možda nikada nećemo pronaći uzrok problema. Što je još gore, može nam se učiniti da smo našli uzrok, pa izvršimo izmenu kojom unosimo novu grešku umesto da ispravimo staru!

Regresiono testiranje. Kao što smo videli u poglavlju 8, svrha testiranja je da se greške uoče, a ne da se isprave. Međutim, prirodna je želja da se pronađeni uzrok problema ispravi što je moguće pre pošto je uočen. Inače, tim za testiranje ne bi mogao da proceni da li sistem pravilno funkcioniše, a prisustvo nekih grešaka onemogućava nastavak testiranja. Prema tome, svaki plan testiranja mora da sadrži skup smernica za ispravljanje grešaka, a ne samo za njihovo otkrivanje. Međutim, kao što je već rečeno, ispravljanje grešaka u toku testiranja može da unese nove greške prilikom ispravljanja starih.

U regresionom testiranju se prepoznaju nove greške, koje su možda unete prilikom ispravljanja postojećih. **Regresioni test** je test koji se primenjuje na novu verziju ili izdanje da bi se proverilo da li se još uvek izvršavaju iste funkcije i da li se izvršavaju na isti način kao u staroj verziji ili izdanju.

Na primer, pretpostavimo da je funkcionalni test za verziju m uspešan i da se prelazi na verziju $m + 1$, gde $m + 1$ sadrži svu funkcionalnost iz m plus neke nove funkcije. Da bi se ispravila neka greška koja je pronađena u nekom ranijem testiranju, zahteva se da se u verziju $m + 1$ doda nekoliko redova koda; sada treba da se doda taj kôd, a zatim da se nastavi sa testiranje $m + 1$. Ako tim poštuje postupak strogog regresionog testiranja, testiranje će se sastojati od sledećih koraka:

1. ubacivanje novog koda;
2. testiranje funkcija za koje znamo da i na njih novi kôd utiče;
3. testiranje bitnih funkcija iz m da bismo proverili da li još uvek ispravno rade (regresioni test);
4. nastavak testiranja $m + 1$.

Ovi koraci su zaštita od toga da dodavanje novog koda poništi efekte prethodnih testova. U podsetniku 9.1 ilustrovane su opasnosti u vezi sa izostavljanjem regresionih testova.

Često se u regresionom testu ponovo koriste najvažniji slučajevi za testiranje iz prethodne faze; ako u planu testiranja predvidate regresiono testiranje, trebalo bi takođe da navedete koje slučajeve treba ponovo iskoristiti.

Razlike, zasebne datoteke i uslovno prevođenje. Postoje tri osnovna načina da se kontrolišu verzije i izdanja, a svi oni utiču na upravljanje konfiguracijom tokom testiranja. Na primer, jedan bezbednosni sistem može da se izdaje u dve konfiguracije: verzija 1 za mašine koje mogu da čuvaju sve podatke u glavnoj memoriji i verzija 2 za mašine sa manje memorije, gde podaci moraju u određenim uslovima da se pohrane na disk. Osnovna funkcionalnost sistema može da bude zajednička, u komponentama A_1 do A_k, ali upravljanje memorijom može da se izvršava u komponenti B_1 za verziju 1, a u komponenti B_2 za verziju 2.

PODSETNIK 9.1 POSLEDICE PROPUŠTENOG REGRESIONOG TESTIRANJA

Ako se propusti odgovarajuće regresiono testiranje, može doći do ozbiljnih posledica. Na primer, Seligman (1997) i Trager (1997) izveštavaju da je 167.000 stanovnika Kalifornije dobilo račune od 667.000 dolara za neopravdane lokalne telefonske pozive zbog jednog problema sa softverom koji je kupljen u Northern Telecomu. Sličan problem su doživeli klijenti u Njujorku.

Problem je izazvala jedna greška u nadogradnji softvera za telefonsku centralu DMS-100. Zbog te greške je interfejs za fakturisanje koristio pogrešan pozivni broj za kancelarije telefonskih kompanija sa više pozivnih brojeva. Zbog toga su lokalni pozivi obračunavani kao međugradski pozivi. Kada su se klijenti žalili, lokalne telefonske kompanije su odgovarale klijentima da je problem u međugradskoj centrali; zatim su iz međugradske centrale vraćali klijente u lokalnu telefonsku kompaniju! Lokalnim telefonskim kompanijama je bilo potrebno oko mesec dana da isprave uzrok problema. Da su u Northern Telecomu prilikom nadogradnje softvera obavili potpuno regresiono testiranje, uključujući i proveru da li se pravilno obrađuju pozivni brojevi, ne bi ni došlo do problema sa fakturisanjem.

Pretpostavimo da se otkrije greška u B1 koja postoji i u B2 i mora da se ispravi tako da svekomponente funkcionišu na isti način. Ili pretpostavite da određena funkcionalnost mora da se doda i u B1 i u B2. Ponekad je teško obezbediti da obe verzije budu ažurne i ispravne. Potrebne promene verovatno neće biti identične, ali njihov rezultat mora sa aspekta korisnika biti isti. Da bismo rešili tu teškoću, možemo da odredimo da jedna verzija bude glavna verzija, a da sve ostale verzije budu njene varijante. Za sve ostale verzije moramo da čuvamo jedino razlike, a ne cele komponente. Datoteka razlika koja se naziva **delta**, sadrži komande za uređivanje pomoću kojih se glavna verzija pretvara u drugu verziju. Pretvaranje glavne verzije u njenu varijaciju možemo nazvati *primena delte*.

Delte su korisne zato što se zajednička funkcionalnost i nadalje menja samo u glavnoj verziji. Osim toga, za delte treba daleko manje prostora nego za cele verzije. Međutim, postoje značajne mane. Ako se glavna verzija izgubi ili ošteti, biće izgubljene sve verzije. Što je još važnije, ponekad je veoma teško predstaviti svaku varijaciju kao transformaciju glavne verzije. Na primer, uzmimo da u glavnoj verziji imamo sledeći kôd:

```
...
26      int total = 0;
...
```

Delta datoteka definiše varijaciju u kojoj se red 26 zamenjuje novim kodom:

```
26      int total = 1;
```

Međutim, uzmimo da se u glavnoj verziji datoteke izvrši izmena kojom se doda jedan red između redova 15 i 16. Tada red 26 postaje red 27 i primena delte će izmeniti pogreš-

nu komandu. Prema tome, za očuvanje korespondencije između glavne verzije i njenih varijacija neophodne su sofisticiranije tehnike za ispravnu primenu razlika (delta).

Delte su posebno korisne za održavanje izdanja. Prvo izdanje se smatra glavnim sistemom, a sledeća izdanja se evidentiraju kao skup razlika u odnosu na prvo izdanje.

Treći pristup kontrolisanju razlika među datotekama je upotreba **uslovnog prevođenja**. To jest, sve verzije se nalaze u istoj komponenti koda. Uslovni iskazi usmeravaju prevodioca da odredi koji iskazi spadaju u koje verzije. Pošto se zajednički kôd pojavljuje samo jednom, moguće je izvršiti jednu izmenu koja se primenjuje na sve verzije. Međutim, ako su razlike među verzijama veoma složene, može se desiti da izvorni kôd bude težak za čitanje i razumevanje. Štaviše, uslovno prevođenje može da postane teško izvodljivo ako imamo veliki broj verzija.

Uslovno prevođenje rešava jedino problem koda. Međutim, zasebne datoteke i delte su korisne ne samo za kontrolisanje koda već takođe i za kontrolu drugih razvojnih proizvoda, kao što su zahtevi, dizajn, podaci za testiranje i dokumentacija. U podsetniku 9.2 opisano je kako delte i zasebne datoteke mogu da budu korisne za organizovanje i menjanje velikih sistema.

PODSETNIK 9.2 DELTE I ZASEBNE DATOTEKE

Sistem za kontrolu izvornog koda (*Source Code Control System*, SCCS) koji se isporučuje uz većinu verzija UNIX-a firme AT&T, namenjen je kontroli softverske osnove projekta. On se takođe može koristiti i za druge projektne dokumente samo ako su oni u tekstualnom obliku. Ako koristimo princip s razlikama, SCCS omogućava više verzija i izdanja, a programer može u svako doba od sistema da zahteva bilo koju verziju ili izdanje. Osnovni sistem se čuva zajedno s transformacijama. To jest, za datu komponentu SCCS čuva u jednoj datoteci osnovni kôd za verziju 1.0 te komponente, razlike za transformisanje te verzije u verziju 2.0 i razlike za transformisanje verzije 2.0 u verziju 3.0. Slično tome, SCCS može da čuva različita izdanja ili kombinacije verzija i izdanja. Prema tome, svako dato izdanje ili verzija uvek se može upotrebiti ili promeniti. SCCS jednostavno primeni odgovarajuće razlike i izvodi je od osnovne verzije. Međutim, promena u međuverziji ili izdanju može da dovede do problema pošto se razlike za sledeću verziju ili izdanje oslanjaju na tekst prethodne verzije. S druge strane, fleksibilnost SCCS-a u obradi više izdanja i verzija omogućava da prodavac može istovremeno da podržava više verzija i izdanja.

Programer zahteva određenu verziju ili izdanje od SCCS-a pomoću komande „get". Ako programer parametrom „-e" označi da će se komponenta menjati, SCCS zaključava tu komponentu za sve buduće korisnike sve dok se promenjena komponenta ponovo ne oslobodi.

Sistem ALS (*Ada Language System*) je programsko okruženje u kome je ključni element dizajna upravljanje konfiguracijom (Babich 1986). Samo okruženje ne nameće nijednu konkretnu strategiju za upravljanje konfiguracijom. Umesto toga ono sadrži komande u UNIX-ovom stilu koje upravljaju alatima za rukovanje konfiguracijom. Za razliku od SCCS-a, ALS čuva revizije kao zasebne datoteke. Osim toga, ALS zamrzava sve verzije i izdanja osim tekućeg. To jest, kada se jednom nova verzija ili izdanje preda korisnicima, stare verzije i izdanja ne mogu više da se menjaju.

ALS dozvoljava da se kolekcije povezanih izdanja ili verzija grupišu u skup varijacija. Skup varijacija može da se sastoji od jedne produkcione verzije i nekoliko razvojnih verzija, ili od jedne verzije sa nekoliko naknadnih izdanja. ALS takođe uz svaku datoteku čuva opisne informacije kao što su datum stvaranja, ime onoga kome je izdata na korišćenje, datum poslednjeg testiranja, pa čak i svrha datoteke. Sistem takođe evidentira pridruživanja tako da se mogu markirati sve datoteke iz jednog sistema ili sve datoteke iz jednog skupa varijacija.

Šema kontrole pristupa za ALS uključuje i zaključavanje na osnovu imena ljudi kojima je dozvoljeno da čitaju, prepisuju, dodaju ili izvršavaju podatke iz datoteke. Sistem takođe dodeljuje dozvole za određene alate kojima se pristupa ili se nešto radi sa datotekom.

Kontrola promena. Tim za upravljanje konfiguracijom tesno sarađuje sa timom za testiranje radi kontrole svih aspekata testiranja. Svaku predloženu izmenu bilo kog dela sistema mora prvo da odobri tim za upravljanje konfiguracijom. Izmena se unosi u sve odgovarajuće komponente i dokumentaciju, a tim obaveštava sve one na koje to može da utiče. Na primer, ako neki test dovede do promene zahteva, verovatno će biti potrebne i promene u specifikaciji zahteva, u dizajnu sistema, u dizajnu programa, u kodu, u pripadajućoj dokumentaciji, pa čak i u samom planu testiranja. Prema tome, promena u jednom delu sistema može da utiče na sve koji rade na razvoju sistema.

Kontrola promena se još više komplikuje ako više učesnika u razvoju unosi promene u istu komponentu. Na primer, pretpostavimo da tokom testiranja dođe do dva otkaza. Džek je zadužen da pronađe i ispravi uzrok prvog, a Džil je određena da pronađe i ispravi uzrok drugog otkaza. Mada na početku ne izgleda da postoji veza između ta dva otkaza, Džek i Džil dolaze do zaključka da je osnovni uzrok u komponenti po imenu *initialize*. Džek može da uzme *initialize* iz sistemske biblioteke, unese svoje promene i vrati ispravljenu verziju u biblioteku. Zatim Džil koja je radila iz originalne verzije unese svoje izmene i zameni Džekove ispravke svojima, ali pri tom poništava njegove! Regresiono testiranje bi moglo da otkrije da greška dodeljena Džeku još uvek nije ispravljena, ali su izgubljeni i vreme i trud.

Da bi se rešio ovakav problem, tim za upravljanje konfiguracijom kontroliše promene. Tim nadgleda biblioteke koda i dokumenata, a učesnici u razvoju moraju da se „zaduže" sa kopijama kada vrše ispravke. U našem primeru, Džil ne bi mogla da dobije primerak komponente *initialize* sve dok Džek ne bi zamenio preuzetu verziju ispravljenom i testiranom verzijom. Tim za upravljanje konfiguracijom bi mogao takođe da učini još jedan korak, a to je da objedini Džekovu i Džilinu verziju u jednu verziju; i da zatim objedinjenu verziju podvrgne regresionom testiranju kao i dodatnom testiranju kojim bi se potvrdilo da su oba nedostatka otklonjena.

Još jedan metod kojim se obezbeđuje da svi učesnici u projektu koriste najažurnije dokumente jeste da oni budu onlajn. Kada pregledamo dokumente na ekranu i odmah ih ažuriramo, izbegavamo vreme koji obično trošimo na štampanje i podelu novih ili revidiranih stranica. Međutim, tim za upravljanje konfiguracijom i dalje ima određenu kontrolu radi provere da li promene u dokumentaciji odslikavaju promene u dizajnu i kodu. I ovde možda moramo da se „zadužimo" nekom verzijom da bismo je menjali, a može se dogoditi da su neki dokumenti zaključani ili nedostupni ako neko drugi na njima radi.

U podsetniku 9.3 opisano je kako Microsoft koristi privatne kopije izvornog koda na kojima učesnici u razvoju jedinično testiraju promene, a zatim se promene sastavljaju u konstrukciju za taj dan.

PODSETNIK 9.3 MICROSOFTOVA KONTROLA GRADNJE

Cusumano i Selby (1997) izveštavaju da Microsoftovi učesnici u razvoju moraju da unesu svoj kôd u produkcionu bazu podataka do određenog vremena u poslepodnevnim časovima. Nakon toga projektni tim ponovo prevodi izvorni kôd i sastavlja novo stanje proizvoda u konstrukciju za sledeće jutro. Kôd koji ima toliko grešaka da se proizvod ne može prevesti i izvršiti mora odmah da se ispravi.

Sam postupak izrađnje ima nekoliko koraka. Prvo učesnik u razvoju preuzima privatnu kopiju datoteke izvornog koda iz zajedničkog mesta na kom se čuvaju zvanične verzije. Zatim on menja privatnu kopiju tako što implementira nova ili menja postojeća svojstva. Posle ispravki, testira se privatna kopija sa novim i promenjenim svojstvima. Zatim se kôd za nova izmenjena svojstva ubacuje u zvaničnu verziju. Konačno, regresiono testiranje obezbeđuje da unete izmene nisu nenamerno uticale na ostalu funkcionalnost.

Pojedinačni učesnici u razvoju mogu po potrebi da kombinuju svoje izmene (ponekad svakodnevno, a ponekad svake nedelje, zavisno od potrebe), ali „gospodar izgradnje" svakodnevno pravi kompletnu verziju proizvoda u koju uključuje zvanične verzije svih datoteka izvornog koda za taj dan. Ovo svakodnevno sastavljanje izvršava se za svaki proizvod i za svako tržište.

Tim testera

Kao što ćemo videti, učesnici u razvoju su pre svega zaduženi za funkcionalni test i za test performanse, ali kupac ima veliku ulogu u završnom testu prihvatljivosti i instalacionim testovima. Međutim, za sva testiranja se timovi sastavljaju iz obe ekipe. Često se programeri koji učestvuju u projektu ne uključuju u testiranje sistema; previše im je bliska struktura implementacije i njene namere da bi bili u stanju da uoče razliku između implementacije i zahtevanih funkcija ili performanse.

Zbog toga je testerski tim često odvojen od osoblja koje radi na implementaciji. Bilo bi idealno da neki članovi testerskog tima imaju iskustva u testiranju. Obično su ovi „profesionalci" bivši analitičari, programeri i dizajneri koji sada posvećuju sve svoje vreme testiranju sistema. Oni poznaju specifikaciju sistema, ali takođe i metode i alate za testiranje.

Profesionalni testeri organizuju i izvršavaju testove. Oni su uključeni od početka, projektuju planove za testiranje i slučajeve za testiranje paralelno sa napretkom projekta. Oni sa timom za upravljanje konfiguracijom sarađuju na izradi dokumentacije i drugih mehanizama za povezivanje testova sa zahtevima, komponentama dizajna i kodom.

Profesionalci u obavljanju testiranja usredsređuju se na razvijanje testova, na metode i procedure. Pošto oni možda ne poznaju najbolje detalje iz zahteva, za razliku od onih koji su ih pisali, u testerski tim se uključuju ljudi kojima su zahtevi poznati. **Analitičari**

koji su bili uključeni u prvobitno definisanje i specifikaciju zahteva korisni su prilikom testiranja zato što razumeju problem na način kako ga kupac vidi. Veliki deo testiranja sistema sastoji se od poređenja novog sistema sa prvobitnim zahtevima, a analitičari imaju dobar osećaj za potrebe i ciljeve kupca. Pošto su sarađivali sa projektantima na oblikovanju rešenja, analitičari imaju predstavu o tome kako sistem treba da radi u cilju rešavanja problema.

Projektanti sistema dodaju timu za testiranje dimenziju namere. Projektanti shvataju šta smo predložili kao rešenje, kao i ograničenja tog rešenja. Oni takođe znaju kako je sistem podeljen na funkcionalne podsisteme ili podsisteme koji zavise od podataka i kako bi sistem trebalo da radi. Kada tim za testiranje priprema slučajeve za testiranje i obezbeđuje pokrivenost testiranjem, on poziva projektante u cilju prikaza svih mogućnosti.

Pošto su testovi i slučajevi direktno povezani sa zahtevima i dizajnom, u tim se uključuje i **predstavnik grupe za upravljanje konfiguracijom**. Kada dođe do greške i treba uneti promene, specijalista za upravljanje konfiguracijom se brine da se promene unesu u dokumentaciju, zahteve, dizajn, kôd i ostale produkte procesa razvoja. U stvari, promene potrebne da bi se ispravila greška možda zahtevaju promene i u drugim slučajevima za testiranje ili u većem delu plana testiranja. Specijalista za upravljanje konfiguracijom implementira te promene i koordinira reviziju testiranja.

Konačno, u tim se uključuju **korisnici**. Oni su najkvalifikovaniji da procene pitanja koja se tiču prikladnosti, lakoće korišćenja i ostalih ljudskih faktora. Ponekad se korisnici malo pitaju u ranim fazama projekta. Predstavnici kupca koji učestvuju u analizi zahteva možda ne nameravaju sami da koriste sistem, ali ih posao povezuje sa onima koji će ga koristiti. Na primer, predstavnici su možda rukovodioci budućih korisnika sistema ili tehnički predstavnici koji su otkrili problem posredno vezan za njihov posao. Međutim, ovi predstavnici su možda toliko udaljeni od stvarnog problema da je opis zahteva neprecizan i nepotpun. Kupac možda nije svestan potrebe da se zahtevi dorade ili prošire.

Prema tome, korisnici predloženog sistema su bitni, pogotovo ako nisu bili prisutni kada su na početku definisani zahtevi sistema. Korisnik je verovatno upoznat sa problemom jer svakodnevno nailazi na njega i može da pruži neprocenjivu pomoć u ocenjivanju sistema i proveri da li on rešava problem.

9.2 FUNKCIONALNI TEST

Testiranje sistema počinje od funkcionalnog testa. Dok su prethodni testovi bili skoncentrisani na komponente i njihova međudejstva, u ovom prvom koraku se zanemaruje struktura sistema i usredsređuje se na funkcionalnost. Od sada je pristup bliži zatvorenoj nego otvorenoj kutiji. Ne moramo da znamo koja se komponenta izvršava, već moramo da znamo šta bi sistem trebalo da radi. Prema tome, funkcionalni test se zasniva na funkcionalnim zahtevima sistema.

Svrha i uloge

Svakoj funkciji se mogu pridružiti one komponente sistema koje je izvršavaju. Za neke funkcije to može da obuhvati i ceo sistem. Skup aktivnosti pridružen jednoj funkciji naziva se **nit**, pa se funkcionalno testiranje ponekad naziva i **testiranje niti**.

Logično je da se uzrok problema lakše pronađe u manjem nego u velikom skupu komponenti. Zbog toga treba pažljivo birati redosled testiranja funkcija kako bi se olakšalo testiranje. Funkcije se mogu definisati na ugnežđeni način, kao što se u nivoima definišu rotacije. Na primer, pretpostavimo da neki zahtev određuje da sistem za nadzor vode prepozna značajne promene nekog od četiri parametra: rastvoreni kiseonik, temperatura, kiselost i radioaktivnost. Specifikacija zahteva može da posmatra registrovanje promene kao jednu od mnogih funkcija celokupnog sistema. Međutim, za potrebe testiranja ćemo možda želeti da posmatramo nadgledanje kao četiri zasebne funkcije:

- registrovanje promene rastvorenog kiseonika;
- registrovanje promene temperature;
- registrovanje promene kiselosti;
- registrovanje promene radioaktivnosti.

Zatim pojedinačno testiramo svaku od njih.

U efikasnom funkcionalnom testu verovatnoća otkrivanja grešaka je visoka. Za funkcionalni test koristimo iste smernice kao i za jedinično testiranje. To jest:

- visoka verovatnoća otkrivanja grešaka;
- testerski tim nezavisan od projektanata i programera;
- poznate očekivane akcije i izlaz;
- testiranje ispravnih i neispravnih ulaza;
- sistem se nikada ne menja da bi se testiranje olakšalo;
- definisani kriterijumi završetka.

Funkcionalno testiranje se vrši u pažljivo kontrolisanoj situaciji. Štaviše, pošto testiramo jednu po jednu funkciju, funkcionalno testiranje može, ako treba, zaista da počne pre nego što se ceo sistem izgradi.

U funkcionalnom testiranju se stvarna performansa sistema poredi sa zahtevima, zato se slučajevi za funkcionalno testiranje prave na osnovu specifikacije zahteva. Na primer, sistem za obradu teksta može se testirati tako što se analizira na koji način sistem rešava:

- stvaranje dokumenta;
- izmenu dokumenta;
- brisanje dokumenta.

U svakoj od tih kategorija testiraju se različite funkcije. Na primer, u izmenama dokumenta može se testirati:

- dodavanje znaka;
- dodavanje reči;
- dodavanje pasusa;
- brisanje znaka;
- brisanje reči;
- brisanje pasusa;

- menjanje fonta;
- menjanje veličine znakova;
- menjanje formatiranja pasusa

i tako dalje.

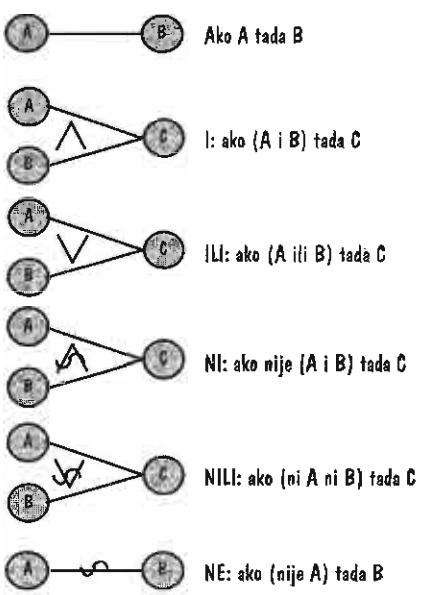
Graf uzroka i posledice

Testiranje bi bilo lakše kad bismo od zahteva mogli automatski da pravimo slučajeve za testiranje. U IBM-u (Elmendorf 1973, 1974) se radilo na konvertovanju prirodnog jezika za definisanje zahteva u formalnu specifikaciju koja se može upotrebiti za određivanje slučajeva za funkcionalno testiranje. Dobijeni slučajevi nisu redundantni; to jest, jedan test slučaj ne ispituje funkcije koje su već testirane u sklopu nekog drugog slučaja. Osim toga, proces je u stanju da pronađe nepotpune i dvosmislene aspekte zahteva, ako takvi postoje.

Ovim procesom se ispituje semantika zahteva i oni se ponovo izlažu u vidu logičkih relacija između ulaza i izlaza ili između ulaza i transformacija. Ulazi se nazivaju **uzroci**, a izlazi i transformacije su **posledice**. Rezultat je Bulov graf u kome se prikazuju te relacije i on se naziva **grafom uzroka i posledica**.

Početnom grafu dodajemo informacije da bismo označili sintaksna pravila i istakli ograničenja koja su posledica okruženja. Zatim konvertujemo graf u tabelu odlučivanja. Svaka kolona tabele odlučivanja odnosi se na jedan slučaj za testiranje, koji će se upotrebiti u funkcionalnom testu.

Prilikom izrade grafa uzroka i posledica postoji nekoliko koraka. Prvo se zahtevi dele tako da svaki zahtev opisuje samo jednu funkciju. Zatim se opisuju svi uzroci i sve posledice. Numerisani uzroci i posledice postaju čvorovi grafa. Stavljanjem uzroka na levu stranu crteža, a posledica na desnu, crtamo logičke relacije prikazane na grafu pomoću notacije sa slike 9.4. Da bi graf bio jednostavniji, mogu se upotrebiti dodatni čvorovi.



SLIKA 9.4 Notacija za graf uzroka i posledica

Obradićemo jedan primer da bismo videli kako se ova vrsta grafova gradi. Pretpostavite da testiramo sistem za nadgledanje vode koji predaje izveštaje jednoj agenciji uključenoj u kontrolu poplava. Definicija zahteva za jednu od funkcija sistema glasi:

> Sistem šalje radniku na brani poruku o bezbednosti nivoa jezera.

Ovom zahtevu odgovara sledeći opis dizajna:

> ULAZ: sintaksa funkcije je NIVO(A, B)
>
> gde je A visina u metrima vode iza brane, a B broj centimetara kiše u proteklih 24 časa.
>
> OBRADA: funkcija izračunava da li se nivo vode nalazi u bezbednim granicama, ili je previše visok ili previše nizak.
>
> IZLAZ: na ekranu se prikazuje jedna od sledećih poruka:
>
> 1. „NIVO = BEZBEDAN" ako je rezultat bezbedan ili niži,
> 2. „NIVO = VISOK" ako je rezultat previsok,
> 3. „NEPRAVILNA SINTAKSA",
>
> zavisno od izračunatog rezultata.

Ove zahteve možemo da razdvojimo na pet „uzroka":

1. Prva četiri znaka komande „NIVO".
2. Komanda sadrži tačno dva parametra razdvojena zarezom i okružena zagradom.
3. Parametri A i B su realni brojevi takvi da se izračunati nivo vode smatra za NIZAK.
4. Parametri A i B su realni brojevi takvi da se izračunati nivo vode smatra za BEZBEDAN.
5. Parametri A i B su realni brojevi takvi da se izračunati nivo vode smatra za VISOK.

Takođe možemo da opišemo tri „posledice":

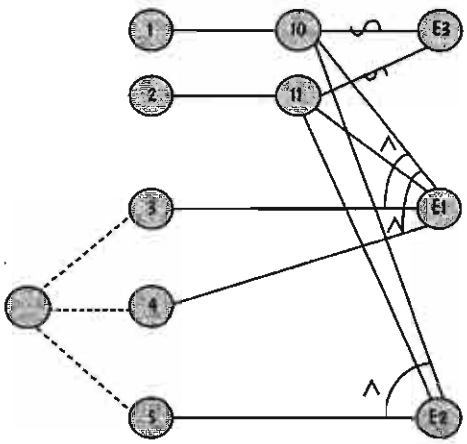
1. Na ekranu se prikazuje poruka „NIVO = BEZBEDAN".
2. Na ekranu se prikazuje poruka „NIVO = VISOK".
3. Na ekranu se prikazuje poruka „NEPRAVILNA SINTAKSA".

To postaju čvorovi našeg grafa. Međutim, funkcija sadrži i proveru ispravnosti dobijenih parametara. U tu svrhu uvodimo dva dodatna čvora:

1. Komanda je sintaksički ispravna.
2. Operandi su sintaksički ispravni.

Sada možemo da nacrtamo relacije između uzroka i posledica, kao što je prikazano na slici 9.5. Obratite pažnju na to da postoje isprekidane linije levo od posledica. Te linije znače da može nastupiti tačno jedna posledica. Na grafu uzroka i posledica mogu se koristiti i druge notacije da bi se prikazale dodatne informacije. Na slici 9.6 prikazane su neke od tih mogućnosti.

SLIKA 9.5 Graf uzroka i posledica

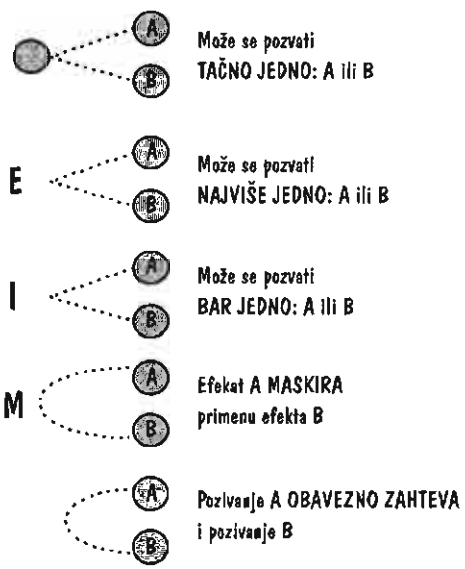


Prema tome, pregledom grafa možemo videti da li:

- je moguće pozvati tačno jedan skup uslova;
- je moguće pozvati najviše jedan skup uslova;
- je moguće pozvati bar jedan skup uslova;
- jedna posledica maskira primenu druge posledice;
- pozivanje jedne posledice zahteva pozivanje druge.

Sada smo spremni da definišemo tabelu odlučivanja koristeći informacije iz grafa uzroka i posledica. U tabeli uzimamo po jedan red za svaki uzrok i svaku posledicu. U našem primeru, tabela odlučivanja treba da ima pet redova za uzroke i tri reda za posledice. Kolone u tabeli odlučivanja pripadaju slučajevima za testiranje. Sadržaj kolona popunjavamo tako što ispitujemo svaku posledicu i navodimo sve kombinacije uzroka koje mogu da dovedu do te posledice.

SLIKA 9.6 Dodatna notacija za graf



U našem primeru NIVO, možemo da odredimo broj kolona u tabeli odlučivanja ispitivanjem linija koje stižu u čvorove posledica na grafu. Na slici 9.5 vidimo da u E3 stižu dve odvojene linije; svaka od njih dobija po jednu kolonu. U E1 stižu četiri linije, ali samo dve kombinacije dovode do te posledice. Svaka od tih kombinacija dobija kolonu u tabeli. Na kraju, samo jedna kombinacija linija dovodi do posledice E2, pa tako dobijamo petu kolonu.

Svaka kolona u tabeli odlučivanja predstavlja skup stanja uzroka i posledica. Prate se takođe stanja ostalih uslova kada se pozove jedna konkretna kombinacija. Stanje uslova označavamo slovom I kada je uzrok pozvan odnosno tačan, a slovom S ako je uzrok onemogućen ili netačan. Ako nam nije važno da li je uzrok pozvan ili onemogućen, možemo da upotrebimo X kao znak da nam to nije važno. Konačno, možemo da označimo da li je konkretna posledica odsutna (A) ili prisutna (P).

Za testiranje funkcije NIVO, pet kolona tabele 9.2 pokazuju odnos između pozvanih uzroka i njihovih posledica. Ako su uzroci 1 i 2 tačni (tj. komande i parametri su ispravni) tada posledica zavisi od tačnosti uzroka 3, 4 ili 5. Ako je uzrok 1 tačan, a uzrok 2 netačan posledica se već zna i nije nam bitno stanje uzroka 3, 4 ili 5. Slično važi i u slučaju da je uzrok 1 netačan - nisu nam više važna stanja ostalih uzroka.

Primetićete da smo teoretski mogli da napravimo 32 slučaja za testiranje: pet uzroka sa svakim od dva moguća stanja daju 25 mogućnosti. Prema tome, upotreba grafa uzroka i posledica bitno smanjuje broj slučajeva za testiranje koje treba razmotriti.

Uglavnom možemo još više da smanjimo broj slučajeva ako dovoljno poznajemo uzroke da možemo da eliminišemo još neke kombinacije. Na primer, ako imamo veliki broj slučajeva, možemo za svaku kombinaciju uzroka da odredimo prioritet. Zatim možemo da odbacimo kombinacije sa niskim prioritetom. Slično tome, možemo da eliminišemo one kombinacije za koje je malo verovatno da će nastupiti ili za koje testiranje nije ekonomski opravdano.

Osim što smanjuje broj slučajeva koje treba uzeti u obzir prilikom testiranja, graf uzroka i posledica nam pomaže da predvidimo mogući ishod isprobavanja sistema. U isto vreme grafovi otkrivaju nenamerne sporedne efekte za određene kombinacije uzroka. Međutim, grafovi uzroka i posledica nisu praktični za sisteme u kojima se javljaju vremenski pomaci, iteracije ili petlje gde sistem reaguje na povratne informacije iz nekih svojih procesa da bi izvršio druge procese.

Tabela 9.2 Tabela odlučivanja za graf uzroka i posledica

	Test 1	Test 2	Test 3	Test 4	Test 5
Uzrok 1	I	I	I	S	I
Uzrok 2	I	I	I	X	S
Uzrok 3	I	S	S	X	X
Uzrok 4	S	I	S	X	X
Uzrok 5	S	S	I	X	X
Posledica 1	P	P	A	A	A
Posledica 2	A	A	P	A	A
Posledica 3	A	A	A	P	P

9.3 TESTIRANJE PERFORMANSE

Pošto smo utvrdili da sistem izvršava funkcije navedene u zahtevima, prelazimo na način na koji se te funkcije izvršavaju. Prema tome, funkcionalni test se bavi funkcionalnim zahtevima, a testiranje performanse nefunkcionalnim.

Svrha i uloge

Performansa sistema se meri u odnosu na ciljeve koje je postavio kupac, onako kako su navedeni u nefunkcionalnim zahtevima. Na primer, funkcionalno testiranje je možda pokazalo da testirani sistem može da izračuna putanju rakete na osnovu njenog potiska, vremenskih uslova i odgovarajućih informacija sa senzora i iz sistema. Testiranje performanse ispituje kvalitet tog izračunavanja. Brzina odgovora na komande korisnika, preciznost rezultata i dostupnost podataka porede se sa performansom koju je propisao kupac.

Testiranje performanse projektuje i izvršava tim za testiranje, a rezultati se predaju kupcu. Pošto testiranje performanse obično uključuje i hardver i softver, u tim se mogu uključiti i hardverski inženjeri.

Vrste testova performanse

Testiranje performanse se zasniva na zahtevima, pa se vrste testova određuju prema vrstama navedenih nefunkcionalnih zahteva.

- **Testovi opterećenja** ocenjuju sistem kada se on optereti do svojih granica u kratkom vremenskom periodu. Ako u zahtevima stoji da sistem treba da opsluži najviše određen broj uređaja ili korisnika, test opterećenja ocenjuje performanse sistema kada su istovremeno aktivni svi ti uređaji ili korisnici. Ovaj test je posebno značajan za sisteme koji obično funkcionišu ispod maksimalnog kapaciteta, ali su žestoko opterećeni u određenim trenucima vršne potražnje.
- **Testovi kapaciteta** posmatraju obradu velike količine podataka u sistemu. Na primer, gledamo da li su strukture podataka (recimo, redovi ili stekovi) definisani dovoljno široko za sve moguće situacije. Osim toga, proveravamo polja, zapise i datoteke, i proveravamo da li će moći da prime sve očekivane podatke. Takođe proveravamo da li sistem reaguje pravilno kada skupovi podataka dosegnu svoj maksimum.
- **Testovi konfiguracije** analiziraju različite softverske i hardverske konfiguracije navedene u zahtevima. Ponekad se gradi sistem koji će koristiti različiti auditorijumi, a sistem u stvari ima ceo spektar konfiguracija. Na primer, možemo da definišemo minimalni sistem koji će opslužiti samostalnog korisnika, a druge konfiguracije se nadovezuju na tu minimalnu konfiguraciju kako bi se opslužili dodatni korisnici. Test konfiguracije ocenjuje sve moguće konfiguracije i proverava da li svaka od njih zadovoljava zahteve.
- **Testovi kompatibilnosti** su potrebni kada sistem ima interfejse prema drugim sistemima. Ispitujemo da li se funkcije interfejsa izvršavaju u skladu sa zahtevima. Na primer, ako sistem treba da komunicira sa velikim sistemom baze podataka i pribavlja informacije iz njega, test kompatibilnosti ispituje brzinu i preciznost pronalaženja podataka.

- **Regresioni testovi** su potrebni kada testirani sistem treba da zameni postojeći sistem. Regresioni testovi garantuju da je performansa novog sistema barem jednaka performansi starog. Regresioni testovi se uvek koriste prilikom faznog razvoja.
- **Testovi bezbednosti** proveravaju da li su zadovoljeni bezbednostni zahtevi. Testiramo karakteristike sistema vezane za dostupnost, integritet i poverljivost podataka i usluga.
- **Vremenski testovi** proveravaju zahteve koji se odnose na vreme odgovora korisniku i vreme izvršenja neke funkcije. Ako neka transakcija mora da se izvrši u zadatom vremenskom intervalu, test izvršava tu transakciju i proverava da li su zahtevi zadovoljeni. Vremenski testovi se obično rade u kombinaciji sa testovima opterećenja da bi se videlo da li su vremenski zahtevi zadovoljeni čak i u situaciji kada je sistem maksimalno opterećen.
- **Testovi okruženja** posmatraju sposobnost sistema da funkcioniše na lokaciji instalacije. Ako su u zahtevima postavljene granice tolerancije na temperaturu, vlažnost, kretanje, prisustvo hemikalija, vlagu, prenosivost, električna ili magnetna polja, prekide napajanja ili neke druge karakteristike okruženja vezanog za lokaciju, tada naši testovi treba da garantuju pravilnu performansu našeg sistema i u tim uslovima.
- **Testovi kvaliteta** procenjuju koliko je sistem pouzdan, podesan za održavanje i dostupan. Ovi testovi uključuju izračunavanje prosečnog vremena do otkaza i prosečnog vremena do oporavka, kao i prosečnog vremena potrebnog za pronalaženje i otklanjanje grešaka. Testove kvaliteta je ponekad teško izvršiti. Na primer, ako se u zahtevima navodi dugačko prosečno vreme između dva otkaza, možda nije izvodljivo da sistem radi dovoljno dugo kako bi se zahtevani prosek proverio.
- **Testovi oporavka** ispituju reagovanje na prisustvo grešaka ili na gubitak podataka, napajanja, uređaja ili usluga. Sistem se podvrgava gubitku sistemskih resursa, a zatim posmatra da li se on pravilno oporavlja.
- **Testovi održavanja** se bave potrebama za dijagnostičkim alatima i procedurama koje pomažu u pronalaženju izvora problema. Može da postoji zahtev da obezbedimo dijagnostičke programe, snimke memorije, praćenje transakcija, logičke šeme i druga pomagala. Proveravamo da li pomagala postoje i da li pravilno funkcionišu.
- **Testiranje dokumentacije** obezbeđuju da svi neophodni dokumenti zaista budu i napisani. Prema tome, ako su potrebna uputstva za korisnike, uputstva za održavanje i tehnička dokumentacija, proveravamo da li ti materijali postoje i da li su informacije u njima konsistentne i precizne, te da li je njihovo korišćenje lako. Štaviše, ponekad se u zahtevima propisuje format i korisnici dokumentacije; tada proveravamo da li dokumenti ispunjavaju te zahteve.
- **Testovi ljudskih faktora** istražuju zahteve koji se tiču interakcije korisnika sa sistemom. Ispitujemo izglede ekrana, poruke, formate izveštaja i druge aspekte koji mogu da utiču na lakoću korišćenja. Osim toga, proveravamo da li operatorske i korisničke procedure ispunjavaju zahteve u pogledu lakoće korišćenja. Ove testove ponekad nazivamo i **testovima upotrebljivosti**.

Mnoge od ovih testova je daleko teže primeniti nego funkcionalne testove. Zahtevi moraju da budu izričiti i detaljni, a kvalitet zahteva često utiče na teškoće prilikom testiranja performanse. Ukoliko zahtev nije jasan i ako ga nije moguće testirati, kao što je objašnjeno u poglavlju 4, tim za testiranje će teško proceniti da li je zahtev zadovoljen.

9.4 POUZDANOST, DOSTUPNOST I MOGUĆNOST ODRŽAVANJA

Jedno od najkritičnijih pitanja u testiranju performanse jeste uveriti se da je sistem pouzdan, njegova dostupnost odgovarajuća i da su mogućnosti njegovog održavanja dobre. Pre ispuruke je nemoguće direktno izmeriti svaku pojedinačnu karakteristiku sistema, a to posebno važi za navedene tri karakteristike za koje često moramo da koristimo posredna merenja. Zato ćemo u ovom odeljku podrobnije razmotriti kako se vrši testiranje da bi se dobili pouzdani i dostupni sistemi koji se mogu održavati.

Definicije

Da biste shvatili šta znače pouzdanost, dostupnost i mogućnost održavanja, uzećemo primer automobila. Automobil se smatra pouzdanim ako skoro uvek radi kako treba. Jasno nam je da neke funkcije mogu da otkažu, a da delove koji se istroše treba popraviti ili zameniti. Međutim, mi očekujemo da pouzdan automobil duže vreme radi bez potrebe za održavanjem. To jest, automobil je pouzdan ako se u dužim periodima konsistentno poželjno ponaša između dva održavanja.

Pouzdanost podrazumeva ponašanje u vremenskom periodu, dok dostupnost opisuje nešto u datom trenutku. Automobil je dostupan ako možete da ga upotrebite kada vam zatreba. Automobil može da bude star dvadeset godina i samo je dva puta popravljan, pa možemo reći da je automobil veoma pouzdan. Ali ako se slučajno nalazi na popravci u trenutku kada vam zatreba, on ipak nije dostupan. Prema tome, nešto može da bude veoma pouzdano, ali da u konkretnom trenutku nije dostupno.

Pretpostavimo da je vaš automobil i pouzdan i dostupan, ali da kompanija koja ga je proizvela više ne postoji. Ova situacija znači da kada vaš automobil otkaže (mada ste rekli da to nije često) serviser ima velike poteškoće da pronađe rezervne delove. Znači, da bi se vaš automobil popravio kako treba i vratio vam se ispravan, on se veoma dugo zadržava u servisu. U tom slučaju, mogućnosti za održavanje vašeg automobila su loše.

Isti pojmovi važe i za softverske sisteme. Zahtevamo da softver funkcioniše konsistentno i ispravno u dugim vremenskim intervalima, da bude dostupan kad nam je potreban i da se brzo i lako popravi ako dođe do greške. Formalno kažemo da je **pouzdanost softvera** verovatnoća da će sistem funkcionisati bez greške pod datim uslovima u datom vremenskom intervalu. Pouzdanost ocenjujemo na skali od 0 do 1: sistem koji je veoma pouzdan imaće meru pouzdanosti blizu 1, a nepouzdan sistem će imati meru blizu 0. Pouzdanost se meri u odnosu na vreme izvršavanja, a ne u odnosu na realno vreme (tj. vreme po satu), da bi preciznije odražavala korišćenje sistema.

Slično tome, **dostupnost softvera** je verovatnoća da sistem u datom trenutku uspešno funkcioniše u skladu sa specifikacijom. Još formalnije, to je verovatnoća da sistem potpuno funkcioniše u datom trenutku, pod pretpostavkom da su takođe dostupni i potrebni

spoljnji resursi. Sistem koji je potpuno uključen i aktivan ima dostupnost 1, onaj koji se ne može koristiti ima dostupnost 0. Dostupnost se meri po proticanju vremena, a ne prema vremenu izvršavanja.

Isto tako, **mogućnost održavanja sistema** je verovatnoća da se, u datim uslovima korišćenja, može izvršiti održavanje u zadatom vremenskom roku pomoću zadatih procedura i resursa. I ona se meri od 0 do 1. Veoma se razlikuje od održavanja hardvera; kod hardverskih grešaka je obično neophodno da sistem bude nedostupan dok se vrši održavanje, ali se održavanje softvera ponekad može obaviti i dok sistem radi.

Pošto se pouzdanost, dostupnost i mogućnost održavanja definišu pomoću otkaza, oni se moraju meriti tek kada je sistem kompletan i u funkciji. Softverski inženjeri obično razlikuju poznate greške od novih; tj. kod utvrđivanja pouzdanosti uzimaju u obzir samo nove greške, a ne one za koje znaju, ali ih još nisu ispravili.

Osim toga, često svakoj grešci dodeljujemo stepen ozbiljnosti kao ocenu njenog uticaja na sistem. Na primer, u vojnom standardu SAD, MIL-STD-1629A definišu se četiri stepena ozbiljnosti greške:

1. *Katastrofalna*: greška koja može da dovede do uništenja ili gubitka sistema.
2. *Kritična*: greška koja može da proizvede ozbiljnu štetu ili tako veliko oštećenje sistema da se izgube sve funkcije.
3. *Marginalna*: greška koja može da proizvede manju štetu ili oštećenje sistema koje dovodi do kašnjenja, gubitka dostupnosti ili degradacije funkcija.
4. *Manja*: greška koja nije tako ozbiljna da bi proizvela štetu ili oštećenje sistema, ali koja dovodi do nepredviđene potrebe za održavanjem ili popravkom.

Podaci o greškama

Kada evidentiramo informacije o otkazima softvera, polazimo od nekoliko pretpostavki o samom softveru. Konkretno, pretpostavljamo da ćemo kada softver otkaže pronaći osnovni uzrok problema i ispraviti ga. Međutim, događa se da prilikom ispravljanja jedne greške unesemo novu ili nenamerno stvorimo uslove (kojih prethodno nije bilo) da druge greške dovedu do otkaza. Uvek se nadamo da će na duže staze doći do povećanja pouzdanosti softvera. (To jest, nadamo se da ćemo imati sve duže i duže intervale između otkaza.) Ali, kratkoročno ponekad može doći i do kraćih intervala između otkaza.

Moguće je nadgledati sistem i evidentirati intervale između otkaza da bismo videli da li pouzdanost raste. Na primer, u tabeli 9.3 data su vremena izvršavanja (u sekundama) između dva uzastopna otkaza jednog sistema komande i kontrole za vreme testiranja kod proizvođača gde je primenjena simulacija stvarnog radnog okruženja sistema (Musa 1979). Na slici 9.7 vidimo graf tih podataka gde se jasno uočava dugoročni porast pouzdanosti pošto se intervali između otkaza uglavnom povećavaju.

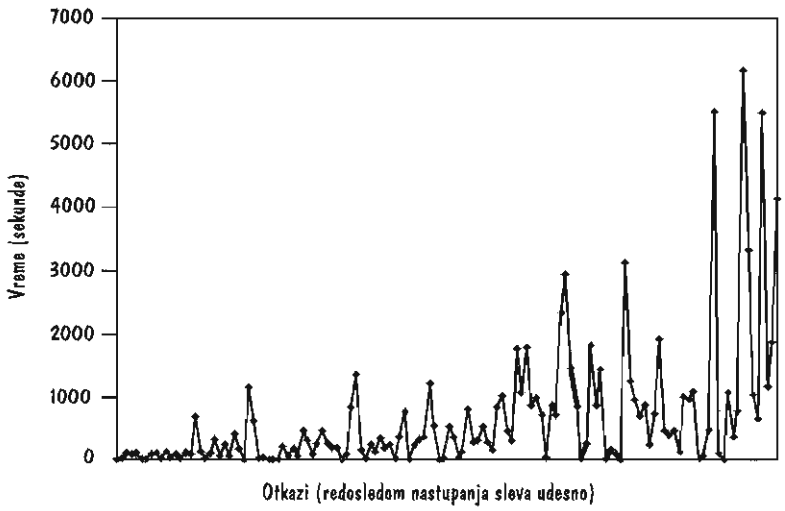
Primetićete velike varijacije između intervala, sa češćim pojavljivanjem kratkih intervala čak i pri kraju skupa podataka. Šta nam ovi podaci govore o pouzdanosti sistema? Kako možemo da ih upotrebimo za predviđanje dužine vremenskog intervala do sledećeg otkaza? Da bismo mogli da odgovorimo na ova pitanja, moramo da razumemo pojam neizvesnosti.

Tabela 9.3 Vremena između otkaza (čitati sleva udesno po redovima)

3	30	113	81	115	9	2	91	112	15
138	50	77	24	108	88	670	120	26	114
325	55	242	68	422	180	10	1146	600	15
36	4	0	8	227	65	176	58	457	300
97	263	452	255	197	193	6	79	816	1351
148	21	233	134	357	193	236	31	369	748
0	232	330	365	1222	543	10	16	529	379
44	129	810	290	300	529	281	160	828	1011
445	296	1755	1064	1783	860	983	707	33	868
724	2323	2930	1461	843	12	261	1800	865	1435
30	143	108	0	3110	1247	943	700	875	245
729	1897	447	386	446	122	990	948	1082	22
75	482	5509	100	10	1071	371	790	6150	3321
1045	648	5485	1160	1864	4116				

U svakom skupu podataka o otkazima sadržana je značajna količina neizvesnosti. Čak i uz potpuno saznanje o svim greškama koje postoje u softveru ne može se sa potpunom izvesnošću odrediti trenutak sledećeg otkaza. Nemoguće je predvideti sledeći otkaz zato što ne znamo kako će se softver koristiti, ne znamo precizno ulaze niti redosled kojim će se oni podnositi softveru, pa ne možemo predvideti koja greška će proizvesti sledeći otkaz. Ovo nazivamo **neizvesnošću tipa 1**, koja znači neizvesnost u načinu korišćenja sistema. Prema tome, u svakom trenutku je vreme do sledećeg otkaza neizvesno; možemo ga posmatrati kao slučajnu promenljivu.

Druga nepoznanica, koju nazivamo **neizvesnost tipa 2**, potiče od toga što ne znamo posledice uklanjanja greške. Kada ispravimo jednu grešku, ne znamo da li je naša ispravka kompletna i uspešna. Čak i ako smo pravilno ispravili grešku, mi ne znamo da li će to poboljšati vreme između otkaza. To jest, neizvestan je stepen do kojeg naša ispravka unapređuje pouzdanost softvera.



SLIKA 9.7 Graf podataka o otkazima iz tabele 9.3

Merenje pouzdanosti, dostupnosti i mogućnosti održavanja

Hteli bismo da izrazimo pouzdanost, dostupnost i mogućnost održavanja kao atribute softvera izražene brojevima između 0 (nepouzdan, nedostupan i nemoguće održavanje) i 1 (potpuno pouzdan, uvek dostupan i sasvim moguće održavanje). Da bismo utvrdili ove mere, ispitujemo atribute podataka o otkazima. Pretpostavimo da beležimo podatke o otkazima i da smo imali i - 1 otkaza. Vreme proteklo između dva otkaza, odnosno vreme do otkaza možemo da označimo kao t_1, t_2,..., ti_{-1}. Prosek svih ovih vrednosti je **prosečno vreme do otkaza** (***Mean Time to Failure***, **MTTF**).

Pretpostavimo da je svaka greška koja je prouzrokovala otkaze ispravljena i da sistem ponovo radi. Označićemo sa *Ti* vreme do sledećeg otkaza koji će se tek dogoditi; *Ti* je slučajna promenljiva. Kada govorimo o pouzdanosti softvera, mi u stvari dajemo iskaze verovatnoće za *Ti*.

Postoji još nekoliko drugih vremenski zavisnih podataka važnih za izračunavanje dostupnosti i mogućnosti održavanja. Pošto dođe do otkaza, postoji i dodatno izgubljeno vreme dok se ne pronađu i otklone greške koje su do njega dovele. **Prosečno vreme do popravke** (***Mean Time to Repair***, **MTTR**) je prosečno vreme potrebno da se popravi neispravna softverska komponenta. Ovu meru možemo kombinovati sa prosečnim vremenom do pojave otkaza da bismo dobili vreme u kojem sistem nije dostupan za upotrebu. To jest, dostupnost merimo kao **prosečno vreme između otkaza** (***Mean Time between Failures***, **MTBF**) koje izračunavamo kao:

$$MTBF = MTTF + MTTR$$

Neki praktičari i istraživači predlažu mere pouzdanosti, dostupnosti i mogućnosti održavanja zasnovane na ovim prosecima. Na primer, uzmimo odnos između pouzdanosti sistema i prosečnog vremena do otkaza. Kako sistem postaje pouzdaniji, njegovo prosečno vreme do otkaza bi trebalo da raste. Upotrebićemo MTTF u jednoj meri čija je vrednost blizu nule kada je vrednost MTTF-a mala, a sve bliža 1 kako MTTF raste. Na osnovu tog odnosa možemo da definišemo meru pouzdanosti sistema kao:

$$R = MTTF/(1 + MTTF)$$

tako da ima vrednosti između 0 i 1 kao što smo tražili. Slično tome, možemo da merimo dostupnost pomoću MTBF:

$$A = MTBF/(1 + MTBF)$$

Ako postoji mogućnost održavanja, tada je cilj da vrednost MTTR-a bude što manja. Zato možemo da merimo mogućnost održavanja kao:

$$M = 1/(1 + MTTR)$$

Drugi istraživači za pojam pouzdanosti koriste surogate mera, kao što je gustina grešaka (tj. broj grešaka na hiljadu redova koda ili broj grešaka po funkcionalnoj tački), ako ne mogu direktno da mere otkaze. Neki istraživači kao što su Voas i Friedman (1995) dokazuju da za pouzdanost softvera nije bitan ukupan broj otkrivenih grešaka ili otkaza, već sposobnost sistema kao celine da prikriva greške koje još nisu otkrivene.

Stabilnost i rast pouzdanosti

Mera pouzdanosti bi trebalo da nam kaže da li se softver poboljšava (tj. ređe otkazuje) kako pronalazimo i ispravljamo greške. Ako vremena između otkaza ostaju ista, tada imamo **stabilnost pouzdanosti**. Ako se vremena produžavaju, imamo **porast pouzdanosti**. Međutim, veoma je teško predvideti kada će sistem da otkaže. Predviđanje je nešto lakše za hardver nego za softver jer se pouzdanost hardvera jasno razlikuje od pouzdanosti softvera, kao što je napomenuto u podsetniku 9.4. Otkaz hardvera ima verovatnoću; možda ne znamo tačno vreme otkaza, ali možemo reći da će konkretan hardverski deo verovatno otkazati u datom vremenskom periodu. Na primer, ako znamo da se guma istroši u proseku za 10 godina, tada shvatamo da verovatnoća otkaza te gume neće odjednom skočiti sa nule 3652. dana (10 godina manje jedan dan) na verovatnoću 1 sledećeg dana. Umesto toga, verovatnoća otkaza polako raste od nule kada kupimo novu gumu do 1 kada se primiče 10 godina korišćenja. Možemo da napravimo grafik porasta verovatnoće u vremenu, a oblik krive će zavisiti od materijala od kojeg je guma napravljena, od dizajna gume, od našeg načina vožnje, od težine automobila i od još nekih faktora. Sve ove parametre koristimo za modelovanje verovatnog otkaza.

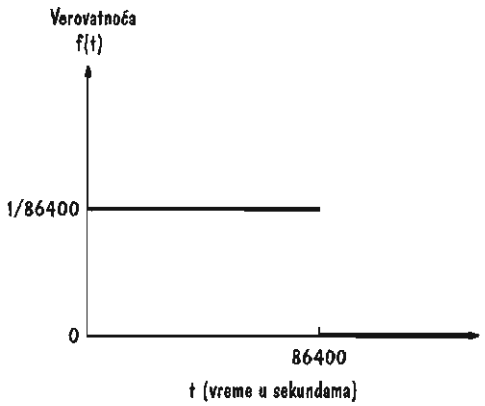
Pristup modelovanju softverskog otkaza je sličan: definišemo **funkciju gustine verovatnoće** f u vremenu t, koju označavamo sa $f(t)$ i kojom opisujemo naše predviđanje trenutka kada bi softver verovatno mogao da otkaže. Na primer, pretpostavite da znamo da će softverska komponenta otkazati u nekom trenutku u sledeća 24 časa (zato što će se neka pomoćna memorija prepuniti), ali da postoji jednaka verovatnoća da ona otkaže u bilo kom jednočasnovnom vremenskom intervalu. U 24 sata postoji 86.400 sekundi, pa možemo da merimo vreme t u sekundama i definišemo funkciju gustine verovatnoće kao 1/86400 za svako t između 0 i 86400, a 0 za svako t veće od 86400. Za ovu funkciju kažemo da je **uniformna** u intervalu od $t = 0$ do 86400 zato što funkcija ima istu vrednost u tom intervalu; to smo prikazali na slici 9.8.

PODSETNIK 9.4 RAZLIKA IZMEĐU POUZDANOSTI HARDVERA I SOFTVERA

Miller (1992) objašnjava zašto se hardverski otkazi suštinski razlikuju od softverskih. Složeni hardver otkazuje kada se neka komponenta pokvari i više ne funkcioniše prema specifikaciji. Na primer, logička kapija može da se zaglavi na 1 ili na 0 ili može da dođe do kratkog spoja u otporniku. Razlog je fizički (korozija ili oksidacija), a greška nastupa u konkretnom trenutku. Da bi se problem otklonio, treba deo popraviti ili zameniti i sistem može da se vrati u prethodno stanje.

Međutim, softverske greške mogu da postoje u proizvodu u dužem vremenskom periodu i da se aktiviraju samo pod određenim uslovima kada se greška pretvori u otkaz. To jest, greška je latentna, a sistem će i dalje otkazivati (pod istim uslovima) ako se ne ispravi dizajn softvera i reši problem u osnovi.

Zbog ove razlike u efektima grešaka, softversku pouzdanost treba definisati drugačije od hardverske. Kada se hardver popravi, on se vraća na svoj prethodni nivo pouzdanosti; pouzdanost hardvera se zadržala. Ali, kad se softver popravi, njegova pouzdanost može da poraste ili da se smanji. Prema tome, cilj uspostavljanja pouzdanosti hardvera je stabilnost, dok je cilj uspostavljanja pouzdanosti softvera porast pouzdanosti.



SLIKA 9.8 Uniformna funkcija gustine

Ali nije svaka funkcija gustine uniformna, a jedan od težih problema prilikom poimanja i merenja pouzdanosti predstavlja prikazivanje ponašanja otkaza odgovarajućom funkcijom gustine verovatnoće. To jest, možemo da definišemo funkciju $f(t)$ i upotrebimo je za izračunavanje verovatnoće da će softverska komponenta otkazati u nekom datom vremenskom intervalu [$t1$, $t2$]. Pošto je ova verovatnoća površina ispod krive između krajnjih tačaka intervala, verovatnoća otkaza između $t1$, i $t2$ je

$$\int_{t_2}^{t_1} f(t)dt$$

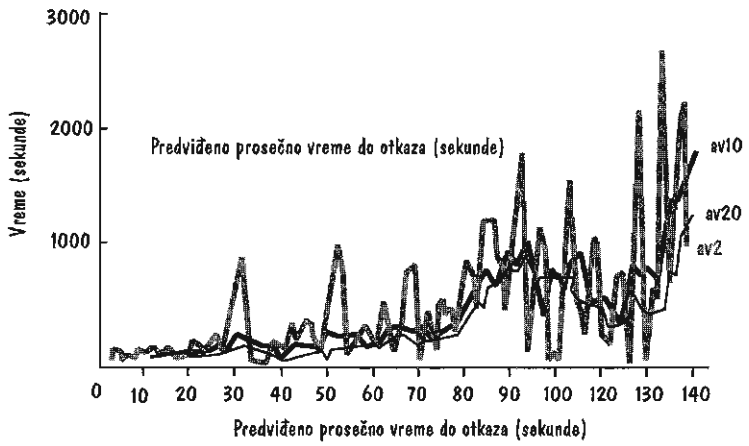
Konkretno, **funkcija raspodele**, $F(t)$ je vrednost ovog integrala nad intervalom od 0 do t. $F(t)$ je verovatnoća da će softver otkazati pre trenutka t, a **funkciju pouzdanosti** $R(t)$, definišemo kao 1 - $Fi(t)$; to je verovatnoća da će softver funkcionisati pravilno do trenutka t.

Predviđanje pouzdanosti

Jednostavne modele predviđanja verovatnoće možemo da gradimo na osnovu istorijskih informacija o otkazima i vremenima otkaza. Na primer, na osnovu Musinih podataka iz tabele 9.3 možemo da predvidimo vreme sledećeg otkaza, pomoću proseka prethodna dva otkaza. To jest, iz tabele vidimo da je t_1 = 1, a t_2 = 30, pa predviđamo da će vreme do otkaza *T3* biti prosek: 31/2 = 15,5. Ovo izračunavanje možemo da nastavimo za svako očitano ti, tako da:

- za i = 4, imamo t_2 = 30 a t_3 = 113, pa će *T4* biti 71,5
- za i = 5, imamo t_3 = 113 a t_4 = 81, pa će *T5* biti 97

i tako dalje. Kao rezultat dobijamo sivu liniju označenu sa „av 2" na slici 9.9. Tehniku možemo da proširimo i uključimo više istorijskih podataka; slika takođe pokazuje da su naša predviđanja ista ako koristimo prethodnih 10 vremena (av 10) kao i kada koristimo prethodnih 20 (av 20).



SLIKA 9.9 Predviđanje vremena sledećeg otkaza na osnovu prethodne istorije

Međutim, istraživači predlažu složenije modele pouzdanosti u kojima se uzimaju u obzir naše pretpostavke o ponašanju softvera dok pronalazimo i ispravljamo greške. Na primer, u nekim modelima se pretpostavlja da su promene u ponašanju sistema iste, bez obzira na vrstu greške koja je ispravljena. U drugim modelima se uzima u obzir da se greške razlikuju, pa su zato i rezultati ispravljanja različiti. Na primer, imaćemo drugačiju funkciju gustine verovatnoće za svaku ispravku, pogotovo kada dolazi do rasta pouzdanosti. Kao što ističu Fenton i Pfleeger (1997), svaki sistem predviđanja mora da sadrži tri elementa:

- *model predviđanja* koji daje kompletnu specifikaciju verovatnoće za slučajni proces (tj. funkcije $Fi(Ti)$ i pretpostavljenu nezavisnost uzastopnih slučajeva);
- *proceduru zaključivanja* za nepoznate parametre modela zasnovanog na vrednostima za $t1$, $t2$,..., ti - 1;
- *proceduru predviđanja* kao kombinaciju modela i procedure zaključivanja za predviđanje budućeg ponašanja otkaza.

U ovom odeljku ispitujemo dva popularna modela predviđanja pouzdanosti; više modela i detaljnije informacije možete naći u knjizi Fentona i Pfleegera (1997).

Dobri modeli pouzdanosti eksplicitno tretiraju oba tipa neizvesnosti vezana za pouzdanost. Neizvesnost tipa 1 se uzima pod pretpostavkom da svaki otkaz nastupa slučajno, pa se vreme do sledećeg otkaza prikazuje eksponencijalnom raspodelom. Na primer, Motorola koristi eksponencijalnu raspodelu u svom testiranju na nula otkaza (podsetnik 9.5). Zato modele pouzdanosti razlikujemo prema načinu na koji tretiraju neizvesnost tipa 2.

PODSETNIK 9.5 MOTOROLINO TESTIRANJE NA NULA OTKAZA

Motorola koristi jednostavan model po imenu *testiranje na nula otkaza* izvedeno iz funkcije za učestanost otkaza (Brettschneider 1989). U modelu se polazi od pretpostavke da je broj otkaza do trenutka i jednako

$$ae^{-bi}$$

za konstantno a i b. Ovaj model možemo da upotrebimo za izračunavanje broja sati testiranja sistema koji obezbeđuje zadatu pouzdanost. Prema tome, model zahteva tri vrednosti: ciljni projektovani broj otkaza (*otkazi*), ukupan broj otkaza primećenih do sada u testiranju (*test-otkazi*) i ukupan broj sati testiranja do poslednjeg otkaza (*sati do poslednjeg otkaza*). Obrazac za broj sati testiranja do nula otkaza je:

$$\frac{[\ln(otkazi)/(0{,}5+otkazi)]\times(sati_do_poslednjeg_otkaza)}{\ln[(0{,}5+otkazi)/(test_otkazi+otkazi)]}$$

Na primer, pretpostavimo da testirate program od 33 hiljade redova. Do sada je došlo do 15 otkaza tokom ukupno 500 sati testiranja. Za poslednjih 50 sati nije prijavljen nijedan otkaz. Cilj vam je da garantujete maksimum od 0,03 otkaza na hiljadu redova koda. Na osnovu informacija koje imate, projektovani prosečan broj otkaza je 0,03 otkaza na 100 redova koda puta 33 000 redova koda, a to je 1. Ako upotrebimo prethodni obrazac, broj sati testiranja potreban da se ostvari cilj biće

$$\frac{[\ln(1/1{,}5)]\times 450}{\ln(1{,}5/16)} = 77.$$

Prema tome, trebalo bi da dostignete željeni stepen pouzdanosti ako bez novog otkaza testirate 77 sati nakon poslednjeg otkrivenog otkaza. Pošto ste već testirali 50 sati bez otkaza, potrebno je samo još 27 sati. Međutim, ako se u tom periodu pojavi otkaz, morate da nastavite testiranje, ponovite izračunavanje i vratite sat na početak.

Model Jelinski-Moranda. **Model Jelinski-Moranda** je najstariji i verovatno najpoznatiji model pouzdanosti (Jelinski i Moranda 1972). U njemu se polazi od pretpostavke da nema neizvesnosti tipa 2. To jest, u modelu se polazi od pretpostavke da su ispravke savršene (da ispravljaju grešku i pri tom ne uvode nove greške). Model Jelinski-Moranda takođe polazi od pretpostavke da svako ispravljanje greške jednako doprinosi unapređenju pouzdanosti.

Da bismo proverili da li model Jelinski-Moranda realno prikazuje otkaze, pretpostavimo da ispitujemo softver koji ima 15 grešaka, a da ispravljanje svake greške doprinosi povećanju stepena pouzdanosti za 0,003. U tabeli 9.4 navedena su prosečna vremena do i-tog otkaza i jedan simulirani skup vremena otkaza (dobijen upotrebom slučajnih brojeva u modelu). Kako i teži 15 (poslednjoj preostaloj grešci), vremena do otkaza postaju sve veća i veća. Drugim rečima, u drugoj koloni imamo prosečno vreme do i-tog otkaza na osnovu protekle istorije, a u trećoj koloni je predviđeno vreme do sledećeg (tj. i-tog) otkaza na osnovu modela Jelinski-Moranda.

Tabela 9.4 Redosled vremena otkaza po Jelinski-Morandu

i	Prosečno vreme do i-tog otkaza	Simulirano vreme do i-tog otkaza
1	22	11
2	24	41
3	26	13
4	28	4
5	30	30
6	33	77
7	37	11
8	42	64
9	48	54
10	56	34
11	67	183
12	83	83
13	111	17
14	167	190
15	333	436

U širokoj upotrebi je model **Musa** koji je zasnovan na modelu Jelinski-Moranda, samo što se vreme između otkaza meri prema vremenu izvršavanja. Vreme po satu se koristi za procenjivanje trenutka dostizanja ciljne pouzdanosti (Musa, Iannino i Okumoto 1990). Musa povezuje pouzdanost sa upravljanjem projektima i podstiče rukovodioce da koriste modelovanje pouzdanosti u raznim okruženjima, pogotovo za telekomunikacije.

Model Littlewood. **Model Littlewood** je realističniji od modela Jelinski-Moranda zato što doprinos pouzdanosti svake ispravljene greške tretira kao zasebnu slučajnu promenljivu. Uzima se da doprinosi imaju gama raspodelu. Littlewood za svoju raspodelu koristi dva izvora neizvesnosti, zato za njegov model kažemo da je **dvostruko stohastički**. Model Littlewood ima tendenciju pronalaženja i uklanjanja grešaka sa velikim doprinosom pouzdanosti u ranijim fazama, pa se sa nastavkom testiranja dobijaju sve manji doprinosi. Model Jelinski-Moranda koristi eksponencijalnu raspodelu vremena otkrivanja grešaka, dok se u modelu Littlewood koristi Pareto raspodela.

Značaj radnog okruženja

U poglavlju 13 porede se preciznosti nekoliko modela pouzdanosti. Ovde razmatramo uobičajenu pretpostavku da će model koji je bio precizan u prošlosti biti precizan i u budućnosti, ako su uslovi korišćenja isti. Naša predviđanja se često zasnivaju na greškama otkrivenim tokom testiranja. Međutim, okruženje u kojem testiramo možda ne odslikava stvarno ili uobičajeno korišćenje sistema.

Još je teže simulirati realnost kada postoje različiti režimi korišćenja sistema, različit stepen iskustva i različita radna okruženja. Na primer, početnik koji koristi paket sa radnim listovima ili paket za knjigovodstvo verovatno neće koristiti iste prečice sa tastature i složene tehnike koje upotrebljavaju iskusni korisnici; profili otkaza će im verovatno biti sasvim različiti.

Musa je ovaj problem uzeo u obzir tako što je upisao tipičnu interakciju korisnika i sistema u **operativni profil**, opisavši takođe i najverovatnije korisnički unete podatke u vremenu. Idealno bi bilo da operativni profil bude raspodela verovatnoće različitih ulaza.

Kada se strategija testiranja oslanja na operativni profil, podaci za testiranje reflektuju raspodelu verovatnoće.

Operativni profil se često pravi tako što se prostor ulaza deli na određen broj odvojenih klasa, pa se svakoj klasi pripisuje verovatnoća izbora ulaza iz te klase. Na primer, pretpostavite da program dozvoljava da izvršite jednu od tri različite opcije iz menija: *kreiranje, brisanje* i *izmenu*. Na osnovu testiranja sa korisnicima utvrdili smo da se opcija *kreiranje* bira dva puta češće od opcija *brisanje* ili *izmena* (a te dve se biraju jednako često). U tom slučaju možemo opciji *kreiranja* dodeliti verovatnoću 0,5, opciji *brisanje* 0,25 i opciji *izmena* takođe 0,25. Zatim u strategiji testiranja slučajno biramo ulaze u skladu sa ovim verovatnoćama.

Ova strategija **statističkog testiranja** ima bar dve prednosti:

1. Testiranje se koncentriše na one delove sistema koji će se najviše koristiti, pa bi trebalo da proizvede sistem koji će korisnicima izgledati pouzdaniji.
2. Predviđanja pouzdanosti zasnovana na rezultatima testa trebalo bi da nam daju precizno predviđanje pouzdanosti sa aspekta korisnika.

Međutim, nije lako pravilno izvršiti statističko testiranje. Ne postoji jednostavan niti ponovljiv način da se definišu operativni profili. Kasnije u ovom poglavlju videćemo kako se razvojem softvera u „čistoj sobi" statističko testiranje integriše u filozofiju izgradnje kvalitetnog softvera.

9.5 ZAVRŠNO TESTIRANJE PRIHVATLJIVOSTI

Kada se završi funkcionalno testiranje i testiranje performanse, tada znamo da sistem zadovoljava sve zahteve koji su navedeni u početnoj fazi razvoja softvera. Sledeći korak je da pitamo kupce i korisnike da li se oni sa tim slažu.

Svrha i uloge

Do sada smo kao učesnici u razvoju projektovali slučajeve za testiranje i obavljali sva testiranja. Sada kupac vodi testiranje i definiše slučajeve koji će se testirati. Svrha testa prihvatanja jeste da se kupcima i korisnicima omogući da utvrde da li sistem koji smo napravili zaista zadovoljava njihove potrebe i očekivanja. Prema tome, testove prihvatanja pišu, vode i procenjuju kupci, a pomoć učesnika u razvoju se javlja samo u slučaju kada kupac zahteva odgovor na neko tehničko pitanje. U testu prihvatanja obično veliku ulogu igraju oni službenici kupca koji su učestvovali u definisanju zahteva, zato što oni znaju kakvu vrstu sistema je kupac tražio.

Vrste testova

Postoje tri načina na koje kupac može da oceni sistem. U **referentnom testu** kupac priprema skup slučajeva za testiranje koji predstavljaju tipične uslove pod kojima će sistem raditi kada bude instaliran. Kupac procenjuje performansu sistema za svaki slučaj za testiranje. Referentni testovi se izvršavaju tako što stvarni korisnici ili poseban tim isprobavaju funkcije sistema. U svakom slučaju, ljudi koji testiraju upoznati su sa zahtevima i mogu da procene stvarnu performansu.

Referentni testovi se obično koriste kada kupac ima posebne zahteve. Dva ili više razvojnih timova dobijaju zadatak da naprave sisteme prema specifikacijama; jedan će sistem biti izabran i kupljen na osnovu uspeha referentnog testiranja. Na primer, kupac može da zatraži od dve komunikacione kompanije da instaliraju mrežu za prenos govora i podataka. Zatim se pravi referentni test za svaki sistem. Oba sistema možda zadovoljavaju zahteve, ali jedan je možda brži ili se lakše koristi od drugoga. Kupac odlučuje koji će da kupi zavisno od toga kako sistemi zadovolje kriterijume referentnog testa.

Kod **pilot testiranja** sistem se instalira na probnoj lokaciji. Korisnici rade sa sistemom kao da je on već trajno instaliran. Za razliku od referentnog testa gde se koristi skup posebnih slučajeva za testiranje koje primenjuju korisnici, pilot testiranje se oslanja na svakodnevni rad sistema kako bi se testirale sve funkcije. Kupac često priprema predloženi spisak funkcija koje svaki korisnik pokušava da uključi u tipičan svakodnevni posao. Međutim, pilot testiranje je daleko manje zvanično i strukturisano od referentnog testiranja.

Ponekad testiramo sistem uz pomoć korisnika iz naše vlastite organizacije ili kompanije pre nego što sistem izdamo kupcu; sami „pilotiramo" sistem pre nego što kupac izvrši stvarno pilot testiranje. Naš test se tu naziva **alfa test**, a kupčev pilot je **beta test**. Ovaj način je uobičajen kada sistem treba da se izda velikom broju kupaca. Na primer, nova verzija operativnog sistema imaće alfa test u našim prostorijama, a zatim beta test kod posebno izabrane grupe kupaca. Uvek pokušavamo da biramo kupce za beta test tako da oni budu reprezentativni za sve vrste korišćenja sistema. (U podsetniku 9.6 upozorava se na opasnosti od korišćenja beta verzije kao „pravog" sistema.)

Čak i kada se sistem razvija samo za jednog kupca, u pilot testiranje se obično uključuje samo jedan mali podskup kupčevih potencijalnih korisnika. Korisnike koji će učestvovati biramo tako da budu reprezentativni za većinu ostalih koji će kasnije koristiti sistem. Za testiranje sistema se može izabrati jedna lokacija ili organizacija i ne treba dozvoliti pristup svim korisnicima kojima je sistem namenjen.

Ako ovaj novi sistem zamenjuje postojeći ili predstavlja deo faznog razvoja, za prihvatanje se može koristiti treća vrsta testiranja. Kod **paralelnog testiranja** novi sistem funkcioniše paralelno sa prethodnom verzijom. Korisnici se postepeno privikavaju na novi sistem, ali i dalje koriste stari kao duplikat novog. Ova postepena tranzicija omogućava korisnicima da porede novi sistem sa starim. On takođe omogućava skepticima da steknu poverenje u novi sistem tako što porede rezultate dobijene od oba sistema, pa mogu da provere da li je novi sistem jednako efikasan i delotvoran kao i stari. Na neki način, paralelno testiranje predstavlja kombinaciju testa kompatibilnosti i funkcionalnog testa u izvođenju korisnika.

Rezultati testiranja prihvatljivosti

Izbor testa prihvatanja zavisi od vrste sistema koji se testira i od želje kupaca. U stvari, može se koristiti kombinacija nekih ili svih pristupa.

PODSETNIK 9.6 NEODGOVARAJUĆE KORIŠĆENJE BETA VERZIJE

Jula 1997. američka Državna uprava za aeronautiku i svemir imala je probleme sa uređajem za određivanje putanje Pathfindera koji je spuštao istraživački uređaj Sojourner na Mars. Softver je omogućavao da se uređaj spusti na Mars, da se oslobodi vozilo Sojourner i da se upravlja komunikacijama između Zemlje i uređaja. Međutim, zbog grešaka u upravljanju stekom i pokazivačima prilikom prebacivanja sa zadatka na zadatak, Pathfinder se stalno vraćao na početak i zbog toga je imao duže prekide u radu.

Sojourner je imao jednostavan kontroler za serijske zadatke 80C85, i radio je sasvim dobro. Ali Nasa je imala potrebu za složenijim softverom koji će upravljati složenijim funkcijama uređaja Pathfinder. Tokom dizajna, Nasa je prvo izabrala procesor, a tek potom softver koji će raditi na njemu. Zbog toga je Nasa izabrala IBM-ovu novu verziju procesora R6000 otpornu na radijacije, sličnu procesoru za PowerPC. 32-bitni čip bio je privlačan jer je koristio komercijalni operativni sistem u realnom vremenu, pa je tako izbegnut trošak izgradnje posebnog softvera. Zato je Nasa tražila operativni sistem za Pathfinder.

Postojalo je više dostupnih operativnih sistema i Nasa je izabrala VxWorks kompanije Wind River Systems (Alameda, California). U trenutku tog izbora VxWorks je bio u fazi testiranja i mogao se kupiti samo za PowerPC, ali još nije bila spremna posebna verzija za R6000. Zbog toga je Wind River Systems preneo verziju operativnog sistema za PowerPC u R6000 koristeći prednosti prenosivosti C koda. Taj je proizvod isporučen Nasi 1994.

Kada je verzija operativnog sistema VxWorks za R6000 stigla u Nasu, konfiguracija Pathfindera je zamrznuta na verziji 5.1.1 iako još nisu bili razrešeni značajni problemi sa operativnim sistemom. Prema tome, softver za Pathfinder je u stvari građen na beta verziji operativnog sistema, a ne na potpuno testiranom čvrstom operativnom sistemu (Coffee 1979).

Testiranje koje vrše korisnici ponekad se koristi kada očekivanja kupca izražena u zahtevima nisu sasvim jednaka onome što smo mi implementirali. Drugim rečima, završnim testiranjem prihvatljivosti kupcu se daje prilika da proveri da li smo napravili ono što je tražio. Ako je kupac zadovoljan, sistem se prihvata kao što je navedeno u ugovoru.

U stvarnosti test prihvatljivosti otkriva više od samih razlika u odnosu na zahteve. On takođe omogućava kupcima da utvrde šta zaista hoće, bez obzira na to da li je tako navedeno u zahtevima. Setite se da je u fazi analize zahteva kupac imao priliku da nam objasni problem koji treba da se reši, a da je dizajn sistema naš predlog rešenja. Tek kada kupci i korisnici dobiju priliku da rade sa sistemom koji treba da im reši problem oni mogu da vide da li je problem zaista rešen. U stvari, rad sa našim sistemom može da pomogne kupcima da otkriju aspekte problema kojih nisu bili svesni (pa čak i nove probleme).

U prethodnim poglavljima smo videli da brza izrada prototipa može da se upotrebi da bi kupac bolje shvatio rešenje pre nego što se implementira ceo sistem. Međutim, često nije praktično ili je skupo graditi prototipove. Štaviše, kada se grade veliki sistemi,

ponekad prođe dosta vremena između početne specifikacije i prvog prikazivanja bar jednog dela sistema. U tom međuvremenu mogu se na neki način promeniti i potrebe kupca. Na primer, savezno zakonodavstvo, ključno osoblje, pa čak i priroda kupčevog poslovanja mogu da se promene što će uticati na prirodu prvobitnog problema. Prema tome, promene u zahtevima mogu da nastupe ne samo zato što specifikacija nije bila tačna na početku razvoja, već takođe i zato što su kupci utvrdili da se problem promenio, pa je potrebno i drugačije rešenje.

Nakon testiranja prihvatljivosti kupac nas obaveštava koji zahtevi nisu zadovoljeni, a koje treba izbrisati, revidirati ili dodati zbog promenjenih uslova. Osoblje koje upravlja konfiguracijom identifikuje ove promene i evidentira potrebne modifikacije dizajna, implementacije i testiranja.

9.6 INSTALACIONI TEST

Konačnu rundu testiranja predstavlja instaliranje sistema na lokacijama korisnika. Ako je prihvatljivost testirana na lokaciji, instalacioni test možda nije ni potreban. Međutim, ako uslovi pod kojima je testirana prihvatljivost nisu bili identični uslovima na samoj lokaciji, neophodno je dodatno testiranje. Za početak instalacionog testa konfigurišemo sistem prema korisničkom okruženju. Povezujemo odgovarajući broj i vrste uređaja sa glavnim procesorom i uspostavljamo komunikacije sa drugim sistemima. Alociramo datoteke i dodeljujemo pristup odgovarajućim funkcijama i podacima.

Instalacioni testovi zahtevaju saradnju sa kupcem da bi se utvrdilo koji su testovi potrebni na lokaciji. Moguće je izvršiti regresiono testiranje kako bi se proverilo da li je sistem pravilno instaliran i da li i na licu mesta radi kao što je radio prilikom prethodnog testiranja. Test slučajevi potvrđuju kupcu da je sistem potpun i da su prisutne sve potrebne datoteke i svi uređaji. Testovi se usredsređuju na dve stvari: potpunost instaliranog sistema i proveru da li uslovi na lokaciji utiču na neke funkcionalne ili nefunkcionalne karakteristike. Na primer, sistem projektovan za rad na brodu mora da se testira da bi se prikazalo da na njega ne utiču loše vremenske prilike niti kretanje broda.

Kada kupac bude zadovoljan rezultatima, testiranje se završava i sistem se formalno isporučuje.

9.7 AUTOMATIZOVANO TESTIRANJE SISTEMA

Mnogi alati za testiranje koji su opisani u poglavlju 8 takođe su korisni i za testiranje sistema. Postoje i drugi koji su posebno projektovani za testiranje velikih grupa komponenti ili kao pomoć za istovremeno testiranje hardvera i softvera.

Simulacija nam omogućava da se usredsredimo na procenu jednog dela sistema tako što se imitiraju karakteristike ostalih delova sistema. **Simulator** predstavlja sistemu sve karakteristike uređaja ili sistema iako sam uređaj ili sistem nije dostupan. Kao što nam simulator letenja omogućava da naučimo da letimo iako nemamo avion, simulator uređaja vam omogućava kontrolu uređaja čak i kada uređaj nije tu. Ova situacija se naročito često događa kada se softver razvija izvan lokacije ili kada se uređaj razvija paralelno sa softverom.

Na primer, pretpostavite da prodavac gradi novi komunikacioni sistem koji se sastoji od hardvera i od softvera u isto vreme dok softverski inženjeri razvijaju upravljački program za njega. Nemoguće je testirati nedovršen uređaj, pa se koriste specifikacije uređaja i pravi simulator koji će nam omogućiti da testiramo očekivane interakcije.

Slično tome, simulator je posebno koristan ako se neki poseban uređaj nalazi na lokaciji kupca ili korisnika, a testiranje se vrši na nekom drugom mestu. Na primer, ako gradite sistem za upravljanje automobilom, za testiranje vam neće biti potreban sam automobil; vaš sistem može umesto toga da koristi simulator automobila. U stvari, ponekad je simulator uređaja korisniji od samog uređaja zato što simulator može da čuva podatke o stanju uređaja u različitim fazama testiranja. Zatim simulator može da izvesti o svom stanju u trenutku prekida što će nam pomoći da pronađemo grešku koja je do prekida dovela.

Simulatori se takođe koriste da izgledaju kao drugi sistemi sa interfejsom prema testiranom sistemu. Ako se šalju poruke ili se pristupa bazi podataka, simulator obezbeđuje informacije potrebne za testiranje bez potrebe da se duplira ceo sistem. Simulator takođe pomaže za testitiranje opterećenja i testiranje kapaciteta jer može da se isprogramira da optereti sistem velikim količinama podataka, zahteva ili korisnika.

Uglavnom, simulatori omogućavaju kontrolu nad uslovima testiranja. Ta kontrola vam dozvoljava da izvršite testiranja koja bi inače bila opasna ili nemoguća. Na primer, testiranje sistema za upravljanje projektilom može biti mnogo jednostavnije i bezbednije ako se koriste simulatori.

Automatizacija može pomoći i u projektovanju slučajeva za testiranje. Na primer, Cohen i dr. (1996) opisuju automatski generator efikasnog testiranja (AETG) razvijen u kompaniji Bellcore u kojem se koriste kombinatorne tehnike projektovanja za proizvodnju slučajeva za testiranje. U njihovom kombinatoričkom pristupu projektovanju oni proizvode testove koji pokrivaju sve dvostruke, trostruke ili *n*-tostruke kombinacije parametara koji se testiraju. Na primer, da bi se pokrile sve dvostruke kombinacije, ako je x_1 važeća vrednost za jedan parametar, a x_2 za drugi, tada postoji test slučaj u kome je prvi parametar x_1, a drugi x_2. U jednom eksperimentu su zahtevi testiranja za konačno izdanje imali 75 parametara sa 10^{29} mogućih kombinacija. Uz pomoć AETG-a istraživači su napravili samo 28 testova koji su pokrili sve dvostruke kombinacije parametara. U drugom eksperimentu je njihova tehnika generisala testove sa boljim pokrivanjem blokova i odlučivanja nego slučajno testiranje. Postoji i treća studija koja pokazuje da je automatizovani sistem doveo do otkrivanja značajnih grešaka u zahtevima i kodu koje nisu otkrivene pomoću drugih metoda testiranja. U podsetniku 9.7 opisan je još jedan primer u kome je automatizacija testiranja ubrzala proces testiranja.

9.8 TEST DOKUMENTACIJA

Testiranje može biti složeno i teško. Na teškoće mogu da utiču softver i hardver sistema, kao i postupci vezani za upotrebu sistema. Osim toga, distribuirani sistem ili sistem sa radom u realnom vremenu zahtevaju veliku pažnju u praćenju i vremenskom raspoređivanju podataka i procesa da bi se mogli doneti zaključci o performansi. Konačno, kod velikih sistema, koordinacija je otežana zbog velikog broja ljudi koji su uključeni u razvoj i testiranje. Da bi se složenost i teškoće prilikom testiranja držale pod kontrolom, koristimo potpuno i pažljivo projektovane dokumente o testiranju.

PODSETNIK 9.7 AUTOMATIZOVANO TESTIRANJE SISTEMA ZA PONUDU OSIGURANJA VOZILA

Mills (1997) opisuje kako njegova kompanija koristi automatizaciju prilikom testiranja sistema za ponudu osiguranja motornih vozila. Svaki sistem sadrži profile rizika iz približno 90 osiguravajućih kompanija i proizvoda tako da broker može da ubaci informacije o automobilu i njegovom vozaču i dobije ponude za premije osiguranja. Ulaz se sastoji od 50 polja, kao što su starost, vozačko iskustvo, područje u UK, vrsta upotrebe, veličina motora i broj vozača. Te informacije služe da se pronađe ponudač u jednom od 20 područja, za jednu od preko 20 grupa vozila, 5 klasa upotrebe, 3 tipa osiguranja i 15 starosnih grupa. Sistem ponuda prati 14 proizvoda u 10 sistema osiguranja gde se svaki sistem menja bar jednom mesečno.

Zato je broj slučajeva za testiranje koji je potreban da bi se temeljito testirao sistem ponuda veoma velik, a procesa testiranja se velikim delom svodi na odlučivanje koliko će slučajeva biti dovoljno. Bates (1997) iznosi proračune koji pokazuju da je testiranje 5.000 uslova za jedan sistem u državnoj Vestminsterskoj banci zahtevalo 21.000 skriptova. Pošto je za svaki skript potrebno 3 minuta ručnog testiranja, bilo bi potrebno 7,5 meseci kad bi to radila jedna osoba na jednoj platformi! Ta situacija naravno nije prihvatljiva za sistem osiguranja koji opisuje Mills, a koji zahteva još više uslova i skriptova za test. Učesnici u razvoju su procenili da bi mogli da testiraju najviše 100 do 200 slučajeva u paketskom režimu, a ljudi iz osiguravajućeg zavoda su naredili razvijaocima da izvrše 100 slučajnih probnih ponuda. Ali, uz pomoć automatizovanog testiranja treća kompanija izvršava 30.000 planiranih probnih ponuda po klijentu za svaki sistem ponuda svakog meseca, a ti postupci testiranja se obave za manje od nedelju dana! Mills izveštava da je najveća razlika između automatizovanog i ručnog testiranja, osim brzine, u tome što se mnoge greške pronalaze ranije u postupku testiranja čime se dobija više vremena da se one isprave pre nego što se izda sledeća verzija sistema.

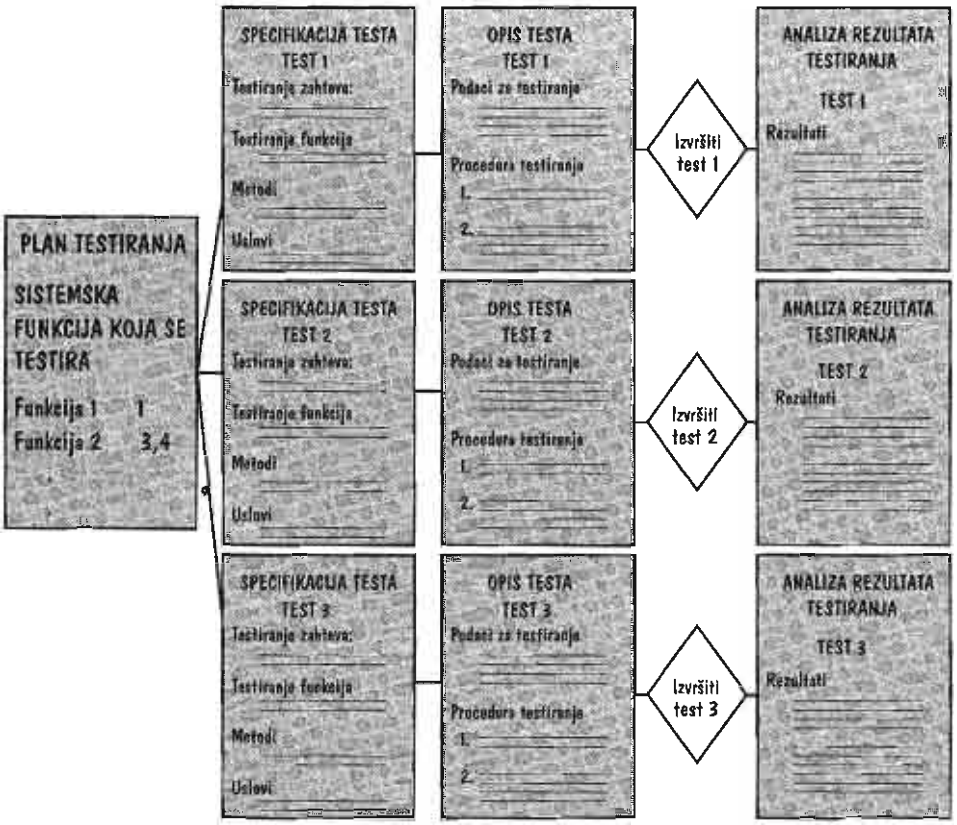
Potrebno je nekoliko vrsta dokumentacije. **Plan testiranja** opisuje sam sistem i plan za izvršavanje svih funkcija i karakteristika. **Specifikacija i ocena testova** opisuje svaki test i definiše kriterijume za ocenu svakog svojstva koje se testira. Zatim, **opis testova** predstavlja podatke koji se testiraju i procedure pojedinačnih testova. Konačno, **izveštaj o analizi testova** opisuje rezultate svih testova. Na slici 9.10 opisane su veze između dokumenata i procesa testiranja.

Planovi testiranja

U poglavlju 8 opisali smo ulogu plana testiranja u postavljanju šeme za sve aktivnosti u testiranju. Sada ćemo pogledati kako se plan testiranja može upotrebiti za usmeravanje testiranja sistema.

Na slici 9.11 prikazane su komponente plana testiranja. Plan počinje od navođenja ciljeva koji bi trebalo:

- da usmere testiranje;
- da usmere tehnički deo rada neophodan tokom testiranja;



SLIKA 9.10 Dokumenta proizvedena tokom testiranja

- da utvrde plan i raspored testiranja i definišu potrebnu opremu, organizaciju, metode testiranja, očekivane rezultate i orijentaciju korisnika;
- da opišu prirodu i obim svakog testa;
- da objasne kako će testovi u potpunosti da procene funkcionisanje i performansu sistema;
- da dokumentuju ulazne podatke, specifične procedure testiranja i očekivane rezultate.

Zatim, plan navodi ostale važne dokumente proizvedene tokom razvoja. Konkretno, plan objašnjava veze između dokumenata sa zahtevima, dizajnerske dokumentacije, dokumentacije u vezi sa i kodom komponenti sa jedne strane, i procedura testiranja sa druge. Na primer, može da se napravi šema imenovanja ili numeracija koja povezuje sve dokumente tako da se, recimo, zahtev 4.9 obrađuje u komponentama dizajna 5.3, 5.6 i 5.8, a testira se u proceduri 12.3.

Nakon ovih uvodnih odeljaka sledi kratak pregled sistema. Pošto oni koji čitaju plan testiranja možda nisu bili uključeni u prethodne faze razvoja, kratak pregled sistema postavlja kontekst za raspored i događaje testiranja. Kratak pregled ne mora da bude detaljan; to može da bude crtež koji prikazuje glavne ulaze u sistem i izlaze iz njega sa opisom glavnih transformacija.

SLIKA 9.11 Delovi plana testiranja



Pošto je testiranje smešteno u kontekst sistema, plan opisuje glavne testove i pristupe testiranju koji će se koristiti. Na primer, u planu testiranja se razlikuju funkcionalni testovi, testovi performanse, testovi prihvatanja i instalacioni testovi. Ako se funkcionalni testovi mogu dalje podeliti po nekom kriterijumu (kao što su podsistemi), plan testiranja postavlja sveukupnu organizaciju testiranja.

Pošto se objasni testiranje komponenti, plan prelazi na raspored događaja. U rasporedu se navode mesto testiranja i vremenski raspored. Raspored testiranja koji se često opisuje kao šema kontrolnih tačaka ili graf aktivnost, trebalo bi da sadrži:

1. Ukupno trajanje testiranja.
2. Glavne podele testiranja i njihova vremena početka i završetka.
3. Eventualne preduslove testiranja (na primer,, upoznavanje sa sistemom, obuka korisnika, proizvodnja podataka za testiranje) i vreme potrebno za svaki od njih.
4. Vreme potrebno za pripremu i pregled izveštaja o testiranju.

Ako testiranje treba da se vrši na nekoliko lokacija, plan će imati posebne rasporede za svaku lokaciju. Dijagram prikazuje hardver, softver i osoblje potrebno za obavljanje testova na svakoj lokaciji, kao i vremenski interval u kojem će svaki od resursa biti potreban. Navode se takođe i posebne potrebe za obukom ili održavanjem.

Plan navodi i materijal za testiranje koji se sastoji od materijala koji treba doneti (na primer, korisničke ili operatorske priručnike, listinge, trake) i materijala na lokaciji (na primer, posebni aparati za testiranje, tabele baze podataka, memorijski medijumi). Primera radi, ako u testiranju treba da se koristi sistem za upravljanje bazom podataka da bi se napravila baza podataka sa primerima, za test se može zahtevati da korisnici na lokaciji definišu elemente podataka pre nego što stigne testerski tim. Slično tome, ako su timu potrebne određene mere bezbednosti ili tajnosti, osoblje na lokaciji testiranja možda treba pre testiranja da pripremi lozinke ili da uspostavi posebne pristupe.

Specifikacije i ocene testiranja

Plan testiranja opisuje opštu podelu testiranja na pojedinačne testove koji se odnose na konkretne stavke. Na primer, ako je u sistemu koji testiramo obrada raspoređena na nekoliko računara, funkcionalni testovi i testovi performanse mogu se dalje podeliti na testove za svaki podsistem.

Za svaki pojedinačni test pravi se posebna specifikacija i procena. U specifikaciji prvo navodimo one zahteve čiju realizaciju test treba da prikaže. Ovaj odeljak se poziva na dokumente zahteva i objašnjava svrhu testa.

Jedan od načina da se uspostavi veza između zahteva i testova jeste tabela ili šema kao u tabeli 9.5. Obratite pažnju na to da zahtevi koji su navedeni na vrhu tabele sadrže broj dokumenta zahteva; svaka funkcija sa leve strane zahtevana je u dokumentu u čijoj koloni se nalazi slovo X.

U tabeli su nabrojane funkcije sistema koje će se prikazivati u testu. Testovi performanse mogu se opisati na sličan način. Umesto da se navedu funkcionalni zahtevi, u tabeli se navode zahtevi u pogledu brzine pristupa, bezbednosti baze podataka i tako dalje.

Često se test sastoji od kolekcije manjih testova koji svi zajedno dokazuju ispunjavanje zahteva. U tom slučaju specifikacija testa pokazuje vezu manjih testova sa zahtevom.

Tabela 9.5 Šema povezanosti testova i zahteva

Test	Zahtev 2.4.1: Izrada i održavanje baze podataka	Zahtev 2.4.2: Selektivno vađenje podataka	Zahtev 2.4.3: Specijalizovani izveštaji
1. Dodavanje novog zapisa	X		
2. Dodavanje polja	X		
3. Menjanje polja	X		
4. Brisanje zapisa	X		
5. Brisanje polja	X		
6. Pravljenje indeksa		X	
Pronaći zapis sa trženim			
7. brojem ćelije		X	
8. visinom vode		X	
9. visinom krošnji		X	
10. pokokrivenošću tla		X	
11. brzinom oceđivanja		X	
12. Štampati celu bazu podataka			X
13. Štampati direktorijum			X
14. Štampati ključne reči			X
15. Štampati rezime simulacije			X

Za svaki test postoji određena filozofija i skup prihvaćenih metoda. Međutim, filozofija i metodi su možda ograničeni drugim zahtevima i realnošću situacije u kojoj se vrši testiranje. U specifikaciji se navode i ti uslovi testiranja. Uslovi mogu biti neki od sledećih:

- Da li sistem koristi stvarne ulazne podatke od korisnika ili uređaja, ili posebne slučajeve proizvodi neki program ili surogat uređaja?
- Koji su kriterijumi za pokrivenost testiranjem?
- Kako će se podaci beležiti?
- Da li za testiranje postoje ograničenja u smislu vremena, interfejsa, opreme, osoblja, baze podataka ili nečeg drugog?
- Ako se test sastoji od niza manjih testova, kojim redom ih treba izvršiti?

Ako podatke testiranja treba pre ocenjivanja obraditi, specifikacija testiranja opisuje tu obradu. Na primer, kada sistem proizvodi velike količine podataka, ponekad se koriste tehnike za svođenje podataka tako da rezultat bude praktičniji za procenu.

Uz svaki test se navodi na koji način će se proceniti da li je on kompletan. Zato se na kraju svake specifikacije opisuje kako ćemo znati da je test završen i da su ispunjeni relevantni zahtevi. Na primer, plan navodi raspon izlaznih rezultata koji zadovoljavaju zahteve.

Nakon kriterijuma završetka daje se metod za procenu. Na primer, podaci dobijeni tokom testiranja možda će se sakupiti i ručno urediti da bi ih zatim pregledao tim koji obavlja testiranje. Alternative mogu biti da tim koristi automatizovani alat za procenu nekih podataka i pregleda zbirne izveštaje ili da stavku po stavku poredi sa očekivanim rezultatima. Za procenjivanje potpunosti testiranja mogu se upotrebiti mere efikasnosti i delotvornosti opisane u podsetniku 9.8.

Opis testa

Opis testa se piše za svaki test definisan u specifikaciji testa. Dokument opisa testa koristimo za usmeravanje testiranja. Ovi dokumenti moraju biti detaljni i jasni i moraju da sadrže:

- sredstva kontrole;
- podatke;
- procedure.

Dokument započinje opštim opisom testa. Zatim navodimo da li će se testiranje pokretati i kontrolisati automatski ili ručno. Na primer, podaci mogu ručno da se unose sa tastature, ali će zatim automatizovani upravljački program izvršavati funkcije koje testiramo. Druga alternativa je da ceo postupak bude automatizovan.

Podaci testiranja mogu da se svrstaju u nekoliko grupa: ulazni podaci, ulazne komande, ulazna stanja, izlazni podaci, izlazna stanja i poruke dobijene od sistema. Svaka od njih se detaljno opisuje. Na primer, daju se ulazne komande da bi tim znao kako da pokrene test, kako da ga zaustavi ili suspenduje, kako da ponovi ili nastavi test koji nije bio uspeo ili nije bio potpun i kako da prekine test. Slično tome, tim mora da protumači poruke da bi utvrdio status sistema i kontrolisao testiranje. Ovde opisujemo kako će tim da utvrdi da li je otkaz nastao zbog ulaznih podataka, zbog neispravnog postupka testiranja ili zbog neispravnog rada hardvera (kad god je to moguće).

PODSETNIK 9.8 MERENJE EFIKASNOSTI I DELOTVORNOSTI TESTA

Jedan aspekt planiranja i izveštavanja o testiranju jeste merenje delotvornosti testa. Graham (1996b) predlaže da se delotvornost testa meri tako što se podeli broj grešaka pronađenih u datom testu sa ukupnim brojem pronađenih grešaka (uključujući i greške pronađene nakon tog testa). Na primer, uzmimo da je integracionim testom pronađeno 56 grešaka, a da je u sveukupnom procesu testiranja pronađeno 70 grešaka. U tom slučaju prema Grahamovoj meri delotvornosti značilo bi da je integracioni test bio 80% delotvoran. Međutim, uzmimo da je sistem isporučen nakon što je pronađeno 70 grešaka, a da je tokom prvih 6 meseci rada pronađeno još 70 grešaka. Tada bi integracioni test bio zaslužan za pronalaženje 56 od ukupno 140 grešaka, pa bi test bio delotvoran samo sa 40%.

Ovaj način ocenjivanja učinka konkretne faze ili tehnike testiranja može da se doteruje na više načina. Na primer, može se svakom otkazu dodeliti stepen ozbiljnosti, pa se delotvornost testova zatim računa prema stepenu. Na ovaj način bi integraciono testiranje bilo delotvorno sa 50% u pronalaženju grešaka koje dovode do kritičnih otkaza, a 80% delotvorno u pronalaženju grešaka koje dovode do manjih otkaza. Druga alternativa bi bila da se delotvornost testova kombinuje sa analizom osnovnog uzroka, pa bismo mogli da opisujemo delotvornost u pronalaženju grešaka što je ranije moguće u toku razvoja. Na primer, integraciono testiranje bi moglo da pronađe 80% grešaka, ali pola tih grešaka je moglo biti pronađeno ranije, na primer tokom pregleda dizajna, zbog toga što je bilo reči o problemima u dizajnu.

Efikasnost testiranja se računa tako što se broj grešaka pronađenih tokom testiranja deli sa radom potrebnim da se izvrši testiranje, pa se dobija vrednost izražena u broju grešaka po čovek/satu. Mere efikasnosti nam pomažu da izračunamo troškove pronalaženja grešaka, kao i relativne troškove pronalaženja grešaka u različitim fazama procesa testiranja.

Mere delotvornosti i efikasnosti mogu da budu korisne za planiranje testiranja, želja nam je da na osnovu protekle istorije testiranja postignemo što veću delotvornost i efikasnost. Prema tome, dokumentacija u vezi sa trenutnim testiranjem trebalo bi da sadrži merenja koja će nam omogućiti da izračunamo delotvornost i efikasnost.

Na primer, podaci o testiranju rutine SORT mogli bi da budu sledeći:

```
INPUT DATA:
Ulazni podaci će se dobijati od programa LIST. Program
generiše slučajan spisak od N reči od alfanumeričkih znakova;
svaka reč je dužine M. Program se poziva sa
   RUN LIST(N,M)
u pokretaču testiranja. Izlaz se smešta u globalno područje
podataka
LISTBUF. Skupovi podataka za testiranje za ovaj test su
sledeći:
Slučaj 1: Upotrebiti LIST sa parametrima N=5, M=5
Slučaj 2: Upotrebiti LIST sa parametrima N=10, M=5
Slučaj 3: Upotrebiti LIST sa parametrima N=15, M=5
Slučaj 4: Upotrebiti LIST sa parametrima N=50, M=10
Slučaj 5: Upotrebiti LIST sa parametrima N=100, M=10
Slučaj 6: Upotrebiti LIST sa parametrima N=150, M=10
```

```
INPUT COMMANDS:
Rutina SORT se poziva komandom
    RUN SORT (INBUF,OUTBUF) ili
    RUN SORT (INBUF)

OUTPUT DATA:
Ako se upotrebe dva parametra, sortirani spisak se smešta u
OUTBUF. Inače će se vratiti u INBUF.

SYSTEM MESSAGES:
Dok traje sortiranje, prikazuje se sledeća poruka:
    "Sorting...please wait..."
Po završetku, SORT prikazuje na ekranu sledeću poruku:
    "Sorting completed"
Da bi se testiranje prekinulo pre završne poruke, pritisnuti
CONTROL-C na tastaturi.
```

Procedura testiranja se ponekad naziva **skript testa** pošto nam daje uputstvo kako da korak po korak izvršimo testiranje. Strogo definisan skup koraka obezbeđuje kontrolu nad testiranjem, tako da možemo da ponovimo uslove i ponovo dovedemo do otkaza, ako se to pokaže potrebnim dok budemo pokušavali da otkrijemo uzrok problema. Ako se test prekine iz nekog razloga, moramo da budemo u stanju da ga nastavimo, a da se ne vratimo na početak.

Na primer, deo skripta za testiranje funkcije „promena polja" (navedene u tabeli 9.5) mogao bi da izgleda ovako:

```
Korak N:     Funkcijski taster 4: Pristup datoteci podataka.
Korak N+1:   Na ekranu se javlja zahtev da se upiše ime
             datoteke podataka. Type "sys:test.txt"
Korak N+2:   Meni se pojavljuje i glasi
                 * brisati datoteku
                 * menjati datoteku
                 * preimenovati datoteku
             Postaviti kursor pored "menjati datoteku" i
             pritisnuti taster RETURN.
Korak N+3:   Na ekranu se javlja zahtev da se upiše broj
                 zapisa. Upisati '4017'.
Korak N+4:   Ekran će se ispuniti poljima podataka iz zapisa
                 4017:
                 Broj zapisa: 4017          X: 0042 Y: 0036
                 Tip zemljišta: glina       Filtriranje: 4
                                                 mtr/sat
                 Vegetacija: kudzu          Visina zastora:
                                                 25 mtr
                 Vodeni sto: 12 mtrs        Konstrukc:
                                                  spoljna
                 Šifra održavanja: 3T/4F/9R
Korak N+5:   Pritisnuti funkcijski taster 9: promena
```

```
Korak N+6:   Stavke na ekranu su istaknute. Postavite kursor
                 na polje VEGETACIJA. Upišite 'trava' preko
                'kudzu' pa pritisnite taster RETURN.
Korak N+7:   Stavke na ekranu neće više biti istaknute.
                 polje VEGETACIJA bi sada trebalo da glasi
                 'trava'.
Korak N+8:   Pritisnuti funkcijski taster 16: Povratak na
                 prethodni ekran.
Korak N+9:   Meni se pojavljuje i glasi
                 * brisati datoteku
                 * menjati datoteku
                 * preimenovati datoteku
             Da bi se proverilo da li je promena evidentirana,
             postaviti kursor pored 'menjati datoteku' i
             pritisnuti taster RETURN.
Korak N+10:  Na ekranu se javlja zahtev da se upiše broj
                 zapisa. Upisati '4017'.
Korak N+11:  Ekran će se ispuniti poljima podataka iz zapisa 4017:
                 Broj zapisa: 4017      X: 0042 Y: 0036
                 Tip zemljišta: glina   Filtriranje: 4 mtr/sat
                 Vegetacija: trava      Visina zastora: 25 mtr
                 Vodeni sto: 12 mtrs    Konstrukc: spoljna
                 Šifra održavanja: 3T/4F/9R
```

Koraci u skriptu za test su numerisani, a podaci u vezi sa svakim korakom su referencirani. Ako opisi nisu na drugom mestu, ovde opisujemo kako treba pripremiti podatke i lokaciju za test. Na primer, mogu se dati detalji za potrebna podešavanje opreme, definicije baze podataka i za komunikacione veze. Zatim, skript tačno opisuje šta treba da se dogodi za vreme testiranja. Koji taster treba da se pritisne, šta se pojavljuje na ekranu, kakav izlaz se dobija, reagovanje opreme i sve ostale manifestacije. Objašnjavamo očekivani efekat ili izlaz i dajemo operatoru ili korisniku uputstva šta treba da radi ako stvarni rezultat nije jednak očekivanom.

Konačno, opis testa objašnjava redosled aktivnosti potrebnih za završetak testa. Te aktivnosti mogu da se sastoje od čitanja ili štampanja kritičnih podataka, prekidanja automatizovanih procedura ili isključivanja delova opreme.

Izveštaj sa analizom rezultata testiranja

Kada se test izvrši, analiziramo rezultat da bismo utvrdili da li funkcije ili performanse koje smo testirali ispunjavaju zahteve. Ponekad je dovoljno samo prikazati funkciju. Međutim, najčešće za funkciju postoje ograničenja performanse. Na primer, nije dovoljno utvrditi da neka kolona može da se sortira ili da se sabere. Moramo da izmerimo brzinu izračunavanja i da obratimo pažnju na ispravnost. Zbog toga je izveštaj o analizi testa potreban iz nekoliko razloga:

- On dokumentuje rezultate testa.
- Ako dođe do otkaza, izveštaj sadrži informacije potrebne da se ponovi otkaz (ako je to potrebno) da bi se pronašao i ispravio uzrok problema.

- On sadrži neophodne informacije za utvrđivanje da li je projekat razvoja završen.
- On obezbeđuje poverenje u performansu sistema.

Izveštaj o analizi testa mogu da čitaju ljudi koji nisu uključeni u postupak testiranja, ali su na druge načine upoznati sa sistemom i njegovim razvojem. Zbog toga izveštaj sadrži kratak opšti pregled projekta, njegove ciljeve i relevantne reference za ovaj test. Na primer, izveštaj o testiranju sadrži one delove iz zahteva, dizajna i implementacione dokumentacije u kojima se opisuju funkcije koje će se izvršiti u ovom testiranju. Izveštaj takođe ukazuje na delove plana testiranja i specifikacione dokumentacije u kojima se pominje ovaj test.

Kada se teren ovako pripremi, u izveštaju o analizi testa navode se funkcije i karakteristike performanse koje je trebalo prikazati i opisuju stvarni rezultati. Prikazuju se rezultati merenja funkcija, performanse i podataka, sa napomenama da li su ostvareni ciljni zahtevi. Ako se otkrije greška ili nedostatak, u izveštaju se opisuje njegov uticaj. Ponekad rezultate testa procenjujemo merom ozbiljnosti. Ova mera pomaže testerskom timu daodluči da li će nastaviti testiranje ili će sačekati dok se greška ne otkloni. Na primer, ako se poremećaj sastoji od nepravilnog znaka u gornjem delu ekrana, testiranje može da se nastavi dok se uzrok traži i ispravlja. Međutim, ako greška dovodi do toga da sistem padne ili da se datoteka izbriše, tim bi mogao da odluči da se testiranje prekine dok se greška ne otkloni.

Obrasci za izveštavanje o problemima

Setićete se da smo u poglavlju 1 rekli da je greška takav problem u nekom sistemskom proizvodu koji može da dovede do otkaza sistema; grešku vidi učesnik u razvoju, a korisnik će doživeti otkaz. Tokom testiranja beležimo podatke o greškama i otkazima u **obrazac za izveštavanje o problemima**. **Obrazac za izveštavanje o odstupanju** je izveštaj o problemu u kojem se opisuju problemi u vezi sa ponašanjem ili atributima sistema koji se razlikuju od očekivanih. Opisuje se šta se očekivalo, šta se umesto toga dogodilo i okolnosti koje su dovele do otkaza. **Obrazac za izveštaj o grešci** objašnjava kako je greška pronađena i ispravljena, posle podnošenja obrasca za izveštavanje o odstupanju.

Svaki obrazac za izveštavanje o problemu bi trebalo da sadrži odgovore na nekoliko pitanja u vezi sa problemom o kojem je reč:

- *Lokacija:* Gde je došlo do problema?
- *Vreme*: Kada je nastupio problem?
- *Simptom*: Šta smo primetili?
- *Konačni rezultat*: Kakve su bile posledice?
- *Mehanizam*: Kako je došlo do problema?
- *Uzrok*: Zašto je došlo do problema?
- *Ozbiljnost*: Koliko je problem uticao na korisnika ili na poslovanje?
- *Cena:* Koliko je to koštalo?

Na slici 9.12 imamo primer pravog obrasca za izveštavanje o grešci iz jednog engleskog preduzeća. Obratite pažnju na to da svaka greška ima broj i da učesnici u razvoju evidentiraju datum kada su obavešteni da je došlo do problema, kao i datum kada je pronađen i otklonjen uzrok problema. Pošto učesnici u razvoju ne rešavaju odmah svaki problem (na primer, u slučaju da drugi problemi imaju veći prioritet) oni takođe evidentiraju stvarni broj časova koji je bio potreban da se popravi ova konkretna greška.

Broj greške	Ned. prijave	Podr. sistema	Vrsta greške	Nedelja rešenja	Trajanje popravke
...	...	...	...	...	...
F254	92/14	C2	P	92/17	5.5

SLIKA 9.12 Obrazac za izveštaj o greškama

Međutim, obrazac za izveštavanje o greškama bi trebalo da sadrži još mnoge druge podatke. Uglavnom, obrazac za izveštavanje o grešci trebalo bi da sadrži sva pitanja i sledeću vrstu detaljnih informacija (Fenton i Pfleeger 1997):

- *Lokacija:* Identifikator unutar sistema, kao što je ime modula ili dokumenta.
- *Vreme:* Faza razvoja u kojoj je greška napravljena, otkrivena i ispravljena.
- *Simptom:* Vrsta poruke o grešci ili aktivnost koja je dovela do otkrivanja greške (na primer, testiranje, pregled, inspekcija).
- *Konačni rezultat:* Otkazi koje je greška izazvala.
- *Mehanizam:* Kako je izvor greške stvoren, otkriven i ispravljen.
- *Uzrok:* Vrsta ljudskog propusta koji je doveo do greške.
- *Ozbiljnost:* Odnosi se na ozbiljnost otkaza do kojeg je došlo ili do kojeg je moglo da dođe.
- *Cena:* Vreme ili rad potrebni za pronalaženje i ispravljanje; može da obuhvati i analizu troškova koji bi nastali da je greška otkrivena u ranijim fazama razvoja.

Primetićete da svaki od ovih aspekata greške reflektuje shvatanje učesnika u razvoju o uticaju ove greške na ceo sistem. S druge strane, izveštaj o odstupanju bi trebalo da reflektuje korisnikov doživljaj otkaza do kojeg je greška dovela. Pitanja su ista, ali odgovori se veoma razlikuju:

- *Lokacija:* Instalacija na kojoj je primećen otkaz.
- *Vreme*: CPU vreme, vreme po satu ili neka druga relevantna mera vremena.
- *Simptom*: Vrsta poruke o grešci ili manifestacija otkaza.
- *Konačni rezultat*: Opis otkaza, kao što je „pad operativnog sistema", „pad kvaliteta usluge", „gubitak podataka", „pogrešan izlaz" ili „nema izlaza".
- *Mehanizam*: Lanac događaja, uključujući komande sa tastature i podatke o stanju neposredno pre otkaza.
- *Uzrok*: Referenca na moguće greške koje su dovele do otkaza.
- *Ozbiljnost*: Uticaj na korisnika ili na poslovanje.
- *Cena:* Cena ispravljanja plus cena propuštene potencijalne poslovne dobiti.

Na slici 9.13 nalazi se pravi obrazac za izveštaj o odstupanju, pogrešno nazvan izveštaj o „grešci". Pomenuta su mnoga pitanja sa našeg spiska i otkazi su dosta dobro opisani. Međutim, navode se samo stavke koje su promenjene, ali ne i opis stvarnog uzroka otkaza. Idealno bi bilo da se u ovom izveštaju pozove i na neki izveštaj o greškama da bi se moglo znati koje greške su dovele do kojih otkaza.

U obrascima za izveštaje o problemima potrebne su nam potpunije informacije da bismo mogli da procenimo delotvornost i efikasnost testiranja i razvojne prakse. Pogotovo kada imamo ograničene resurse, istorijske informacije zabeležene u izveštajima o

problemima pomažu nam da shvatimo u kojim aktivnostima lako dolazi do grešaka i koji su postupci korisni za njihovo pronalaženje i ispravljanje.

IZVEŠTAJ O GREŠCI **S.P0204.6.10.3016**

IZVESTILAC: Joe Bloggs

KRATAK NAZIV: Izuzetak 1 u 620. redu dps_cc.c koji pokreće NAS

PUNI OPIS Pokrenuto ispitivanje izdržljivosti NAS, radio je nekoliko minuta. Onemogućili aktivni NAS link (emulator prebačen na pauzu linka), zatim ponovo omogućili onemogućeni link nakon čega je došlo do navedenog izuzetka. (Ja mislim da je ponovno omogućavanje samo prividan razlog.) (tokom punjenja baze podataka)

DODELJENO NA REŠAVANJE: DATUM:

KATEGORIZACIJA: 0 ①2 3 Design Spec Docn
POSLATI KOPIJE UZ OBAVEŠTENJE ZA:
PROCENITELJ: DATUM: 8/7/92

ID KONFIGURACIJE	DODELJENO	DEO
dpo_s.c		

KOMENTARI: dpo_s.c izgleda pokušava da koristi nedozvoljen CID, umesto da odbaci poruku. AWJ

PROMENJENE STAVKE

ID KONFIGURACIJE	IMPLEMENTOR/DATUM	PREGLEDAČ/DATUM	BROJ IZDANJA	INTEGRATOR/DATUM
dpo_s.c v.10	AWJ 8/7/92	MAR 8/7/92	6.120	RA 8-7-92

KOMENTARI:

ZATVORENO

KONTROLOR GREŠAKA: DATUM: 9/7/92

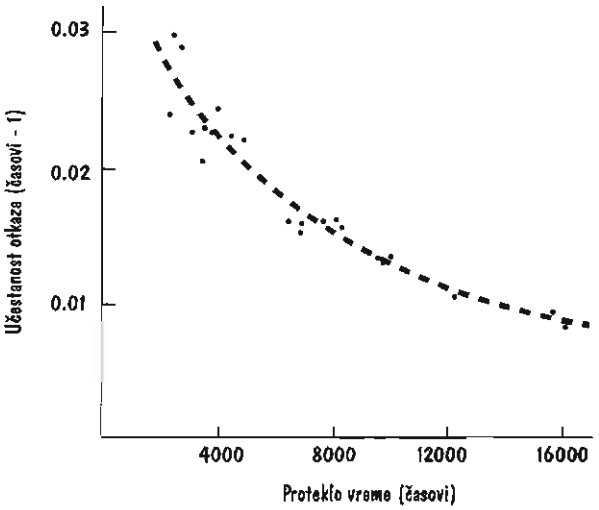
SLIKA 9.13 Izveštaj o odstupanjima iz razvoja jednog sistema za kontrolu saobraćaja (Pfleeger i Hatton 1997)

9.9 TESTIRANJE SISTEMA KOD KOJIH JE BEZBEDNOST KRITIČNA OSOBINA

U poglavlju 1 smo videli nekoliko primera sistema čije otkazivanje može da dovede do povređivanja ili do smrti ljudi. Za takve sisteme kažemo da je kod njih **bezbednost kritična**, pošto su posledice njihovog otkaza toliko ozbiljne. Anthes (1997) je izvestio o mnogim drugim slučajevima kada je otkazivanje softvera dovelo do neprihvatljive štete. Na primer, od 1986. do 1996. primljeno je 450 izveštaja Uprave za hranu i lekove SAD, sa opisima softverskih grešaka u medicinskoj opremi. A 24 od tih izveštaja se odnosilo na softver koji je doveo do smrti ili do povređivanja. Neki od problema bili su:

- Pumpa za intravensku primenu lekova se ispraznila i pacijentu ubrizgala vazduh .
- Monitor nije uključio alarm kada je pacijentovo srce prestalo da kuca.
- Respirator je generisao „nepredviđene udisaje".
- Na digitalnom displeju je došlo do kombinovanja imena jednog pacijenta sa zdravstvenim podacima drugog.

Ovi problemi nisu nužno pokazatelj opadanja kvaliteta softvera. Oni pre odražavaju povećanje količine softvera u sklopu sistema kod kojih je bezbednost kritična osobina.



SLIKA 9.14 Procene učestanosti otkaza na osnovu skupa podataka Musa (Fenton i Pfleeger 1997)

Nažalost, nije uvek jasno na koji način proces razvoja softvera utiče na karakteristike izgrađenog proizvoda, tako da je teško garantovati dovoljnu bezbednost u sistemima kod kojih je ona suštinsko svojstvo. Posebno ne znamo stepen doprinosa pouzdanosti proizvoda svakog primenjenog postupka ili tehnike. U isto vreme, kupci zahtevaju da se postignu nivoi ultra visoke pouzdanosti kakvi ranije nisu postignuti.

Na primer, Airbus 320 je automatizovana letelica, što znači da softver kontroliše većinu njenih vitalnih funkcija. Pošto se otkazivanje automatizovanog softvera u avionu ne može tolerisati, zahtev u vezi sa pouzdanošću sistema formulisan je tako da učestanost otkaza bude 10^{-9} na sat (Rouquet i Traverse 1986). Zahtevi prema softveru moraju biti još strožiji.

Propisana učestanost otkaza znači da sistem može da toleriše najviše jedan otkaz na svakih 10^9 sati. Drugim rečima, sistem sme da otkaže najviše jednom u preko 100.000 godina rada. Kažemo da sistem ima **ultra visoku pouzdanost** kada ima najviše jedan otkaz u 10^9 časova. Jasno je da na ovakve slučajeve ne možemo primeniti uobičajene tehnike za procenu pouzdanosti jer bi to značilo da bi sistem trebalo da radi (najmanje) 100.000 godina, a da mi pri tome pratimo pojave otkaza. Prema tome, moramo da nađemo druge postupke koji će nam pomoći da se uverimo u pouzdanost softvera.

Na slici 9.14 prikazan je drugi način za posmatranje problema ultra visoke pouzdanosti. Napravili smo grafički prikaz sa podacima o otkazima sistema koji se koristi; za ispravljanje uzroka otkaza implementirane su promene u dizajnu softvera i hardvera. Na grafikonu je primenjen model Littlewood-Verrall za izračunavanje učestanosti otkaza (*Rate of Occurrence of Failures*, ROCOF) u različitim trenucima tokom testiranja. Isprekidana linija koja je ručno ubačena ukazuje na naizgled jasnu zakonitost sve manjeg pojavljivanja. Pošto se nagib linije izravnava, do otkaza dolazi sve ređe. To jest, moramo sve duže da testiramo da bi sistem ponovo otkazao, pa prema tome pouzdanost raste.

Međutim, uopšte nije jasno kolika će biti konačna pouzdanost. Nismo u stanju da procenimo da li se kriva asimptotski približava nultoj ili nenultoj verovatnoći zbog potencijalnog indukovanja novih grešaka prilikom ispravljanja starih. Čak i da smo uvereni

da je moguće dostići ultra visok stepen pouzdanosti, ipak bismo morali da testiramo izuzetno dugo da bismo takvo uverenje i dokazali.

U stvari, čak i ako dugo testiramo program i pri tom ne dođe do otkaza, još uvek nemamo potreban stepen garancije. Littlewood je pokazao da ako je program radio x časova bez otkaza, verovatnoća je 50:50 da će i sledećih x časova takođe raditi bez otkaza. Da bi se dostigao nivo garancije kakav je bio potreban za letelicu A320, bilo bi potrebno da softver funkcioniše bez otkaza nekoliko milijardi časova (Littlewood 1991). Prema tome, čak i da je sistem zaista dostigao traženu pouzdanost, mi ne bismo uspeli da se u to uverimo u prihvatljivom vremenskom periodu.

Obezbeđivanje visokih nivoa pouzdanosti je težak, ali i najvažniji izazov koji se mora prihvatiti ako želimo da i dalje koristimo softver u sistemima u kojima je bezbednost osobina od presudnog značaja. Mnogi softverski inženjeri predlažu upotrebu formalnih tehnika provere zahteva, dizajna i kôda. Međutim, formalno procenjivanje prirodnog jezika nije moguće, a značajne informacije mogu da se izgube prilikom prevođenja prirodnog jezika u matematičke simbole. Čak i formalno dokazivanje specifikacije i dizajna nije sasvim pouzdano zato što se ponekad učine greške pri dokazivanju. Zato se kompanije kao što je Baltimore Gas and Electric (Podsetnik 9.9) oslanjaju na kombinaciju različitih tehnika za proveru kvaliteta da bi se pronašlo ili sprečilo što je moguće više grešaka. U podsetniku 9.10 navedeni su još neki pristupi kojima se u ovoj delatnosti postiže veća bezbednost. U isto vreme, istraživači pokušavaju da nađu još neke metode koji će učesnicima u razvoju pomoći da utvrde i obezbede pouzdanost. Razmotrićemo tri takve tehnike: više projekata, softverske bezbednosne slučajeve i čistu sobu.

PODSETNIK 9.9 PROVERA KVALITETA SOFTVERA U BALTIMORSKOJ KOMPANIJI GAS AND ELECTRIC

U državi Maryland, baltimorska kompanija Gas and Electric (BG&E) ne koristi nikakve posebne alate ili tehnike za razvoj softvera kada razvija softver koji služi za kontrolu njihova dva nuklearna reaktora čija bezbednost ima izuzetno veliki značaj. Naprotiv, rukovodioci smatraju da će postići veliku pouzdanost ako temeljito proveravaju definicije zahteva, izvode kvalitetne preglede, pažljivo testiraju, sve to u potpunosti dokumentuju i temeljito upravljaju konfiguracijom.

Da bi se obezbedilo rano otkrivanje svih problema, sistem se dva puta pregleda. Dizajn, kôd i testove sistema pregledaju grupa za informacione sisteme i inženjerska grupa za nuklearni dizajn.

Vlada SAD je objavila „savezna pravila za kriterijume kvaliteta za nuklearne elektrane" i softver koji izrađuje grupa za informacione sisteme u BG&E mora da se povinuje tim propisima. Štaviše, kada proizvođač isporuči softver koji će se koristiti u sklopu kontrolnog sistema, BG&E šalje tim revizora kod proizvođača da bi se uverili da li on ima program za obezbeđenje kvaliteta softvera u skladu sa propisima (Anthes 1997).

PODSETNIK 9.10 PREDLOZI ZA IZGRADNJU SOFTVERA U KOJEM JE BEZBEDNOST KRITIČNA

Anthes (1997) iznosi nekoliko koraka za izgradnju i testiranje sistema u kojima je bezbednost kritična, prema predlogu industrijskih konsultanata:

- Treba biti svestan da testiranjem ne mogu da se uklone sve greške i rizici.
- Ne treba brkati bezbednost, pouzdanost i sigurnost. Sistem koji je 100% pouzdan može i dalje da ne bude ni siguran ni bezbedan.
- Treba uspostaviti tesnu vezu između softvera i bezbednosnih organizacija.
- Treba izgraditi i koristiti bezbednosni informacioni sistem.
- Treba postepeno uvoditi upravljačku kulturu bezbednosti.
- Treba poći od pretpostavke da će korisnici napraviti svaku moguću grešku.
- Nemojte polaziti od pretpostavke da neće nastupiti događaji čija je verovatnoća mala, a uticaj velik.
- Istaknite značaj definicije zahteva, testiranja, pregleda koda i specifikacija, kao i kontrole konfiguracije.
- Nemojte dozvoliti da kratkoročni troškovi budu važniji od dugoročnih rizika i troškova.

Više projekata

Rad na nekoliko projekata, kao što je opisano u poglavlju 5, zasniva se na jednostavnoj filozofiji. Isti sistem se gradi prema istoj specifikaciji zahteva, ali na nekoliko nezavisnih načina, svaki sa vlastitim dizajnom. Svaki sistem se izvršava paralelno sa ostalima, a glasanjem se odlučuje šta da se radi kada se rezultati jednog sistema razlikuju od ostalih. Osnovna pretpostavka je u maloj verovatnoći da će bar tri od pet grupa učesnika u razvoju napisati nepravilan softver za konkretan zahtev, pa se povećava verovatnoća pouzdanosti (Avizienis i Kelly 1984). Ovom tehnikom je izgrađeno nekoliko sistema uključujući i softver za američki svemirski šatl i za Airbus A320 (Rouquet i Traverse 1986). Međutim, postoji empirijska evidencija koja nagoveštava da nezavisno razvijene softverske verzije neće otkazivati nezavisno jedna od druge; više projekata ne obezbeđuje uvek veću pouzdanost u poređenju sa jednim.

Na primer, Knight i Leveson (1986) su sproveli eksperiment u kojem je nezavisno razvijeno 27 verzija jednog softverskog sistema. Oni su analizirali greške otkrivene u svim sistemima i naišli na veliko ponavljanje istih grešaka. Knight i Leveson pretpostavljaju da, stoga što se projektantima softvera tokom školovanja preporučuje korišćenje opštih tehnika i pristupa projektovanju, možemo očekivati da različiti projektanti i programeri prave iste greške. Eckhardt i Lee (1985) opisuju teoretski scenario, zasnovan na pojmu variranja složenosti ulaznih dejstava, koji potvrđuje ove empirijske nalaze.

Miller (1986) ističe da čak i kad gradimo redundantne sisteme koji nezavisno otkazuju, moramo ipak pokušati da procenimo zavisnost između svih parova verzija. On pokazuje da je to dokazivanje isto toliko težak problem kao da smo dobijene sisteme testirali po modelu crne kutije i tvrdi da je to u suštini nemoguć zadatak.

Softverski bezbednosni slučajevi

Testiranje je neminovno povezano sa dizajnom i implementacijom softvera. Možemo da ispitamo dizajn da bismo lakše definisali slučajeve za testiranje i utvrdili da li smo uzeli u obzir svaki mogući scenario. Fenelon i dr. (1994) predlažu razmatranje kvaliteta sistema u kojima je bezbednost kritična osobina navođenjem svih sistemskih ciljeva, analizom kako dizajn zadovoljava te ciljeve i napokon proverom da li implementacija odgovara dizajnu. Sve u svemu, želimo da sistem bude **bezbedan**, to jest, da ne bude nesreća ili gubitaka. Bezbednosne ciljeve možemo da raščlanimo i svakoj komponenti dizajna dodelimo ograničenja ili učestanost otkaza tako da zadovoljavanje svakog cilja na nižem nivou u zbiru omogući ispunjavanje ciljeva bezbednosti sistema. Na taj način pripremamo **bezbednosni slučaj** sistema, eksplicitno izlažući načine na koje softver treba da zadovolji ciljeve performanse sistema u kojima je bezbednost kritična.

Sistem može da se analizira iz četiri različite perspektive: kada je poznat uzrok, kada je uzrok nepoznat, kada su posledice poznate, kada su posledice nepoznate. U svakom slučaju, želimo da uspostavimo veze između situacija koje dovode do normalnog ponašanja i onih koje dovode do potencijalnog otkaza. U tabeli 9.6 prikazani su koraci koje možemo da preduzmemo u svakom od navedenih slučajeva. Kada se koriste tokom dizajna, ove analize nam pomažu da planiramo kako da izbegnemo otkaze; kada se koriste tokom testiranja, one nam pomažu da pronađemo značajne slučajeve za testiranje.

U poglavlju 5 smo videli kako se pomoću analize stabla grešaka ispituju moguće posledice i pronalazi najverovatniji osnovni uzrok. Ovu analizu dopunjava analiza režima otkaza i posledica (***Failure Modes and Effects Analysis***, **FMEA**) tako što polazi od poznatog režima otkaza i pronalazi vezu sa nepoznatim posledicama u sistemu. Kažemo da je **rizik** stanje sistema koje u određenim uslovima dovodi do nesreće. **Režim otkaza** je situacija koja može da dovede do rizika. Na primer, sistemsko prekoračenje SRI-ja iz Ariane-4 jeste rizik; on nije doveo do otkaza Ariane-4 zato što nisu nastupili odgovarajući uslovi (ali su nastupili u Ariani-5). Režim otkaza za Arianu-5 je situacija u kojoj je SRI radio duže od vremena za koje je bio projektovan.

FMEA zahteva veliki napor i zasnovan je na iskustvu analitičara. On se obično primenjuje tako što se izvrši početna analiza dizajna softvera i pri tom izdvajaju režimi koji bi mogli da dovedu do otkaza. Zatim se ispituje na koji način kombinacijom osnovnih režima otkaza može doći do stvarnih otkaza. U podsetniku 9.11 opisano je kako je FMEA u kombinaciji sa drugim tehnikama mogao da unapredi bezbednost mašine Therac-25.

Tabela 9.6 Perspektive za analizu bezbednosti

	Poznat uzrok	Nepoznat uzrok
Poznata posledica	Opis ponašanja sistema	Deduktivna analiza uključujući analizu sa stablom grešaka
Nepoznata posledica	Induktivna analiza, uključujući režime otkaza i analizu posledica	Istraživačka analiza uključujući studije rizika i operativnog rada

PODSETNIK 9.11 BEZBEDNOST I THERAC-25

Od juna 1985. do januara 1987. jedna mašina Therac-25 za terapiju zračenjem bila je pomenuta u šest poznatih nesreća koje su dovele do smrti i ozbiljnih povreda zbog prevelike doze zračenja. Leveson i Turner (1993) veoma detaljno opisuju samu mašinu, nesrećne slučajeve i softverska pitanja, a njihov članak bi trebalo da bude obavezna lektira za sistemske i softverske inženjere koji projektuju i grade sisteme u kojima je bezbednost kritična.

Softver je pisala samo jedna osoba na asemblerskom jeziku PDP-11 i pri tom upotrebila postojeći kod iz jedne prethodne mašine po imenu Therac-6. Deo softvera je testiran na simulatoru, ali veći deo je testiran u sklopu jednog šireg sistema, testiranje je pretežno obavljano na integrisanom sistemu (tj. jedinično testiranje i testiranje softvera bilo je minimalno).

U kompaniji AECL (*Atomic Energy of Canada Limited*) izvršena je analiza bezbednosti sistema Therac-25. AECL je počeo sa analizom režima otkaza i posledica radi utvrđivanja pojedinačnih otkaza koji dovode do značajnih rizika. Nakon toga, da bi se utvrdila višestrukost otkaza i odredili kvantifikatori rezultata, izvršena je analiza sa stablom grešaka. Konačno, angažovan je spoljni konsultant da bi izvršio detaljnu inspekciju koda za softverske funkcije koje su se odnosile na najozbiljnije rizike skeniranje elektronskim snopom, izbor energije zračenja, prekidanje snopa i kalibracija doze. Konačni izveštaj AECL-a preporučio je deset izmena u hardveru Therac-25, među kojima su bile i blokade kao podrška softverski kontrolisanom biranju energije i skeniranju elektronskim snopom.

Leveson i Turner (1993) opisuju kako je osnovni uzrok problema bila jedna greška u merenju vremena koja je bilo teško ponoviti. Oni ističu da većina nesreća vezanih za računare potiče od grešaka u zahtevima, a ne od grešaka prilikom kodiranja i navode nekoliko osnovnih principa softverskog inženjerstva koji su bili prekršeni u sistemu Therac-25:

- Dokumentaciju treba praviti u toku razvoja softvera, a ne kasnije.
- Treba upražnjavati kontrolne metode kojima se obezbeđuje kvalitet softvera i to kao sastavni deo procesa razvoja. To uključuje rano postavljanje standarda koji se koriste za procenu poluproizvoda.
- Jednostavan dizajn se, u poređenju sa složenim, lakše shvata, kodira i testira.
- Softver treba projektovati tako da se predvide otkazi i prikupljaju informacije o njima.
- Nije dovoljno pretpostaviti da će testiranje sistema otkriti softverske probleme. Softver bi trebalo temeljito testirati i podvrći formalnoj analizi još u fazi komponente i na nivou sistema, pre integracije sa hardverom.

Studije rizika i upotrebljivosti (*Hazard and operability studies*, HAZOPS) uključuju strukturnu analizu radi predviđanja rizičnosti sistema i predlaganja načina da se oni izbegnu ili da se na njih reaguje. Te studije se zasnivaju na tehnici razvijenoj u Imperial Chemical Industries (UK) 60-ih godina sa ciljem da se analizira dizajn jedne nove hemijske fabrike. U okviru opsežnog pregleda, po sistemu HAZOPS koriste se karakteristične reči u kombinaciji sa analizom toka kontrole i podataka između obrađivanih komponenata, da bi se analitičarima olakšalo prepoznavanje rizika. U tabeli 9.7 prikazani su primeri karakterističnih reči za jedan sistem gde je vremenski raspored događaja, kontrolisan podacima i signalima, važan za koordinaciju zadataka.

Tabela 9.7 HAZOP karakteristične reči

Karakteristična reč	Značenje
Nema	Nije poslat niti primljen nikakav podatak niti kontrolni signal.
Previše	Količina podataka je preterana ili prebrzo stižu.
Nedovoljno	Količina podataka je nedovoljna ili presporo stiže.
Delimično	Podaci ili kontrolni signal su nepotpuni.
Dodatno	Podaci ili kontrolni signal imaju dodatnu komponentu.
Rano	Signal stiže prerano prema sistemskom satu.
Kasno	Signal stiže prekasno prema sistemskom satu.
Pre	Signal stiže pre očekivanog roka.
Kasnije	Signal stiže posle očekivanog roka.

Fenelon i dr. (1994) prilagodili su HAZOPS za softverske situacije i metod koji su razvili nazivali su SHARD. Oni zasnivaju svoje karakteristične reči na tri aspekta svakog rizika:

1. *Isporuka:* Softver vrši uslugu kada ne bi trebalo ili ne vrši uslugu kad bi trebalo: nečinjenje ili činjenje.
2. *Vreme:* Usluga je izvršena ili previše rano ili previše kasno: rano ili kasno.
3. *Vrednost:* Usluga je neispravna, a greška se ili lako vidi ili ne: gruba nepravilnost ili fina nepravilnost.

Ovaj opšti okvir proširuje se na veći skup karakterističnih reči, kao što je prikazano u tabeli 9.8.

Pošto se utvrdi režim otkaza, traže se mogući uzroci i posledice. Kada pronađemo razumni uzrok i posledicu, nakon toga tražimo strategije: bilo da bi se izbegao uzrok bilo da bi se umanjile posledice. Tokom testiranja možemo da biramo slučajeve za testiranje za svaki režim otkaza da bismo mogli da utvrdimo da li sistem reaguje na odgovarajući način (tj. da li dolazi do katastrofalnog otkaza).

Tabela 9.8 SHARD karakteristične reči

Tok		Činjenje		Vremenska kategorizacija otkaza		Vrednost	
Protokol	Tip	Nečinjenje	Činjenje	Rano	Kasno	Fina	Gruba
Bazen	Logički	Nije ažuriran	Neželjeno ažuriranje	-	Stari podaci	Zastoj pri...	-
	Numerički	Nije ažuriran	Neželjeno ažuriranje	-	Stari podaci	Pogrešna tolerancija	Van tolerancije
	Kompleksan	Nije ažuriran	Neželjeno ažuriranje	-	Stari podaci	Neispravna	Nekonsi-stentna
Kanal	Logički	Nema podatka	Dodatni podaci	Rano	Kasno	Zastoj pri...	-
	Numerički	Nema podatka	Dodatni podaci	Rano	Kasno	Pogrešna tolerancija	Van tolerancije
	Kompleksan	Nema podatka	Dodatni podaci	Rano	Kasno	Neispravna	Nekonsi-stentna

Čista soba

Sredinom 80-ih su istraživači u IBM-u predložili novi proces razvoja softvera projektovan sa ciljem proizvodnje softvera visokog kvaliteta u visoko produktivnom timu. Taj proces, po imenu **„čista soba"**, reflektuje ideje koje su se primenjivale u proizvodnji integrisanih kola i služile za postizanje minimalnog broja grešaka (Mills, Dyer i Linger 1987).

Principi i tehnike čiste sobe. Pristup čiste sobe sadrži dva temeljna principa:

1. da se softver proveri prema specifikacijama, a ne da se čeka pojava greške pri jediničnom testiranju;
2. da se proizvede softver bez grešaka ili gotovo bez grešaka.

Ti principi se primenjuju kombinacijom nekoliko tehnika opisanih u ovom poglavlju i u prethodnim poglavljima. Prvo, specifikacija softvera se daje po principu kutija, što je opisano u poglavlju 5. Sistem se definiše kao crna kutija, usavršava se kao kutija stanja i još jednom usavršava kao providna kutija. Struktura sa kutijama podstiče analitičare da rano uoče nedostatke u zahtevima jer ih je tada lakše i jeftinije otkloniti.

Zatim se specifikacija providne kutije konvertuje u nameravanu funkciju, izraženu na prirodnom jeziku ili pomoću matematičkih izraza, zavisno od potrebe. Teorema ispravnosti definiše relaciju, iskazanu kao jedan od tri uslova ispravnosti, koji opisuje ispravnost svake nameravane funkcije u odnosu na njene kontrolne strukture.

Na primer, uslovi ispravnosti za uobičajene strukture mogu se predstaviti u formi pitanja (Linger - nepoznat datum):

```
Kontrolne strukture:          Uslov ispravnosti:
Sequence                      Za sve argumente:
   [f]
   DO
     g:                       Da li g, a zatim h vrši f?
     h
   OD
Ifthenelse
   [f]
   IF p                       Kad god je p tačno
   THEN                         radi g vršeći f, i
      g                       kad god je p netačno
   ELSE                         radi h vršeći f?
     h
   FI
Whiledo
   [f]                        Da li je završetak garantovan, i
   WHILE p                    kad god je p tačno
   DO                           radi g vršeći f, i
    g                         kad god je p netačno
   OD                           ne radi ništa vršeći f?
```

Projektni tim pregleda ove relacije i proverava ispravnost stanja formalnim proveravanjem ispravnosti. Na primer, program i njegovo proveravanje moglo bi da izgleda ovako (Linger - nepoznat datum):

```
Program:                    Provere:
[f1]                        f1 = [DO g1;g2;[f2] OD] ?
DO
   g1
   g2
   [f2]                     f2 = [WHILE p1 DO [f3] OD] ?
   WHILE
    p1
    DO [f3]                 f3 = [DO g3;[f4];g8 OD]?
     g3
    [f4]                    f4 = [IF p2 THEN [f5] ELSE [f6] FI]?
    IF
       p2
    THEN [f5]               f5 = [DO g4;g5 OD] ?
       g4
       g5
    ELSE [f6]               f6 = [DO g6;g7 OD] ?
       g6
       g7
    FI
    g8
   OD
OD
```

Ova provera zamenjuje jedinično testiranje, koje nije dozvoljeno. U ovoj fazi, softver se proverava u odnosu na specifikaciju.

Završni korak uključuje statističko testiranje, pri čemu se slučajevi biraju na osnovu verovatnoće korišćenja, kao što smo videli ranije u ovom poglavlju. Rezultati se koriste u modelu kvaliteta da bi se odredilo očekivano prosečno vreme do otkaza i druge mere kvaliteta. Istraživači u IBM-u smatraju da se pri uobičajenom testiranju prekrivenosti greške pronalaze slučajnim redosledom, dok je statističko testiranje efikasnije i unapređuje sveukupnu pouzdanost softvera. Cobb i Mills (1990) izveštavaju da je statističko testiranje više nego 20 puta efikasnije u produžavanju MTTF-a od testiranja prekrivanja.

Očekivanja od „čiste sobe" „Čista soba" je više puta empirijski procenjivana. Na primer, Linger i Spangler (1992) napominju da su prvi timovi koji su primenjivali ovu metodu u IBM-u i drugim kompanijama proizveli preko 300.000 redova koda uz visoku produktivnost i uz učestanost grešaka od 2,9 na hiljadu redova koda. Oni tvrde da to predstavlja smanjenje za red veličine u odnosu na ranijih 30 do 50 grešaka na hiljadu redova koda koliko je bilo uobičajeno kod tradicionalnog razvoja. Štaviše: „...iskustvo pokazuje da su greške zaostale iza provere ispravnosti obično proste omaške koje se lako pronalaze i ispravljaju statističkim testiranjem, a ne teške greške u dizajnu i interfejsu na koje se često nailazi kod tradicionalnog razvoja" (Linger i Spangler 1992). Objavljeni rezultati odnose se na timove od 3 do 50 članova, sa različitim vrstama aplikacija koje su razvijane na različitim proceduralnim i objektno orijentisanim programskim jezicima.

Tabela 9.9 Rezultati Nasine studije SEL (Basili i Green 1994) © 1996 IEEE

Karakteristika	Eksperiment	Studijski primer 1	Studijski primer 2	Studijski primer 3
Veličina tima	Razvojni timovi od po tri člana (10 eksperimentalnih i 5 kontrolnih); ista nezavisna grupa za testiranje	Razvojni tim od 3 člana; tim za testiranje od 2 člana	Razvojni tim od 4 člana; tim za testiranje od 2 člana	Razvojni tim od 14 članova; tim za testiranje od 4 člana
Veličina projekta i aplikacija	1500 redova fortranskog koda; sistem razmene poruka za diplomski laboratorijski kurs	40.000 redova fortranskog koda; sistem za zemaljsku podršku dinamike leta	22.000 redova fortranskog koda; sistem za zemaljsku podršku dinamike leta	160.000 redova fortranskog koda; sistem za zemaljsku podršku dinamike leta
Rezultati	Timovi u čistim sobama koristili su manje računarskih resursa, uspešnije ispunjavali zahteve i ostvarili veći procenat pravovremenih isporuka.	U projektu je utrošen veći rad na dizajnu, manje računarskih resursa, a ostvarena bolja produktivnost i pouzdanost od one koja je bila osnovna za to okruženje.	Na projektu je nastavljen trend bolje pouzdanosti uz očuvanu osnovnu produktivnost.	Pouzdanost projekta samo neznatno bolja od osnovne, dok je produktivnost pala ispod osnovne produktivnosti.

U laboratoriji za softversko inženjerstvo (*Software Engineering Laboratory*, SEL) Nasinog centra Goddard Space Flight izvršeno je rigorozno testiranje u čistoj sobi. Izvršen je niz kontrolisanih eksperimenata i studijskih primera da bi se utvrdilo do koje mere ključni elementi pristupa ispunjavaju ono što je obećano. Kao što vidite iz rezultata u tabeli 9.9, ovaj pristup daje dobre rezultate na malim, ali ne i na velikim projektima. Zbog toga u SEL-u i dalje razvijaju model čiste sobe; konkretno, trenutni SEL model se primenjuje na projekte ispod 50.000 redova koda, ali se menja za veće projekte. Osim toga, učesnici u razvoju u SEL-u više ne koriste modelovanje pouzdanosti i predviđanje jer nemaju dovoljno podataka na kojima bi zasnivali projekcije. Basili i Green (1994) napominju da su ih ove studije izvršene u okruženju dinamike leta ubedile da ključni principi čiste sobe daju manju učestanost grešaka, veću produktivnost, potpuniji i konsistentniji skup komentara u kodu, kao i drugačiju distribuciju rada učesnika u razvoju. Međutim, oni upozoravaju da se SEL okruženje razlikuje od IBM-ovog i da se čista soba mora prilagoditi okruženju u kom se koristi.

Oprezno sa čistom sobom". Mada se u velikom delu literature tvrdi da čista soba poboljšava kvalitet softvera, Beizer (1997) tvrdi da te rezultate treba tumačiti s rezervom. On tvrdi da se preskakanjem jediničnog testiranja u čistoj sobi uvodi opasno loša praksa, protivno „poznatoj teoriji testiranja i zdravom razumu". Kako kaže Beizer: „Ne možete da pronađete grešku ako ne izvršite kôd u kojem se greška nalazi" , a pravoverna čista soba se za proveru pouzdanosti oslanja samo na statističko testiranje i potpuno isključuje svako jedinično testiranje. Štaviše, kao što je objašnjeno u podsetniku 9.12, može se doći u zabludu ako se vrši samo statističko testiranje.

PODSETNIK 9.12 KADA STATISTIČKO TESTIRANJE MOŽE DA DOVEDE U ZABLUDU

Pri operativnom testiranju se polazi od pretpostavke da se greške najčešće manifestuju u operacijama koje se najčešće koriste i to sa najčešće korišćenim ulaznim vrednostima. Kitchenham i Linkman (1997) ističu da je ova pretpostavka tačna za konkretnu operaciju, ali ne za ceo skup operacija u sistemu. Da bi to objasnili, oni opisuju primer gde operacija šalje zahteve za štampanje datoteka na jedan od četiri štampača. Kada zahtev stigne, moguće je da ne budu dostupni svi štampači. Postoje tri mogućnosti:

1. Štampač je dostupan i ne postoje interni redovi za čekanje na štampu. Ovo stanje se naziva *nezasićenim* stanjem.
2. Nijedan štampač nije dostupan i ne postoji red za čekanje na štampu; potrebno je da se inicijalizuje interni red i da se zahtev stavi u taj red. Ovo stanje naziva se *tranzicionim* stanjem.
3. Nijedan štampač nije dostupan, a red za štampu već postoji, pa se zahtev za štampanje stavlja u taj red. Ovo stanje se naziva *zasićenim* stanjem.

Iz iskustva možda znamo da se na zasićeno stanje nailazi u 79% slučajeva, na nezasićeno u 20% slučajeva, a na tranziciono u 1% slučajeva. Pretpostavimo da je verovatnoća da dođe do otkaza jednaka za svako od ta tri stanja i da iznosi 0,001. Tada će doprinos svakog stanja sveukupnoj verovatnoći da dođe do otkaza biti (0,001) * (0,20) odnosno 0,0002 za nezasićeno stanje, (0,001) * (0,79) odnosno 0,00079 za zasićeno stanje i (0,001) * (0,01) odnosno 0,00001 za tranziciono stanje. Pretpostavimo da imamo tri greške, i da je po jedna pridružena svakom stanju. Kitchenham i Linkman (1997) pokazuju da bismo za 50% šanse otkrivanja svake greške morali da izvršimo 0,5/0,0002 = 2.500 slučajeva za testiranje kako bismo otkrili grešku u nezasićenom stanju, 0,5/0,00001 = 500.000 slučajeva kako bismo otkrili grešku u tranzicionom stanju i 0,5/0,00079 = 633 slučajeva kako bismo otkrili grešku u zasićenom stanju. Prema tome, testiranje prema operativnom profilu će otkriti veću grešaka.

Međutim, oni napominju da su tranzicione situacije često najsloženije i najviše podložne greškama. Na primer, tako poletanje i sletanje predstavljaju mali procenat operativnog profila aviona, ti režimi rada su odgovorni za veliki procenat sveukupnih otkaza. Prema tome, pretpostavimo da je verovatnoća da se izabere ulazno stanje koje će dovesti do greške jednaka 0,001 za zasićeno i nezasićeno stanje, ali 0,1 za tranziciono stanje. Tada će doprinos svakog režima u sveukupnoj verovatnoći otkaza biti (0,001) * (0,20) odnosno 0,0002 za nezasićeno stanje, (0,001) * (0,79) odnosno 0,00079 za zasićeno stanje i (0,1) * (0,01) odnosno 0,001 za tranziciono stanje. Kada se preračuna u test slučajeve kao malo pre, potrebno nam je 2.500 test slučajeva da bismo otkrili grešku u nezasićenom stanju, 633 u zasićenom stanju, ali samo 500 za otkrivanje tranzicione greške. Drugim rečima, operativni profil bi usmerio testiranje na zasićeni režim, a u stvari bi trebalo da se usredsredimo na tranzicione greške.

Beizer ističe da se čista soba nikada ne poredi sa:

- pravilnim jediničnim testiranjem usmerenim na ciljeve pokrivanja;
- testiranjem koje izvršavaju softverski inženjeri obučeni za tehnike testiranja;
- testiranjem u organizacijama koje koriste dizajn testiranja i tehnike za automatsko testiranje;
- pravilnim integracionim testiranjem.

Štaviše, čista soba polazi od pretpostavke da imamo iskustva u merenju pouzdanosti softvera. Međutim, u literaturi o pouzdanosti čiju rekapitulaciju možete naći u priručniku o pouzdanosti autora Lyu (1996), ukazuje se na to da postoje mnogi problemi sa tehnikama pouzdanosti. Konkretno, kao što smo videli, ne može se garantovati da će operativni profili, koji su bitni za dobro modelovanje, biti precizni, pa čak ni da će biti relevantni.

Beizer opisuje rezultate 24 empirijske studije u kojima se procenjivala čista soba, uključujući studiju Basili i Green (1994), i ističe da je postojalo nekoliko fatalnih nedostataka.

- Učesnici su znali da je reč o eksperimentu, pa je efekat Hawthornea mogao da utiče na rezultate. To jest, na povećanje kvaliteta je mogla da utiče i činjenica da su učesnici znali da će se procenjivati njihovi proizvodi, a ne sama tehnika „čiste sobe".
- „Kontrolna" grupa koja je vršila testiranje nije imala nikakvu obuku niti iskustvo sa metodama pravilnog testiranja.
- Za podršku testiranju nisu korišćeni nikakvi alati za proveru pokrivenosti, niti tehnike za automatizaciju.
- Varanje nije kontrolisano, pa je moguće da je tehnika čiste sobe u stvari primenjena na već očišćen kôd.

Beizer, naprotiv, primećuje da nijedno istraživanje nije proglasilo postojeću teoriju testiranja kao matematički neispravnu. Pošto nalazi da je većina empirijskih dokaza neubedljiva, on smatra da bi retrospektivna empirijska analiza mogla da eliminiše neka predubeđenja (Vinter 1996). Da bi se poredila dva metoda, možemo da razvijemo softver prvim metodom beležeći pri tom sve greške otkrivene tokom razvoja i prve godine korišćenja. Zatim bi se drugi metod primenio na kôd da bi se videlo da li će on otkriti neku već otkrivenu grešku. Slično tome, na sistem razvijen po drugom metodu mogao bi se retroaktivno primeniti prvi.

Još uvek se ne zna pobednik ove rasprave. Ali Beizer pokreće neka značajna pitanja i upozorava nas da budemo skeptični ne samo prema tehnikama softverskog inženjerstva za koje se tvrdi da rešavaju glavne probleme vezane za kvalitet i produktivnost, već i prema ocenjivačkim tehnikama koje se koriste kada neko hoće da nas ubedi kako je jedna tehnika bolja od druge. Moramo veoma pažljivo da testiramo ne samo softver već i svoje teorije i pretpostavke.

9.10 PRIMER IZ INFORMACIONIH SISTEMA

Koncepti opisani u ovom poglavlju imaju praktičan značaj za učesnike u razvoju sistema Piccadilly. Tim za testiranje mora da izabere metod za odlučivanje kako da izvrši testiranje sistema, kada da prekine testiranje i koliko grešaka i otkaza može da očekuje. Na

ta pitanja nije lako odgovoriti. Na primer, iz literature nije jasno koja gustina grešaka se očekuje, a koja je prihvatljiva:

- Joyce (1989) izveštava da je avionski sistem za svemirski šatl u Nasi imao gustinu grešaka od 0,1 na hiljadu redova koda.
- Jones (1991) tvrdi da vodeće softverske kompanije imaju gustinu grešaka od 0,2 na hiljadu redova koda ili ne više od 0,025 grešaka, koje su prijavili korisnici, po funkcionalnoj tački.
- Musa, Iannino i Okumoto (1990) opisuju anketu pouzdanosti koja je pronašla prosečnu gustinu od 1,4 greške na hiljadu redova koda u kritičnim sistemima.
- Cavano i LaMonica (1987) pominju ankete vojnih sistema u kojima je nađena gustina grešaka od 5,0 do 55,0 grešaka na hiljadu redova koda.

Prema tome, postavljanje ciljeva za gustinu grešaka i izračunavanje kriterijuma za završetak je u najmanju ruku teško. Kao što se primećuje u podsetniku 9.13, analogije sa hardverom ponekad nisu primerene za donošenje odluka o kvalitetu. Umesto toga, učesnici u razvoju Piccadillyja bi dobro uradili da ispitaju evidenciju o greškama i otkazima iz ranijih projekata koji su na neki način slični: po upotrebljenom jeziku, po funkcijama, po broju članova tima ili po tehnikama dizajna. Mogu se napraviti model ponašanja grešaka i otkaza, procena verovatnih rizika i projekcije na osnovu podataka koji se prikupljaju u fazi testiranja.

Uz funkcije sistema Piccadilly vezane su mnoge promenljive zato što cena vremena za reklame zavisi od toliko različitih karakteristika: dana u nedelji, doba dana, konkurentskih programa, broja i vrste ponavljanja reklame i tako dalje. Prema tome, treba razmotriti više različitih slučajeva za testiranje, pa bi za pravljenje i praćenje tih slučajeva i njihovih rezultata bilo korisno upotrebiti automatizovani alat za testiranje.

Bach (1997) predlaže nekoliko faktora koje treba razmotriti pri izboru alata za testiranje.

- *Kvalitet:* Da li alat poseduje sva potrebna kritična svojstva, pogotovo za proveru valjanosti rezultata i za upravljanje podacima i test skriptovima?
- *Pouzdanost:* Da li alat može da radi dugo bez otkaza?
- *Kapacitet:* Da li alat može bez otkaza da radi u okruženju masovne industrijske proizvodnje?
- *Lakoća učenja:* Da li se alat može savladati za kratko vreme?
- *Lakoća upotrebe:* Da li se alat lako koristi ili su mu funkcije nezgrapne i teške?
- *Performanse:* Da li će vam alat uštedeti vreme i novac prilikom planiranja, razvoja i primenjivanja testa?
- *Kompatibilnost:* Da li alat može da se koristi u vašem okruženju?
- *Nenametljivost:* Da li alat simulira stvarnog korisnika na realan način?

Bach upozorava da se ne oslanjamo samo na opise iz priručnika za korisnike niti na prezentacije na prodajnim manifestacijama. Da bi se procenili svi činioci, važno je da se alat koristi na pravom projektu i posmatra kako funkcioniše u našem okruženju. Učesnici u razvoju na projektu stanice Piccadilly trebalo bi da procene više alata u svom okruženju i izaberu onaj koji ih oslobađa mučnog pravljenja svih mogućih slučajeva za testiranje. Međutim, nijedan alat ne može da ih zameni u ocenjivanju koji su elementi značajni da bi se test slučajevi razlikovali.

PODSETNIK 9.13 ZAŠTO POJAM PRVE KLASE NIJE PRIMENLJIV NA SOFTVER

Kada razmišljamo o sistemima vrhunskog kvaliteta, često po analogiji sa hardverom pomišljamo na primenu uspešnih hardverskih tehnika na softver. Ali Binder (1997) objašnjava zašto neke hardverske tehnike ne mogu da se primene na softver. Konkretno, razmotrite ideju da se gradi softver koji će zadovoljiti takozvane kriterijume „prvoklasnog" kvaliteta. Proizvedeni delovi obično imaju raspon tolerancije unutar kojeg se kaže da su zadovoljili ciljeve dizajna. Na primer, ako deo treba da teži 45 miligrama mogli bismo da prihvatimo delove između 44,9998 miligrama i 45,0002 miligrama. Ako je težina dela izvan tog raspona, reći ćemo da je neispravan. Definicija prvoklasnog kvaliteta kaže da se u milijardu delova dopušta samo 3,4 komada van prihvatljivog raspona. Kako se povećava broj delova u proizvodu, verovatnoća da dobijemo proizvod bez grešaka opada, pa verovatnoća da dobijemo proizvod od sto delova bez ijedne greške (ako su delovi projektovani prema ograničenjima prve klase) iznosi 0,9997. Taj pad kvaliteta možemo da rešimo tako što ćemo smanjiti broj delova, smanjiti broj kritičnih ograničenja za svaki deo ili pojednostaviti proces kojim se delovi sastavljaju.

Međutim, Binder ističe da ova analogija sa hardverom nije primenljiva na softver iz tri razloga: zbog procesa, zbog karakteristika i zbog jedinstvenosti. Prvo, pošto su ljudi različiti, softverski proces neminovno sadrži visok stepen varijacije među „delovima" koji se ne može kontrolisati. Drugo, softver zadovoljava uslove ili ih ne zadovoljava. Ne postoje različiti stepeni zadovoljavanja, kao što bi bili „ne zadovoljava, donekle zadovoljava, prilično zadovoljava, sasvim zadovoljava". Zadovoljavanje je binarno i čak ne može da se poveže sa pojedinačnom greškom; ponekad mnoge greške doprinose samo jednom otkazu, a obično ne znamo tačno koliko grešaka postoji u jednom sistemu. Štaviše, i sam razlog otkaza se možda nalazi u drugoj aplikaciji koja se povezuje sa našom (kao kada spoljašnji sistem šalje sistemu koji se testira pogrešnu poruku). Treće, softver nije rezultat masovne proizvodnje. Nezamislivo je da gradite na hiljade identičnih softverskih komponenti koristeći istovetan razvojni proces, proveravajući usklađenost na malom uzorku i tek zatim pokušavajući da ispravite proces ako iz njega dobijate previše sistema koji ne ispunjavaju zahteve. U mehaničkom procesu možemo da proizvedemo na milione kopija, ali je to irelevantno kada je reč o softverskim greškama. „Kada se koristi kao slogan, prvoklasnost jednostavno znači neki (subjektivan) veoma nizak procenat grešaka. Precizan statistički smisao se gubi." (Binder, 1997.)

9.11 PRIMER IZ PRAKSE

U prethodnim poglavljima smo videli da je na otkaz inercijalnog referentnog softvera SRI na Ariane-5 došlo i zbog problema u zahtevima i neodgovarajućih pregleda. Komisija koja je istraživala otkaz takođe je razmatrala preventivnu ulogu koju je mogla da odigra simulacija. Napomenuli su da bi bilo nemoguće izolovati i testirati SRI tokom leta, ali bi softverske ili hardverske simulacije mogle da generišu signale prema predviđenim

parametrima leta dok bi okretnica obezbeđivala ugaono kretanje. Istražitelji misle da bi se uslovi za otkaz otkrili da se prilikom testova prihvatanja pošlo tim putem.

Osim toga, testiranje i simulacije su se vršile u FSF-u (*Functional Simulation Facility*), s ciljem da se ocene:

- sistemi navođenja, navigacije i kontrole;
- redundantnost senzora;
- funkcije svih faza rakete;
- usklađenost računara na letelici sa sistemom kontrole leta.

U ovoj laboratoriji su inženjeri izvršili simulacije sa zatvorenom petljom za ceo let, uključujući operacije na zemlji, telemetrijski tok i dinamiku lansirne rampe. Smatrali su da su proverili da li je ispravna nominalna putanja, kao i putanje degradirane pomoću internih parametara lansirne rampe, parametara atmosfere i otkaza opreme. Tokom tih testova, nisu korišćeni stvarni SRI-ji; oni su simulirani namenski razvijenim softverom. Sa stvarnim SRI-jom su izvršena samo neka testiranja otvorene petlje i to samo da bi se proverila električna integracija i usklađenost komunikacije.

Istražitelji napominju da

> nije obavezno, čak i kada je poželjno, da svi delovi podsistema budu prisutni u svim testovima u datoj fazi. Ponekad je to fizički nemoguće ili nije moguće u potpunosti ili na reprezentativan način. U tim slučajevima je logično da se oni zamene simulatorima ali samo nakon pažljive provere da li su prethodne faze testiranja potpuno pokrile celo područje (Lions i dr. 1996).

U stvari, u istražnom izveštaju opisana su dva načina da se upotrebe SRI-ji. Prvi način je mogao da obezbedi preciznu simulaciju, ali bi bio veoma skup. Drugi je bio jeftiniji, ali je njegova preciznost zavisila od preciznosti simulacije. Međutim, u oba slučaja bi se veći deo elektronike i sav softver testirao u stvarnom radnom okruženju.

Zašto se SRI-ji nisu koristili u simulaciji zatvorene petlje? Prvo, smatralo se da su SRI-ji provereni na nivou opreme. Drugo, preciznost navigacionog softvera u računaru na letelici zavisila je od merenja SRI-ja. Međutim, ta preciznost ne bi mogla da se postigne probnim signalima; smatralo se da će se pomoću modela postići bolje simuliranje režima otkaza. Na kraju, SRI je radio sa osnovnim ciklusom od jedne milisekunde, ali su u FSF-u koristili šest milisekundi i na taj način još više smanjili preciznost simulacije.

Istražitelji su potvrdili da su ovi razlozi tehnički opravdani. Ali su takođe istakli da testiranja sa simulacijom sistema imaju svrhu ne samo da se provere interfejsi već takođe i sistem kao celina u toj konkretnoj aplikaciji. Zaključak je bio da

> Apsolutno jeste postojao rizik u pretpostavci da je kritičan deo opreme kakav je SRI proveren samostalno ili prethodnim korišćenjem na Ariani-4. Mada je poželjan visok stepen preciznosti, u FSF testovima sistema svakako je bilo bolje načiniti kompromis što se tiče preciznosti, a postići ostale ciljeve, između ostalih da se dokaže pravilna integracija sistema sa opremom kao što je SRI. Preciznost sistema navođenja mogla se efikasno prikazati analizom i računarskom simulacijom (Lions i dr. 1996).

9.12 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju opisana su mnoga važna pitanja u vezi sa testiranjem softvera, uključujući i ona koja se tiču pouzdanosti i bezbednosti. Kao samostalan učesnik u razvoju trebalo bi da testiranje imate u vidu već od samog početka životnog ciklusa sistema. Tokom analize zahteva trebalo bi da razmišljate o sistemskim funkcijama koje će beležiti informacije o stanju i podatke koji će vam pomoći da pronađete osnovni uzrok ako softver bude otkazao. Tokom dizajna bi trebalo da koristite analizu sa stablom grešaka, analizu režima otkaza i posledica i ostale tehnike koje pomažu da se izbegnu otkazi ili da se njihove posledice ublaže. Tokom pregleda dizajna i koda možete da napravite slučaj za proveru bezbednosti da biste sebe i svoje kolege uverili da je vaš softver veoma pouzdan i da će rezultirati bezbednim sistemom. Tokom testiranja, možete veoma pažljivo da razmotrite sve moguće test slučajeve, da koristite automatizaciju gde je to primereno i obezbedite da vaš dizajn uzima u obzir sve moguće opasnosti.

9.13 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Kao samostalan učesnik u projektu možete preduzeti korake potrebne da bi komponente koje projektujete, razvijate i testirate radile u skladu sa specifikacijom. Često problemi koji se jave u fazi testiranja potiču od interfejsa među komponentama. Integraciono testiranje je korisno da bi se proverile kombinacije komponenti, ali testiranje sistema je bliže realnosti - pa je veća verovatnoća da dođe do otkaza. Vaš tim mora da bude spreman na komunikaciju tokom ove vrste testiranja i da eksplicitno iznese sve pretpostavke od kojih se pošlo. Vaš tim mora pažljivo da ispita granične uslove sistema i obradu izuzetaka.

Tehnike kao što je čista soba zahtevaju intenzivan timski rad, planiranje i koordinaciju da bi se razvile strukture kutija i projektovalo i izvršilo statističko testiranje. Aktivnosti u vezi sa testovima prihvatljivosti zahtevaju tesnu saradnju sa kupcima i korisnicima; pošto oni vrše testiranja i pronalaze probleme, vi morate brzo utvrditi uzrok da biste omogućili nastavak testiranja nakon ispravke. Prema tome, dok se razvoj u nekim svojim delovima odvija izolovano i jedinično, testiranje sistema je grupni zadatak.

9.14 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Osim ovde opisanih pristupa testiranju postoje i mnogi drugi. Empirijsko istraživanje i dalje pomaže u procenjivanju koje su vrste testiranja najbolje za otkrivanje kojih vrsta grešaka. Taj empirijski rad, u kombinaciji sa teorijom testiranja, obećava testiranje koje opravdava troškove.

Hamlet (1992) predlaže nekoliko ključnih pitanja koja bi istraživači trebalo da razmotre:

- U testiranju pouzdanosti, prefinjeni načini podele sistema prilikom testiranja ponekad nisu ništa bolji od nasumičnog testiranja.
- Potrebno je bolje definisanje pojma pouzdanosti softvera. Moguće je da bujanje stanja (tj. javljanje veoma velikog broja stanja zbog kojih nastaje veliki broj slučajeva za testiranje) nije toliko opasno kao što se čini. Možda možemo da grupišemo slična stanja i zatim biramo uzorak slučajeva za testiranje iz svake grupe.

- Možda se mogu odrediti karakteristike programa i sistema za koje potreban broj slučajeva za testiranje nije enormno velik; možemo uslovno reći da je te sisteme lakše testirati od onih sa enormno velikim brojem slučajeva.
- Voas i Miller (1995) su definisali tehniku kojom se ispituje osetljivost stanja na promene. Greške u neosetljivim stanjima se prilikom testiranja obično teško otkrivaju. Istraživači moraju da istraže u kakvoj su vezi osetljivost i lakoća s kojom se testiranje obavlja.

Osim toga, istraživači bi trebalo da razlikuju testiranje radi pronalaženja grešaka od testiranja radi povećanja pouzdanosti. Neki istraživači (kao što su Frankl i dr. 1997) prikazali su da težnja prvom cilju neće uvek ostvariti i drugi.

9.15 SEMINARSKI RAD

Objasnili smo da projekti nisu do te mere različiti kao što nam se možda čini. Uporedite svoj projekat sa projektima ostalih studenata ili ostalih timova u vašem odeljenju. Po čemu se razlikuju? Po čemu su slični? U kojoj vrsti testiranja bi te sličnosti ili razlike olakšale, odnosno otežale otkrivanje grešaka u kodu?

9.16 KLJUČNE REFERENCE

Časopis *IEEE Software* je posvetio nekoliko specijalnih izdanja temama kojima smo se bavili u ovom poglavlju. Izdanja za juni 1992. i maj 1995. bila su posvećena pouzdanosti, dok je izdanje za mart 1991. bilo usredsređeno na testiranje.

Na godišnjoj međunarodnoj konferenciji o softverskom inženjerstvu obično se pojave dobri radovi sa novitetima iz teorije testiranja. Na primer, Frankl i dr. (1997) ispituju razliku između testiranja za poboljšanje pouzdanosti i testiranja za pronalaženje grešaka. Dobre referentne knjige za testiranje su Beizer (1990); Kaner, Falk i Nguyen (1993); i Kit (1995). Svaka od njih daje realan uvid zasnovan na iskustvu u ovoj delatnosti.

Nekoliko kompanija se bavi ocenjivanjem alata za testiranje softvera i objavljuje preglede njihovih mogućnosti. Na primer, Ovum Ltd. (info@ovum.mhs.compuserve.com) pravi detaljne analize na dvadesetak stranica za svaki od nekoliko desetina alata za testiranje. Alati se opisuju u smislu njihove primenljivosti na analizu i proveru valjanosti zahteva, planiranje i upravljanje, simulaciju, pripremanje testova, izvršavanje testova, analizu pokrivenosti, analizu izvornog koda i generisanje slučajeva za testiranje.

Pouzdanost softvera i sistemi u kojima je bezbednost važna privlače sve veću pažnju i ima mnogo dobrih članaka i knjiga o tim ključnim pitanjima, uključujući i radove Levesona (1996, 1997). Osim toga, Centar za pouzdane računarske sisteme (*Dependable Computing Systems Centre*) na katedri za računarske nauke univerziteta u Yorku, UK, razvija tehnike i alate za procenu pouzdanosti softvera. Više informacija možete dobiti od njihovog direktora, Johna McDermida, na adresi jam@minster.york.ac.uk.

Testiranje upotrebljivosti je veoma važno; sistem koji je ispravan i pouzdan, ali je težak za upotrebu može u suštini da bude gori od lako upotrebljivog, a nepouzdanog sistema. Testove upotrebljivosti kao i opštija pitanja u vezi sa upotrebljivošću temeljito su obradili Hix i Hartson (1993).

9.17 VEŽBE

1. Razmotrite izradu dvoprolaznog asemblera. Skicirajte njegove funkcije i opišite kako biste ga testirali da bi svaka funkcija bila temeljito testirana pre nego što se pređe na sledeću. Predložite i napišite plan razvoja i objasnite zašto se plan izgradnje i plan testiranja moraju projektovati zajedno.
2. Sertifikacija je potvrda nekog spoljnog autoriteta da je naš sistem ispravan. Sistem se često odobrava da osnovu njegovog poređenja sa unapred definisanim standardom performanse. Na primer, Uprava za odbranu SAD izdaje potvrdu za Ada prevodioce nakon testiranja prema dugačkom spisku funkcionalnih specifikacija. Prema terminologiji koju smo koristili u ovom poglavlju, može li se to nazvati funkcionalnim testom? Testom performanse? Testom prihvatanja? Instalacionim testom? Objasnite zašto.
3. Kada razvijate plan gradnje, morate uzeti u obzir dostupne resurse učesnika u razvoju i kupaca, uključujući tu vreme, osoblje i novac. Navedite primere ograničenja resursa koja mogu da utiču na broj delova definisanih u razvoju sistema. Objasnite na koji način ova ograničenja utiču na plan izgradnje.
4. Pretpostavimo da kalkulator ima funkciju za izračunavanje nagiba i odsečka prave. Zahtev u dokumentu zahteva glasi: „Kalkulator će na ulazu prihvatati jednačinu u obliku Ax + By + C = 0 i kao odgovor dati nagib i odsečak." Sistemska implementacija ovog zahteva je funkcija LINE sa sintaksom LINE(A,B,C), gde su A i B koeficijenti za x i y, a C je konstanta u jednačini. Rezultat su vrednosti za D i E, gde je D nagib, a E odsečak. Izrazite ovaj zahtev kao skup uzroka i posledica i nacrtajte odgovarajući graf uzroka i posledica.
5. U poglavlju 4 opisali smo da je potrebna mogućnost testiranja zahteva. Objasnite zašto je ovo svojstvo bitno za testiranje performanse. Potkrepite svoje objašnjenje primerima.
6. Kakvi bi testovi performanse mogli da se zahtevaju za sistem za obradu teksta? Za sistem platnih spiskova? Za automatizovani bankarski šalterski sistem? Za sistem kojim se nadgleda kvalitet vode? Za kontrolni sistem elektrane?
7. Kontrolni sistem za vazdušni saobraćaj može se projektovati za jednog ili više korisnika. Objasnite kako bi takav sistem mogao da ima različite konfiguracije i skicirajte kako bi se mogao projektovati skup testova konfiguracije.
8. Sistem za navigaciju treba da se instalira na avionu. Koja pitanja treba razmotriti prilikom projektovanja instalacionog testa?
9. Sledeći izveštaj CNN-a opisuje softverski element čija je pravilna implementacija mogla da spreči pad aviona korejskog avioprevoznika na letu 801 za Guam avgusta 1997. Koja vrsta testiranja je mogla da obezbedi proveru ispravnog rada komponente u dovoljno velikom okruženju oko aerodroma Guam?

Softverska greška pogodila radarski sistem na aerodromu Guam

10. avgust 1997.
Postavljeno na Internet u: 10:34 EDT (1434 GMT)
AGANA, Guam (CNN) - Avion korejske aviokompanije koji se srušio u Guamu prošle nedelje mogao je da bude upozoren da previše nisko leti da radarski sistem nije bio pogođen softverskom greškom. To su u nedelju javili istražitelji. Sistem po imenu FAA Radar Minimum Safe Altitude Warning izdaje upozorenje službenicima na zemlji koji zatim javljaju pilotu da avion leti previše nisko. Međutim, savezni agenti koji su istraživali nesreću kažu da je sistem koji se nalazi u vojnoj bazi SAD na tom ostrvu bio nedavno modifikovan i da je očigledno u softver uneta greška. Istražitelji iz Odbora za nacionalnu

bezbednost saobracaja SAD tvrde da se ne može sa sigurnošću reći da je ta greška kriva za pad aviona u kojem je poginulo 225 ljudi, ali da nije bilo te greške, pilot je mogao biti upozoren da podigne avion na veću visinu.

„Moguće...preventiva"

„To nije uzrok - možda je moglo da pomogne u sprečavanju", rekao je George Black član Odbora za nacionalnu bezbednost saobracaja. Istaživačima je na taj sistem skrenuo pažnju jedan operater kontrole leta koji im je rekao da nije dobio upozorenje pred rušenje aviona. Grešku su pronašli službenici Savezne uprave za avijaciju.

Sistem za upozoravanje o visini projektovan je da pokrije kružno područje prečnika 55 nautičkih milja (102 kilometra). Međutim, pošto je softver menjan, sistem je pokrivao samo jedan krug širine od jedne milje po obodu tog područja. Let 801 nije bio obuhvaćen kada se srušio.

Black kaže da je softver promenjen kako sistem ne bi više davao toliko lažnih upozorenja. „Izmena je izmenila previše", rekao je on.

Nije bilo odmah jasno koliko dugo je ta greška postojala i koliko aviona je sletelo na aerodrom nakon te modifikacije. Istraživači su napomenuli da proveravaju da li je greška uticala i na druge aerodrome, pošto se slična softverska oprema dostavlja aerodromima na celoj teritoriji SAD.

Avionska kompanija brani pilota

Novosti o softverskoj greški stigle su dok su zvaničnici korejske aviokompanije branili pilota nesrećnog Boeinga 747 kao veterana koji je bio više nego sposoban da upravlja avionom. Postojali su izveštaji koji su isticali mogućnost da je bila u pitanju greška pilota. „Park Yong-Chul bio je iskusan pilot sa skoro 9.000 časova leta", izjavili su u kompaniji Korean Air. U izjavi su takođe prikazali raspored letenja i odmora pilota Parka u nedelji koja je prethodila nesreći. Pre poslednjeg leta imao je odmor od 32 časa i 40 minuta.

Let 801 kompanije Korean Air udario je u brdo iznad međunarodnog aerodroma Guam u sredu ujutru. U avionu su bile 254 osobe, od toga 23 člana posade. Istražitelji kažu da je 29 ljudi preživelo nesreću. Istražitelji su javili da je pilot imao punu kontrolu nad letelicom u vreme pada i ponovo ispitaju gomile podataka i evidencija o letu da bi utvrdili zašto je tako nisko leteo.

Čak i bez sistema za upozoravanje, kažu istražitelji, pilot je imao još nekoliko drugih instrumenata koji su mogli da ga upozore da je avion isuviše blizu brda. „Ovo je samo jedan podatak", rekao je vodeći istražitelj Gregory Feith. „Da, moglo je da pomogne, ali koliko mi znamo to nije uzrok pada."

Postojali su i drugi problemi

Sistem za upozoravanje nije bio jedini neispravan deo opreme na aerodromu. Instrument za prikazivanje „nagiba poniranja", deo instrumenta za sletanje koji usmerava avione na pistu, nije bio u funkciji zbog redovnog održavanja. Avionska kompanija tvrdi da su znali da instrument nije tu.

U izjavi avionske kompanije tvrdi se da je do pada aviona moglo doći zbog kombinacije različitih problema u opremi i lošeg vremena. „Još uvek ne isključujemo mogućnost da je do iznenadne promene visine došlo zbog žestoke kiše, otkaza instrumenta za prikaz nagiba poniranja ili zbog drugih elemenata koji bi, u kombinaciji, mogli da budu odgovorni za nesreću," izjavili su u kompaniji Korean Air.

U subotu je avion prebacio pistu prilikom sletanja na aerodrom Guam, ali je uspeo da se stabilizuje i bezbedno sleti u drugom pokušaju. Nije bilo jasno zbog čega je avion promašio pistu u prvom naletu.

Prikupljanje tela iz nesreće otežavali su nepristupačan kamenjar na brdu gde se dogodila nesreća i loše stanje velikog broja tela.

Dopisnik Jackie Shymanski za Associated Press i Reuters doprinela je pravljenju ovog izveštaja.

10. Dajte primer iz kojeg se vidi da je testiranje ponekad nemoguće bez upotrebe simulatora uređaja. Dajte drugi primer iz kojeg se vidi da postoji potreba za simulatorom sistema.
11. Dajte komentar na obrazac za izveštaj o odstupanjima sa slike 9.15 imajući u vidu pitanja koja bi čitanje tog obrasca trebalo da nam razjasni.
12. Sistem za platni spisak je tako projektovan da se vode informacije o svakom pojedincu zaposlenom u kompaniji. Svake nedelje se zapis o radniku ažurira brojem časova koje je on te nedelje odradio. Svake dve nedelje štampa se izveštaj sa brojem odrađenih sati od početka fiskalne godine. Svakog meseca, plata svakog radnika za taj mesec elektronski se prenosi na njegov račun u banci. Za svaki tip testova performanse, opisanih u ovom poglavlju, navedite da li bi ga trebalo primeniti na ovaj sistem.
13. Kompanija Wellie's PLC je naručila u Robusta Systems da joj se razvije sistem sa podrškom računara koji će služiti za testiranje jačine za celokupnu proizvodnju gumene obuće. Willie's ima devet fabrika na različitim lokacijama u celom svetu i svaki sistem će se konfigurisati u skladu sa veličinom fabrike. Objasnite zašto bi Robusta i Willie's trebalo da izvrše instalaciono testiranje po završetku testa prihvatanja.

OBRAZAC IZVEŠTAJA O ODSTUPANJU

Broj izveštaja: _____ **Testiranje vršio:** _____
Datum: _____ **Vreme:** _____
Redni broj testa: _____
Korak skripta u trenutku otkaza: _____
Opis otkaza: _____

Aktivnost pred samo otkazivanje:

Očekivani rezultati:

Pogođeni zahtevi:

Uticaj otkaza na test:

Uticaj otkaza na sistem:

Stepen ozbiljnosti:
1 (MALI) 2 3 4 **5 (VELIKI)**

SLIKA 9.15 Primer obrasca za izveštaj o odstupanju

14. Napišite skript testa za funkcije NIVO koju smo opisali u ovom poglavlju.
15. U ovom poglavlju smo predložili meru pouzdanosti izraženu pomoću prosečnog vremena do otkaza, meru dostupnosti pomoću prosečnog vremena između dva otkaza, a meru mogućnosti održavanja pomoću prosečnog vremena do popravke. Da li su ove mere konsistentne sa definicijama predstavljenim u ovom poglavlju? To jest, ako se pouzdanost, dostupnost i mogućnost održavanja definišu kao verovatnoće, da li ćemo dobiti iste vrednosti kao sa pomenutim merama? Ako nije tako, da li se jedne mere mogu transformisati u druge ili postoje temeljne i nepomirljive razlike?
16. Otkazuje sistem u kome je bezbednost kritična i nekoliko ljudi strada. Prilikom istrage o uzroku otkaza, istražna komisija otkriva da je u planu testiranja izostavljeno razmatranje slučaja koji je doveo do otkaza sistema. Ko je odgovoran za smrt? Tim za testiranje koji nije primetio da nedostaje taj slučaj? Planeri testa koji nisu napisali potpun plan za testiranje? Rukovodioci koji nisu proverili plan za testiranje? Kupac zato što nije izvršio potpuni test prihvatljivosti?
17. Ako se za neki sistem zahteva ultra visoka pouzdanost, pa to znači da takvu pouzdanost nikada nije moguće u potpunosti proveriti, da li se sistem ipak može koristiti?
18. Ponekad kupci angažuju nezavisnu organizaciju (odvojenu od organizacije koja vrši razvoj) da uradi nezavisnu proveru valjanosti (*verification and validation*, V&V). V&V osoblje ispituje sve aspekte razvoja uključujući proces i sam proizvod da bi se proverio kvalitet konačnog proizvoda. Ako je korišćen nezavisan V&V tim, a sistem ipak doživi katastrofalan otkaz, koga treba smatrati odgovornim: rukovodioce, V&V tim, dizajnere, kodere ili testerski tim?
19. U ovom poglavlju smo uveli dve funkcije: funkciju distribucije, $F(t)$ i funkciju pouzdanosti $R(t)$. Ako se pouzdanost sistema povećava kako ga testiramo i popravljamo, šta se dešava na grafikonima ovih funkcija?
20. U podsetniku 9.6 opisane su dve verzije softvera VxWorks, jedna za čip 68000 i jedna za čip R6000. Objasnite pitanja koja se odnose na upravljanje konfiguracijom kada se gradi jedan sistem za dva različita čipa. Da li je strategija za upravljanje konfiguracijom mogla da upozori prodavca prilikom prenošenja verzije 68000 na R6000?
21. Predskazivač testa je zamišljena osoba ili mašina koja je u stanju da proceni kada su stvarni rezultati testa jednaki očekivanim. Objasnite potrebu za uključenje predskazivača testa u teoriju testiranja.
22. Skicirajte i izradite plan za testiranje sistema Piccadilly.

10 Isporučivanje sistema



U ovom poglavlju obradivaćemo:
- obuku;
- dokumentaciju.

Približavamo se kraju razvoja sistema. U prethodnim poglavljima pokazali smo kako da prepoznamo problem, projektujemo rešenje, implementiramo ga i isprobamo. Sada smo spremni da kupcu predstavimo rešenje i obezbedimo da sistem nastavi da radi ispravno.

Mnogi softverski inženjeri shvataju isporuku sistema kao formalnost - ceremoniju presecanja vrpce, ili predaju ključa računara. Međutim, čak i kod sistema „ključ u ruke" (gde isporučilac predaje sistem kupcu i nije odgovoran za njegovo održavanje), isporučivanje obuhvata više od postavljanja sistema na njegovo mesto. To je vreme u toku razvoja kada želimo da pomognemo korisnicima da razumeju kako radi naš proizvod i da se osećaju udobno služeći se njime. Ako isporuka nije uspešna, korisnici neće ispravno koristiti sistem i mogu biti nezadovoljni njegovom performansom. U oba slučaja, korisnici nisu tako produktivni ili efikasni kao što bi mogli biti, a trud koji smo uložili da izgradimo sistem visokog kvaliteta bio bi uzaludan.

U ovom poglavlju, istražićemo dva ključna pitanja za uspešnu primopredaju od razvijaoca korisniku: obuku i dokumentaciju. Kada projektujemo sistem, planiramo i razvijamo sredstva koja će pomoći korisniku da koristi taj sistem. Sistem prati dokumentacija kojoj se korisnici obraćaju za rešavanje nastalih problema ili za dalje informacije. Primeri takve dokumentacije su vam na raspolaganju na veb stranici koja prati ovu knjigu.

10.1 OBUKA

Sistem koriste dve vrste ljudi: korisnici i operateri. O njima možemo da razmišljamo na isti način kao o vozačima i mehaničarima. Glavna funkcija automobila je da obezbedi prevoz. Vozač koristi automobil da bi išao od jednog do drugog mesta. Međutim, mehaničar servisira ili podržava automobil da bi omogućio vozaču da ga vozi. Mehaničar nikada stvarno ne vozi automobil, ali bez dodatnih funkcija, koje koristi mehaničar, automobil uopšte ne bi radio.

Na isti način, **korisnik** izvršava glavne funkcije sistema, da bi mu one pomogle da reši problem opisan u dokumentu za definisanje zahteva. Dakle, korisnik je taj koji rešava problem za kupca. Međutim, sistem često ima dopunske zadatke koji podržavaju glavne funkcije sistema. Na primer, jedna dopunska funkcija može da definiše ko ima pristup sistemu. Neka druga periodično pravi rezervne kopije bitnih datoteka podataka, da bi omogućila oporavak od otkaza sistema. To su pomoćne funkcije koje korisnici obično ne izvršavaju direktno. Umesto toga, **operater** vrši te funkcije da bi podržao glavni rad. U tabeli 10.1 dati su primeri funkcija korisnika i operatera.

Vrste obuke

Ponekad je ista osoba i korisnik i operater. Međutim, zadaci korisnika i operatera imaju veoma različite ciljeve, pa se u obuci za svaki od ovih poslova ističu različiti aspekti sistema.

Obuka korisnika. Obuka za korisnike zasniva se najviše na glavnim funkcijama sistema i korisnikovim potrebama da im pristupa. Na primer, ako je sistem namenjen upravljanju zapisima u pravničkoj firmi, korisnik mora da bude obučen za funkcije upravljanja zapisima: njihovo stvaranje i uzimanje, menjanje i brisanje stavki itd. Pored toga, korisnici moraju da se kreću kroz zapise, da bi pristupili onima koji su im potrebni. Ako informacije treba da se zaštite pomoću lozinke ili od slučajnog uništenja, korisnici uče specijalne zaštitne funkcije.

U isto vreme, korisnici ne moraju da poznaju unutrašnji rad sistema. Oni mogu da sortiraju skup zapisa, a da ne znaju da li je to Šelovo, mehurasto ili brzo sortiranje. Korisnik koji pristupa sistemu ne mora da zna ko sem njega pristupa u isto vreme, ili na kom disku se nalazi tražena informacija. Imajući u vidu da su to funkcije podrške a ne glavne funkcije, njima se bavi jedino operater.

Obuka korisnika predstavlja osnovne funkcije, tako da korisnici razumeju šta su te funkcije i kako da ih izvode. Obuka se odnosi na to kako se funkcije izvode sada (sa postojećim sistemom) i kako će se one izvoditi kasnije, sa novim sistemom. To nije lako, zato što su korisnici često prisiljeni da odbace radnje koje već znaju, da bi naučili neke nove. (U psihološkim studijama, to se zove **interferencija zadataka**.) Sličnosti i suptilne razlike između starih i novih aktivnosti mogu da ometaju učenje, pa se u projektovanju obuke ta poteškoća mora uzeti u obzir.

TABELA 10.1 Funkcije korisnika i operatera

Funkcije korisnika	Funkcije operatera
Manipulisanje datotekama podataka	Odobravanje korisničkih pristupa
Simuliranje aktivnosti	Odobravanje pristupa datotekama
Analiziranje podataka	Pravljenje rezervnih kopija
Sopštavanje podataka	Instaliranje novih uređaja
Crtanje grafikona i dijagrama	Instaliranje novog softvera
	Obnavljanje oštećenih datoteka

Obuka operatera. Obuka operatera je usredsređena na ovladavanje funkcijama za podršku radu sistema i obuhvata objašnjenja o tome kako sistem funkcioniše, a ne šta radi. U ovom slučaju je manje verovatno da će zadaci uticati jedni na druge, izuzev kada sistem u velikoj meri oponaša neki drugi sistem sa kojim je operater ranije imao iskustvo.

Operateri se obučavaju na dva nivoa: kako da podignu i uspostave rad novog sistema i kako da pružaju podršku korisnicima. Prvo, operateri uče kako da konfigurišu sistem, administriraju pristup sistemu, dodeljuju veličine zadataka ili prostor na disku i nadgledaju i poboljšavaju performansu sistema. Nakon toga se operateri usredsređuju na posebne osobine razvijenog sistema: kako da oporave izgubljene datoteke ili dokumente, komuniciraju sa drugim sistemima i pozivaju različite pomoćne programe.

Potrebe za specijalnom obukom. Korisnici i operateri se obično obučavaju na usmerenom i sveobuhvatnom kursu za korišćenje sistema. Obuka često počinje sa osnovama: kako je konfigurisana tastatura, kako se bira iz menija itd. Ove i ostale funkcije se postupno uvode i detaljno obrađuju. Ovakva potpuna obuka se pruža budućim korisnicima sistema u fazi njegove isporuke.

Međutim, obučene korisnike kasnije mogu da zamene novi, često zbog prelaska na drugo radno mesto. Zato je neophodno obezbediti kontinuitet obuke.

Ponekad korisnici žele da razjasne stvari koje su propustili prilikom prvobitne obuke. Čak i kada je početna obuka bila potpuna, korisnicima i operaterima može biti teško da prihvate sve čemu su ih učili. Korisnici često vole da obnove neke funkcije koje su im prvobitno predstavljene u sklopu inicijalne obuke.

Potrebu za tim možete sagledati ako se prisetite kako je izgledalo učenje vašeg prvog programskog jezika. Naučili ste sve važeće iskaze, ali ste sintaksu i značenje nekih od njih zapamtili bolje od drugih. Da biste ovladali jezikom, vraćali ste se vašim beleškama ili udžbenicima, da biste obnovili komande koje ste ređe koristili.

Korisnici koji povremeno rade sa sistemom susreću se sa sličnim problemom. Znanje koje se na obuci dobije o funkcijama sistema koje se kasnije retko koriste može lako da se zaboravi. Na primer, zamislite sistem za obradu teksta u okviru velike korporacije. Primarni korisnici takvog sistema su daktilografi koji svakodnevno unose dokumente i prosleđuju ih sa jedne lokacije na drugu. Stalno korišćenje sistema pomaže daktilografima da ostanu upoznati sa većinom funkcija. Međutim, predsednik korporacije koristi sistem možda jednom ili dva puta nedeljno, da bi napravio dokument ili podsetnik kojima konačnu formu kasnije daje daktilograf. Obuka za korisnika koji retko koristi sistem je različita od standardne korisničke obuke. Predsednik ne mora da zna sve specijalne karakteristike sistema i zato se obuka za povremenog korisnika odnosi samo na osnovne funkcije izrade i izmene dokumenata.

Operateri se sreću sa sličnom poteškoćom. Ako se jedna funkcija sistema koristi polugodišnje, na primer, skladištenje arhivskog materijala na posebnom disku, operater može zaboraviti proceduru arhiviranja koju šest meseci nije sprovodio. Bez obnavljanja obuke, korisnici i operateri su skloni da koriste samo one funkcije koje su im dobro poznate, a izbegavaju one uz koje bi postali efikasniji i produktivniji.

Isto tako, mogu da se organizuju posebni kursevi obuke za one koji imaju specifične potrebe. Ako sistem proizvodi grafikone i izveštaje, neki korisnici treba da znaju kako da ih prave, dok drugi jedino treba da pristupaju postojećima. Obuka može da obuhvati samo ograničene funkcije sistema ili da se svede na obnavljanje jednog dela iz ukupnih funkcija sistema.

Pomoćna sredstva za obuku

Obuka može da se vrši na mnogo načina. Bez obzira na način obučavanja, ona korisnicima i operaterima mora da pruža informacije u svakom trenutku, a ne samo prilikom inicijalne isporuke sistema. Ako u nekom periodu korisnici zaborave kako da pristupaju datoteci ili koriste novu funkciju, mora se obaviti obuka vezana za pronalaženje i usvajanje potrebne informacije.

Dokumenti. Formalna dokumentacija prati svaki sistem i podržava obuku. Dokumenti sadrže sve informacije koje su potrebne da bi se sistem ispravno i efikasno koristio. Nezavisno od njihove forme (priručnik, onlajn dokument), dokumenti su dostupni korisnicima i operaterima u toku eksploatacije sistema. Sistemski priručnici su često slični priručniku za vlasnika automobila; njima se obraćate kada se pojavi neki problem ili pitanje. Ne morate ga pročitati od korica do korica pre nego što gurnete ključ za paljenje i povezete se; slično tome, korisnici i operateri obično ne čitaju materijale za obuku pre nego što pokušaju da koriste sistem. Jedna studija je pokazala da samo 10% do 15% korisnika na intenzivnom programu obuke uopšte pročita priručnik (Sharer 1983). Prilikom ispitivanja koje je obavljeno šest meseci kasnije, ustanovljeno je da niko više nije čitao korisnički priručnik, a stranice za ažuriranje informacija nisu ni stavljane u njega. U ovom i drugim slučajevima, korisnici u obuci za korišćenje sistema daju prednost dobro definisanim ikonama, onlajn pomoći, demonstracijama i lekcijama.

Ikone i onlajn pomoć. Sistem treba projektovati tako da njegove funkcije posredstvom korisničkog interfejsa budu jasne i razumljive. U većini računarskih sistema sledi se primer kompanija Apple i Xerox da ikone budu mehanizam za biranje funkcija sistema. Jednim pritiskom na taster miša bira se ikona, a sa dva se poziva funkcija. Obuka za takve sisteme je relativno jednostavna, zato što se funkcija lakše prisećamo pregledom ikona nego pamćenjem komandi i sintakse.

Slično tome, onlajn pomoć olakšava obuku. Korisnik može da pregleda dodatne informacije o nameni ili upotrebi funkcije, bez potrebe da odvaja vreme za pretraživanje dokumenata u papirnom obliku. Sofisticirana onlajn pomoć zahvaljujući tehnologiji hipermedija omogućava korisniku da informacije traži po skupovima povezanih onlajn dokumenata, do željenog nivoa detalja. Međutim, kao što je opisano u podsetniku 10.1, sistem automatizovane obuke zahteva održavanje kao i svaki drugi softverski sistem.

PODSETNIK 10.1 I SISTEMI ZA OBUČAVANJE SU SOFTVER

Važno je zapamtiti da obuka zasnovana na računaru često obuhvata složeni softver i da ima i loše strane. Oppenheimer (1997) ističe da se korisnici „usredsređuju na to kako da manipulišu softverom, umesto na razmatrani problem, pa spoznavanje nije sve veće, nego se smanjuje". On objašnjava da tehnike kao što je simulacija mogu da maskiraju mnoge naše pretpostavke koje činimo u vezi sa sistemom, umesto da nam ih otkriju i podstaknu nas da ih preispitamo. Zato dobijamo korisnike koji stvari uzimaju zdravo za gotovo, umesto onih sa svešću da je okvirno rešenje moguće menjati.

Demonstracije i lekcije. Demonstracije i lekcije individualizuju obuku, a odziv korisnika i operatera je pozitivan. Potrebe korisnika su najvažnije, a demonstracija ili lekcija se usredsređuju na poseban aspekt sistema. Demonstracije i lekcije se obično organizuju kao niz prezentacija, tako da se na svakom predavanju u nizu uči jedna funkcija ili jedan aspekt sistema.

Demonstracije i lekcije mogu da budu fleksibilnije ili dinamičnije nego papirni ili onlajn dokumenti. Korisnicima se dopada pristup u kome se pokazuje i objašnjava, gde mogu da isprobaju demonstriranu funkciju. Demonstracija može da bude formalna prezentacija u učionici. Međutim, obuka zasnovana na računaru ili Webu se pokazala uspešnom u demonstriranju i podučavanju koncepata i funkcija sistema.

Neki programi prilikom osmišljavanja obuke u nju uvrštavaju i multimediju na različite načine. Na primer, video disk, video traka ili propisane veb lokacije mogu da se upotrebe u cilju demonstracije funkcija, nakon čega polaznici mogu da isprobaju funkciju na sopstvenim računarima. Drugi softver i hardver za obučavanje, kao što je Robotel, dozvoljavaju nastavniku da nadgleda šta svaki polaznik radi ili da preuzme kontrolu nad njegovim sistemom da bi pokazao poseban niz komandi koje treba izabrati mišem ili pomoću tastera. Requirement Laboratory na Imperial College (u Londonu) koristi prednosti ove tehnologije.

Demonstracije i lekcije obično na različite načine ističu onog što studenti uče. Slušanje, čitanje i uvid u rad funkcije doprinose lakšem pamćenju funkcija i tehnika. Mnogim ljudima živa reč duže drži pažnju od pisane.

Ključni činilac za uspeh demonstracija i lekcija je davanje povratne informacije korisnicima. Instruktor, bilo na traci, Webu, televiziji ili lično, maksimalno hrabri polaznika.

Stručni korisnici. Ponekad nije dovoljno da se odgleda demonstracija ili učestvuje u lekciji. Potreban vam je vid glume da biste se uverili da možete vladati sistemom. U tom cilju korisno je postaviti jednog ili više korisnika ili operatera u ulogu „stručnjaka". Stručnjaci se obučavaju pre ostalih korisnika i koriste kao demonstratori ili pomoćnici u učionici. Drugi polaznici se tada osećaju lagodnije jer među stručnjacima prepoznaju korisnike slične sebi, ali koji su uspeli da ovladaju tehnikama. Stručnjaci mogu da ukažu na mesta gde su oni imali probleme koje su prevazišli. Na taj način, stručnjaci uveravaju polaznike da je ono što izgleda nemoguće, u stvari moguće.

Stručni korisnici mogu služiti kao povremeni instruktori, po završetku perioda formalne obuke. Oni deluju kao konsultanti, odgovarajući na pitanja i stavljajući se drugima na raspolaganje kada se problemi pojave. Mnogim korisnicima je neugodno da postave pitanje u učionici, ali neće oklevati da se sa istim pitanjem obrate drugom iskusnijem korisniku.

Stručni korisnici daju povratne informacije analitičarima sistema o tome kako su korisnici zadovoljni sistemom, potrebama za dodatnom obukom i pojavama otkaza. Korisnici ponekad imaju poteškoća da analitičarima objasne potrebu za promenom ili poboljšanjem sistema. Stručnjaci uče jezik i jednih i drugih i time pomažu da se izbegnu problemi koji se često javljaju u komunikaciji između korisnika i analitičara sistema.

Smernice za obuku

Obuka je uspešna jedino ako zadovoljava vaše potrebe i odgovara vašim sposobnostima. Lične prednosti, stilovi rada i organizacioni pritisci imaju udela u uspešnosti obuke. Ru-

kovodilac koji ne ume da kuca ili pravilno piše možda ne želi sekretara odeljenja koji to zna. Radniku može da bude neprijatno ili da se stidi da u učionici ispravlja svog pretpostavljenog. Neki studenti više vole da uče čitajući, drugi slušajući, a treći kombinujući različite tehnike.

Sistemi sa mogućnošću personalizacije obezbeđuju prevazilaženje razlika u predznanju, iskustvu i sklonostima. Dok nekom polazniku određeni koncept može da bude potpuno nepoznat i zato mu za njegovo proučavanje treba mnogo vremena, drugom je taj koncept poznat te ga može preskočiti. Čak i veština rada na tastaturi može da ima uticaja: vežbu koja zahteva dosta kucanja iskusan daktilograf može da obavi brže. Kako su predznanja različita, različiti delovi obuke se mogu primeniti na različite polaznike. Korisnici koji znaju da koriste tastaturu mogu da preskoče delove vezane za kucanje, a operateri koji su dobro upućeni u računarske koncepte ne treba da uče kako rade periferijski uređaji. Za one koji su već upoznati sa nekim funkcijama mogu se razviti pregledni moduli.

Materijal na časovima obuke ili demonstraciji treba da se podeli u prezentacione celine, a obim svake od njih treba da bude ograničen. Suviše materijala odjednom može da preoptereti, pa je bolje imati više kraćih nego manji broj dužih sesija.

Najzad, lokacija polaznika može da opredeli vrstu obuke. Instalacija na više stotina lokacija širom sveta podrazumeva bilo sistem zasnovan na Webu, bilo sistem za obučavanje koji se instalira na eksploatacionom računarskom sistemu umesto da se svi budući korisnici radi obuke dovode na jedno mesto.

10.2 DOKUMENTACIJA

Dokumentacija je deo sveobuhvatnog pristupa obuci. Kvalitet i vrsta dokumentacije mogu da budu od presudnog značaja ne samo za obuku, nego i za uspeh sistema.

Vrste dokumentacije

Ima više činilaca koje treba uzeti u obzir prilikom izrade dokumentacije za obuku ili referentne dokumentacije. Svaki od njih može da bude važan za utvrđivanje da li će se dokumentacija uspešno koristiti.

Vrsta publike. Sistem zasnovan na računaru koriste razni ljudi. Pored korisnika i operatera, drugi članovi razvojnog tima i osoblja kupca čitaju dokumentaciju kada se pojave pitanja ili kada se predlažu i uvode promene. Na primer, pretpostavimo da analitičar radi s kupcem da bi odredio da li da gradi novi sistem ili modifikuje stari. Analitičar analizira opšti opis sistema da bi razumeo šta i kako postojeći sistem radi. Taj opis namenjen analitičaru razlikuje se od onog koji je namenjen korisniku jer analitičar poznaje detalje iz oblasti računarstva koji za korisnika nisu važni. Slično tome, opisi koji su potrebni operaterima nisu nimalo značajni za korisnika.

Na primer, paket S-PLUS (verzija 3.0, iz firme MathSoft, Seattle, Washington) prati više knjiga, uključujući i onu koja se zove *Pročitaj prvo mene*. Svaka knjiga namenjena je različitoj publici i ima drugačiju namenu. *Pročitaj prvo mene* započinje s poglavljem *Vodič kroz dokumentaciju*, u kome se na četiri kratke strane opisuje svaka knjiga koja je uključena u dokumentaciju:

- *Polagani uvod u S-PLUS* za neiskusnog korisnika računara;
- *Brzi tečaj S-PLUS* za iskusnog korisnika računara;
- *S-PLUS: Korisnički priručnik* , gde se objašnjava kako da se počne, služi podacima i koristi naprednija grafika;
- *S-PLUS: Vodič kroz statističku i matematičku analizu*, gde se opisuje statističko modelovanje;
- *S-PLUS: Priručnik za programere*, gde se opisuju programski jezici S i S-PLUS;
- *S-PLUS: Dopuna priručnika za programere*, sa informacijama specifičnim za datu verziju softvera;
- *S-PLUS: Korisnički priručnik za Trellis Graphics*, gde se opisuje posebno grafičko svojstvo kao dopuna statističkoj analizi;
- *S-PLUS: Globalni indeks*, koji obezbeđuje unakrsno referenciranje između knjiga u sklopu dokumentacije.

Izradu kompletne dokumentacije moramo započeti analizom publike kojoj je dokumentacija namenjena. Priručnici i vodiči mogu da se pišu za korisnike, operatere, ljude iz sistemske podrške i ostale.

Korisnički priručnici. **Korisnički priručnik** je referentni vodič ili uputstvo iz koga uče korisnici sistema. Priručnik bi trebalo da bude kompletan i razumljiv, pa se u njemu sistem ponekad predstavlja po nivoima, počevši od opšte namene i napredujući ka detaljnim opisima funkcija. Na početku priručnika opisuje se njegova namena i poziva na dokumente ili datoteke u kojima se nalaze detaljnije informacije. Ta uvodna informacija je posebno korisna za uveravanje korisnika da dokument sadrži vrstu informacija koju oni traže. Radi lakše upotrebe navode se specijalni termini, skraćenice ili akronimi koji se koriste u priručniku.

Zatim se u priručniku sistem opisuje detaljnije. Pregled sistema sadrži sledeće stavke:

1. namenu ili ciljeve sistema;
2. mogućnosti i funkcije sistema;
3. svojstva, karakteristike i prednosti sistema, uključujući i jasnu sliku o tome šta se sa sistemom postiže.

Pregled ne treba da bude veći od nekoliko pasusa.

Na primer, *S-PLUS: Korisnički priručnik za*počinje odeljkom pod naslovom „Kako se koristi ova knjiga". U prvom pasusu objašnjava se namena *S-PLUS*-a: „moćna alatka za analizu podataka, koja vam obezbeđuje pogodna svojstva za istraživačku analizu podataka, savremene statističke tehnike i mogućnost da pravite svoje S-PLUS programe" (MathSoft 1995). Zatim sledi spisak ključnih tehnika koje će korisnik iz te knjige naučiti:

- izdavanje S-PLUS komandi;
- stvaranje jednostavnih objekata podataka;
- stvaranje S-PLUS funkcija;
- stvaranje i menjanje grafikona;

- manipulisanje podacima;
- podešavanje S-PLUS sesije.

Svakom korisničkom priručniku potrebne su uz tekst i prateće ilustracije. Na primer, dijagram kojim se opisuju ulazi i njihovi izvori, izlazi i njihova odredišta i glavne sistemske funkcije, pomaže korisnicima da shvate šta sistem radi. Slično tome, uz dijagram je i prateći tekst o korišćenoj opremi.

Sistemske funkcije bi trebalo da budu opisane jedna po jedna, nezavisno od detalja o samom softveru. To znači da korisnik treba da nauči šta, a ne kako sistem radi.

Bez obzira na to koje funkcije sistem izvršava, funkcionalni opis u korisničkom priručniku treba da obuhvati bar sledeće elemente:

- pregled glavnih funkcija i njihovih međusobnih odnosa;
- opis svake funkcije u formi niza ekrana koje korisnik može da vidi, namenu svakog od njih i rezultat izbora svake stavke iz menija ili pritiska na funkcijski taster;
- opis svih ulaza koje očekuje svaka funkcija;
- opis svih izlaza koje može da generiše svaka funkcija;
- opis posebnih svojstava koja svaka funkcija može da pozove.

Na primer, u korisničkom priručniku glavni delovi sistema S-PLUS objašnjavaju se uz napomenu da S-PLUS radi i grafikone i statistiku. Zatim se svaka funkcija objašnjava tako da korisnici mogu da je razumeju. U knjizi *S-PLUS: Korisnički priručnik* prvo se opisuju grafičke funkcije:

- tačkasti dijagrami;
- više dijagrama na jednoj stranici;
- histogrami;
- kutijasti dijagrami;
- razdvojeni simboli za grupe;
- legende;
- dijagrami normalne raspodele verovatnoće;
- upareni tačkasti dijagrami;
- brisanje;
- trodimenzionalni grafikoni;
- detaljniji dijagrami;
- ostali dijagrami;

a zatim statističke funkcije:

- problem sa jednim ili dva šablona za kontinualne podatke;
- analiza varijanse;
- generalizovani linearni modeli;
- generalizovani aditivni modeli;
- lokalna regresija;

- modeli zasnovani na stablu;
- analiza opstanka.

Potpun i iscrpan korisnički priručnik je beskoristan ako traženu informaciju u njemu ne pronađete brzo i lako. Loše napisan korisnički priručnik za posledicu ima frustrirane korisnike koji se nelagodno osećaju pri korišćenju sistema i ne koriste ga onako efikasno kao što bi mogli. Prema tome, korisna je svaka tehnika koja povećava čitljivost ili pristup informacijama, kao što su to rečnici, tabele, numerisanje, unakrsno referenciranje, predstavljanje bojama, dijagrami i višestruki indeksi. Na primer, tabela funkcijskih tastera se mnogo lakše razume od tekstualnog opisa njihovog rasporeda. Slično tome, jednostavan dijagram, kao što je tabela 10.2 može da pomogne korisniku u pronalaženju ispravne kombinacije tastera.

Priručnici za operatere. **Priručnik za operatere** predstavlja operaterima materijal na isti način kao i korisnički priručnik. Jedina razlika između priručnika za operatere i korisničkog priručnika je u publici kojoj su namenjeni: korisnici žele da znaju detalje o sistemskim funkcijama i njihovoj upotrebi, a operateri detalje o sistemskim performansama i pristupu. Prema tome, u priručniku za operatere objašnjavaju se konfiguracije hardvera i softvera, metode za odobravanje i zabranu pristupa korisniku, procedure za dodavanje i uklanjanje periferijskih uređaja iz sistema i tehnike za dupliranje ili izradu rezervnih kopija datoteka i dokumenata.

TABELA 10.2 Uređivanje komande linije preko tastature (MathSoft 1995)

Akcija	Taster
Ponavljanje prethodne komande	Strelica nagore
Sledeća komanda	Strelica nadole
Ponovo pozivanje desete komande unazad	PgUp
Ponovo pozivanje prve izdate komande	Ctrl + PgDn
Ponovo pozivanje desete komande unapred	PgDn
Ponovo pozivanje poslednje izdate komande	Ctrl + PgUp
Kraj reda	End
Početak reda	Home
Nazad jednu reč	Ctrl + strelica ulevo
Napred jednu reč	Ctrl + strelica udesno
Brisanje komandne linije	Esc
Brisanje levo od znaka za umetanje	Backspace
Brisanje desno od znaka za umetanje	Delete
Umetanje na mestu znaka za umetanje	Unose se željeni znakovi
Biranje levo od znaka za umetanje	Shift + strelica ulevo
Biranje desno od znaka za umetanje	Shift + strelica udesno
Biranje do kraja linije	Shift + End
Biranje do početka linije	Shift + Home
Traženje odabranog teksta	Funkcijski taster 8
Kopiranje izabranog teksta na klipbord	Ctrl + Delete
Isecanje izabranog teksta na klipbord	Shift + Delete
Brisanje izabranog teksta	Delete

Baš kao što se korisnik upoznaje sa sistemom po nivoima, to isto radi i operater. Prvo se daje opšti pregled sistema, a zatim detaljniji opis njegove namene i funkcija.

Priručnik za operatere može donekle da se preklopi sa korisničkim priručnikom, zato što i operateri treba da budu svesni sistemskih funkcija, iako ih nikada neće koristiti. Na primer, operateri ne moraju nikada da naprave tabelarni proračun i iz njega izvuku grafikone i dijagrame. Saznanje o tome da sistem ima takve mogućnosti pomaže im da bolje podržavaju njegov rad. Na primer, operater može da nauči nazive softverskih rutina koje izvode funkcije tabelarnih proračuna i grafikona i hardvera koji se koristi da bi se crtali i štampali grafici. Na osnovu toga ako korisnik izvesti o problemu u vezi sa grafičkom funkcijom, operater može da zna da li problem može da se reši na osnovu funkcija podrške, ili o tome treba da se obavesti osoblje zaduženo za održavanje.

Opšti vodič kroz sistem. Ponekad želite da naučite šta radi sistem, bez detalja o svakoj njegovoj funkciji. Na primer, kao šef revizorskog odeljenja u velikoj kompaniji, možete da pročitate opis sistema da biste odlučili da li sistem odgovara vašim potrebama. Taj opis sistema ne treba da sadrži svaki sistemski ekran i njegove opcije. Međutim, detalji bi trebalo da vam dozvole da odlučite da li je sistem kompletan ili dovoljno tačan za potrebe vaše kompanije.

Opšti vodič kroz sistem se time bavi. On je više namenjen kupcu nego projektantu. Opšti vodič kroz sistem je sličan projektnoj dokumentaciji sistema; u njemu se opisuje rešenje problema u terminima koje kupac može da razume. Pored toga, u opštem vodiču kroz sistem daje se slika hardveske i softverske konfiguracije i opisuje filozofija na kojoj počiva konstrukcija sistema.

Opšti vodič kroz sistem je sličan blistavoj netehničkoj brošuri koju budućim kupcima daju prodavci automobila. Automobil se opisuje u terminima koji se odnose na: vrstu i veličinu motora i karoserije, statističke podatke o performansi, ekonomičnost potrošnje goriva, standarde, opciona svojstva itd. Kupca ne mora da zanima tačna konstrukcija ubrizgivača goriva da bi odlučio da li hoće, ili neće da kupi automobil. Slično tome, u opštem vodiču kroz sistem ne moraju se dati algoritmi koji se koriste za izračunavanje adrese sledećeg zapisa ili komande koja se upotrebljava za pristup tom zapisu. Umesto toga, u vodiču se opisuju samo informacije koje su potrebne da bi se napravio novi zapis i da bi mu se pristupilo.

Dobar opšti vodič kroz sistem obezbeđuje unakrsno referenciranje. Ako čitaoci vodiča žele više informacija o tačnom načinu na koji je funkcija implementirana, oni nalaze referencu na odgovarajuće strane korisničkog priručnika. S druge strane, ako čitaoci žele više informacija o podršci sistemu, oni se upućuju na priručnik za operatere.

Uputstva za učenje i automatizovani pregledi sistema. Neki korisnici više vole da prođu kroz sistem oslonjajući se na njegove stvarne funkcije nego posredstvom tekstualnog opisivanja rada funkcija. Za takve korisnike mogu se razviti automatizovana uputstva za učenje i pregled sistema. Korisnik poziva program ili procedure pomoću kojih se glavne funkcije sistema objašnjavaju korak po korak. Ponekad je dokument kombinovan sa posebnim programom; korisnik najpre čita o funkciji, a zatim prelazi na sledeći programski korak da bi tu funkciju izvršio.

Ostala sistemska dokumentacija. Isporuku sistema prati niz drugih dokumenata. Neki od njih nastaju kao produkti međufaza u razvoju sistema. Na primer, dokument o zahtevima se piše posle analize zahteva i po potrebi se ažurira. Dizajn sistema je opisan

u projektnoj sistemskoj dokumentaciji, dok je konstrukcija programa opisana u odgovarajućem dokumentu.

Implementacioni detalji su opisani u programerskoj dokumentaciji koju smo opisali u poglavlju 7. Međutim, jedan dopunski dokument može da pomogne onima koji će održavati i unapređivati sistem. **Vodič za programere** je tehnički pandan korisničkom priručniku. Baš kao što se u korisničkom priručniku predstavlja slika sistema po nivoima, od opšteg pregleda sistema do opisa funkcija, tako se u vodiču za programere daje pregled konfigurisanosti hardvera i softvera. Pregled je praćen detaljnim opisom softverskih komponenti i njihovim odnosom prema funkcijama. Da bi programer mogao da pronađe kôd koji izvršava određenu funkciju, bilo zbog otkaza, izmene ili unapređenja funkcije, vodič za programere unakrsno referencira korisnički priručnik.

U vodiču za programere takođe se ističu oni aspekti sistema koji omogućavaju inženjerima održavanja da pronađu izvor problema. Tu se opisuju funkcije podrške kao što su izvršavanje dijagnostičkih programa, prikaz izvršenih redova koda ili segmenata memorije, raspored koda za otkrivanje i ispravljanje grešaka i drugi alati. Tehnike održavanja detaljnije ćemo istražiti u poglavlju 11.

Vodič za programere takođe pomaže inženjerima održavanja da implementiraju unapređenja sistema. Na primer, pretpostavite da u komunikacionoj mreži treba dodati novu lokaciju. Vodič za programere ukazuje na delove koda koji se bave komunikacijama; objašnjava raspoložive alate za ažuriranje koda i prateće dokumentacije.

Pomoć za korisnika i otkrivanje i otklanjanje problema

Korisnici i operateri se obraćaju dokumentaciji da bi utvrdili uzrok problema i, ako je to potrebno, pozvali pomoć. Korisniku se obezbeđuje više vrsta pomoći uključujući referentne dokumente i onlajn datoteke za pomoć.

Referentni vodič kroz poruke o otkazima. Ako sistem otkrije otkaz, korisnici i operateri se o tome obaveštavaju na jednoobrazan i konsistentan način. Na primer, ako unesete dva imena ili broja, kao „3X", bez ikakvog operatora ili drugog sintaksnog elementa koji ih povezuje, S-PLUS 3.0 daje sledeću poruku o grešci:

```
> 3 x
Syntax error : name ("x") used illegaly at this point :
3 x
```

Ili vam se može reći da izraz nije funkcija:

```
> .5(2,3) Error: "0.5" is not a function
```

Setite se da se u projektu sistema predlaže filozofija za otkrivanje, izveštavanje i rad sa otkazima. Projektom je obuhvaćen širok spektar poruka o otkazima sistema, a korisnička dokumentacija daje spiskove svih mogućih poruka i njihovih značenja. Kad god je to moguće, poruke o otkazima ukazuju na neispravnost koja je dovela do otkaza. Međutim, ponekad uzrok nije poznat, ili nema dovoljno mesta da se prikaže cela poruka na ekranu ili u izveštaju. Zato **referentni vodič kroz poruke o otkazima**, budući da predstavlja

poslednje raspoloživo sredstvo, mora da u potpunosti opiše otkaz. Poruka o otkazu na ekranu može da obuhvati sledeće informacije:

1. naziv komponente koda koja se izvršavala kada se pojavio otkaz;
2. broj reda izvornog koda u komponenti koja se izvršavala;
3. ozbiljnost otkaza i njegov uticaj na sistem;
4. sadržaj svake relevantne sistemske memorije ili pokazivača na podatke, kao što su registri ili pokazivači steka;
5. prirodu otkaza, ili broj poruke o otkazu (radi unakrsnog referenciranja sa referentnim vodičem kroz poruke o otkazima).

Na primer, na korisnikovom ekranu može da se pojavi sledeća poruka o otkazu:

```
FAILURE 345A1: STACK OVERFLOW
OCCURED IN: COMPONENT DEFRECD
AT LINE: 12300
SEVERITY: WARNING
REGISTER CONTENTS: 0000 0000 1100 1010 1111 0000
```

Korisnik upotrebljava broj otkaza da bi pogledao u referentni vodič, u kome ta stavka izgleda ovako:

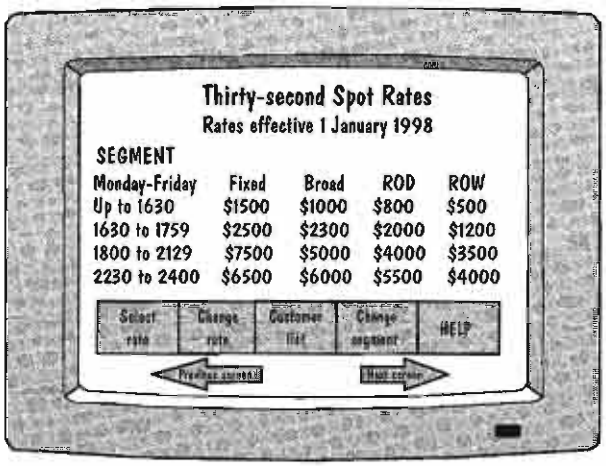
> Otkaz 345A1: Prekoračenje steka
> Ovaj problem se pojavljuje kada se definiše više polja za zapis nego što sistem može da prihvati. Poslednje definisano polje neće biti uvršteno u zapis. Veličinu zapisa možete promeniti korišćenjem funkcije Record Maintenance iz menija Maintenance u cilju sprečavanja ponovne pojave ovog otkaza.

Imajte u vidu da poruka o otkazu odražava određenu filozofiju za rukovanje greškama i otkazima. U okviru alternativnog dizajna sistema, oporavak bi se mogao automatizovati kroz interakciju sa korisnikom ili u sklopu pozadinske aktivnosti. U primeru koji smo opisali korisnik je taj koji treba da reši problem.

Onlajn pomoć. Mnogi korisnici vole da im je automatizovana pomoć na dohvat ruke, a ne da je traže oslanjajući se na referentni vodič. Neki sistemi uključuju funkciju onlajn pomoći. Često se na ekranu nalazi funkcija pomoći kao meni koji može da se otvori, ili je za njeno pokretanje potrebno pritisnuti funkcijski taster na kome piše „Help (F1)".

Za detaljnije informacije može da se omogući biranje druge ikone ili pritisak na drugi taster. Neki sistemi vas takođe upućuju na neku stranicu u dokumentu za podršku. Dakle, informaciju možete da dobijete direktno iz automatizovanog sistema, a ne da je tražite u dokumentu.

Brzi referentni vodiči. Korisno srednje rešenje je **brzi referentni vodič**. Ovaj zbirni pregled glavnih sistemskih funkcija i njihove upotrebe osmišljen je s ciljem da bude podsetnik, na stranici-dve, koji korisnici ili operateri drže pored radne stanice. Pregledanjem vodiča možete da pronađete kako da izvršite često korišćene ili najvažnije funkcije, bez potrebe da čitate dugačko objašnjenje. Takav vodič je posebno važan kada morate da zapamtite definicije specijalnih funkcionalnih tastera ili da koristite kodove i skraćenice (kao u UNIX-ovim komandama). U nekim sistemima, brzi referentni vodič je na raspolaganju na ekranu i može se prikazati pritiskom na funkcijski taster.



SLIKA 10.1 Ekran sistema Piccadilly

10.3 PRIMER IZ INFORMACIONIH SISTEMA

Sistem Piccadilly verovatno ima mnogo korisnika koji dobro poznaju pravljenje televizijskih programa ili oglašavanje prodaje, ali nisu obavezno dobri poznavaoci računarstva. Zato bi sistem trebalo da ima obimnu korisničku dokumentaciju i funkcije za pomoć. Mnogi ljudi više vole da uče na poslu nego da posećuju višednevne tečajeve, pa obučavanje za Piccadilly može da se obavi onlajn oslanjajući se na ekrane stvarnog sistema.

Na primer, korisnici moraju da nauče kako da menjaju cene oglašavanja, biraju cene da bi izračunali troškove oglašavanja za kupce i da se kreću kroz ekrane sistema da bi dobili dodatne informacije. Sistem za obuku upoznaje korisnika sa stvarnim Piccadillyjevim ekranima, kao što je onaj prikazan na slici 10.1. Potom može da se doda softver za obuku da bi dozvolio korisnicima da razumeju prirodu i namenu svake sistemske funkcije. Pretpostavimo da korisnik čita ekran sa cenama spotova, kao što je prikazano. Ako mišem izabere reči „Spot Rates", korisnik može automatski da dobije ekran za obuku na kome je opisano značenje termina i ukazano na dodatne datoteke za pomoć koje se odnose na funkcije cene spotova.

Ovakav pristup obuci je koristan ne samo za nove korisnike, nego i za one povremene, kojima su potrebni podsetnici kako se radi promena cena i, uopšte, kako se izvršavaju funkcije sistema. Na taj način obuku može da koristi svako, u bilo kom delu životnog veka sistema.

Ova vrsta softvera za obuku veoma napredna. On mora da bude u interakciji sa sistemom Piccadilly da bi normalan rad bio moguć, a takođe i da obezbedi dodatna objašnjenja i manipulisanje da bi se odradile vežbe tokom obuke. Zbog toga se o obuci i dokumentaciji ne sme razmišljati kasnije; one moraju da se projektuju zajedno sa ostatkom sistema i da održavaju sve promene u sistemu i njegovo širenje. Za razvoj softvera za obuku obično je potrebno dosta vremena, što treba da bude planirano kao normalni deo aktivnosti upravljanja projektom. U stvari, obuka i dokumentacija se obično smatraju svojstvima sistema, koja su opisana u dokumentima o zahtevima i koja evoluiraju sa ostatkom sistema.

10.4 PRIMER IZ PRAKSE

Istražitelji koji su ispitivali razloge otkaza Ariane-5 zapazili su da neke pretpostavke o ponovno upotrebljenom softveru iz Ariane-4 nisu bile evidentirane u dokumentaciji:

> ... sistemske specifikacije za SRI ne ukazuju na operativna ograničenja koja se javljaju usled odabrane implementacije. Takvo deklarisanje ograničenja bi moralo biti obavezno za svaki uređaj sa kritičnom misijom, a poslužilo bi da se identifikuje svaka neusaglašenost sa putanjom Ariane-5. (Lions i dr. 1996.)

Dakle, ponovna upotreba softvera iz Ariane-4 naglašava potrebu za kompletnom sistemskom dokumentacijom. Kada su razmatrali ponovnu upotrebu koda SRI iz Ariane-4, projektantima Ariane-5 je moralo da se obezbedi da pročitaju osnovne pretpostavke svojstvene kôdu. Posebno, dokumentacija Ariane-4 je trebalo da eksplicitno ukaže da kôd SRI nije bio namenjen za toliko dug rad koliko je radio na Ariane-5. Znači, projektantske odluke za Ariane-4 trebalo je da budu obuhvaćene u projektnim dokumentima za SRI Ariane-4 i pažljivo ispitane od strane projektanata Ariane-5. Mnogo je jednostavnije da se projektantske odluke uključe u dokumentaciju nego da projektanti budu prinuđeni da čitaju kôd kako bi shvatili da li je on funkcionalan i koliko je ograničen.

10.5 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju smo razmatrali obuku i dokumentaciju koji su potrebni kao prateći elementi prilikom isporučivanja sistema. Kao projektant, trebalo bi da zapamtite sledeće:

- obuka i dokumentacija treba da se planiraju i prate od početka projekta;
- softver za obuku i dokumentaciju treba da bude integrisan sa redovnim softverom sistema;
- svi moduli za obuku i dokumentaciju bi trebalo da uzmu u obzir promenljive potrebe različite publike: korisnika, operatera, kupaca, programera i drugih koji rade ili su u interakciji sa sistemom.

10.6 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Vaš razvojni tim ne sme da ostavi obuku i dokumentaciju za poslednji minut. Obuka i dokumentacija mogu da se planiraju čim se završi analiza zahteva. U stvari, korisnički priručnik bi trebalo da se napiše na osnovu zahteva, u isto vreme kada ispitivači pišu scenarije za testove sistema i prihvatanja. Na taj način, instruktori i dokumentaristi bi mogli da održavaju stalan kontakt sa svim projektantima da bi se promene u sistemskim zahtevima i projektu mogle preneti u dokumentaciju i materijal za obuku.

Pored toga, dopuna obuke i dokumentacije treba da se planira na vreme. Razvojni tim može da uspostavi mehanizme koji bi omogućili pristupanje dopunama na centralnoj lokaciji ili automatsko otpremanje do korisnika preko interneta ili veb lokacija. Dokumentacija posebno treba da bude ažurna, da daje podatke o poznatim greškama (i načinima njihovog zaobilaženja), novoj funkcionalnosti, novim ispravkama grešaka i druge informacije u vezi sa otkazima. Prilikom ažuriranja neophodno je obavestiti korisnike, operatere i programere o novim kursevima za obuku ili obnavljanje znanja, pitanji-

ma koja se često postavljaju, korisničkim skupovima i proslediti im i druge upotrebljive informacije.

10.7 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Postoji dosta istraživanja u oblasti obrazovanja i obuke i bilo bi pametno da ih istraživači u oblasti softverskog inženjerstva prouče kada dizajniraju obuku i dokumentaciju za sisteme koji sadrže softver. Treba da znamo više o:

- najpoželjnijim vrstama obuke i dokumentacije za korisnike i operatere;
- odnosu između interfejsa čovek-računar i očuvanja ideja;
- odnosu između interfejsa čovek-računar i ličnog pogleda na učenje;
- delotvornim načinima za brzo i efikasno dostavljanje informacija svima kojima su potrebne.

Istraživači takođe mogu da rade na novim podsticajima za interakciju između korisnika. Na primer, onlajn grupa korisnika može da im pomogne da nauče nove trikove, razmenjuju informacije o prilagođavanju aplikacija i da razumeju najefikasnije načine za korišćenje sistema.

10.8 SEMINARSKI RAD

Napišite korisnički priručnik za sistem *Aranžer pozajmica*. Namenjen je analitičarima kredita. U svom korisničkom priručniku, napravite osvrt na ostale dokumente za koje mislite da bi trebalo da budu na raspolaganju, kao što je onlajn pomoć.

10.9 KLJUČNE REFERENCE

Američko društvo za obuku i razvoj (*American Society for Training and Development*, ASTD) podržava konferencije, kurseve za obuku i informacije o obuci i razvoju. ASTD takođe izdaje časopis, *Training and Development*.

10.10 VEŽBE

1. Izrada prototipa dozvoljava korisnicima da isprobaju radni model sistema pre nego što je stvarni sistem završen. Objasnite kako izrada prototipa može da bude kontraproduktivna ako stvara međuđejstva zadataka za vreme obuke.
2. Dajte primer sistema za koji su obuka korisnika i obuka operatera iste.
3. Korisnik automatizovanog sistema ne treba da poznaje računarske koncepte. Međutim, poznavanje računara je korisno većini operatera. U kojim slučajevima korisnik automatizovanog sistema ne mora da zna o računarskom sistemu na kome je zasnovan automatizovani sistem? Da li je to neznanje znak dobre konstrukcije sistema? Dajte primere koji podržavaju vaš odgovor.

4. Ispitajte korisničku dokumentaciju za vaš računarski sistem u školi ili na poslu. Da li je ona jasna i lako razumljiva? Da li bi bila razumljiva korisniku koji malo zna o računarima? Da li se lako tumače poruke o otkazima? Postoji li spisak poruka o otkazima u korisničkom priručniku? Da li je lako pregledati teme u dokumentaciji? Kako biste promenili dokumentaciju da biste je poboljšali?
5. Pretpostavite da je filozofija sistema da rešava problem „u pozadini", bez znanja korisnika. U bezbednosno kritičnom sistemu koje su legalne i etičke implikacije neupoznavanja korisnika sa činjenicom da se dogodio otkaz? Da li bi sistem trebalo da izvesti o otkazu i korektivnoj akciji?
6. U tabeli 10.3 nalaze se neke od poruka o otkazima u referentnom vodiču za stvarni BASIC interpreter. Komentarišite jasnost, količinu informacija i pogodnost za korisnika ili operatera.

TABELA 10.3 Poruke o otkazima za BASIC

Broj	Poruka
23	Prekoračenje bafera reda Napravljen je pokušaj da se unese red koji ima suviše znakova.
24	Isteklo vreme Uređaj koji ste tražili nije raspoloživ u ovom trenutku.
25	Greška uređaja Unesen je neispravan identifikator uređaja.
26	FOR bez NEXT Naišlo se na instrukciju FOR bez odgovarajuće instrukcije NEXT.
27	Nema papira Uređaju za štampanje je nestalo papira.
28	Greška koja se ne može štampati Za postojeći uslov ne postoji poruka o otkazu.
29	WHILE bez WEND Naišlo se na instrukciju WHILE bez odgovarajuće instrukcije WEND.
30	WEND bez WHILE Naišlo se na instrukciju WEND bez odgovarajuće instrukcije WHILE.
31-40	Greška koja se ne može štampati Za postojeći uslov ne postoji poruka o otkazu.

11 Održavanje sistema



U ovom poglavlju obradivaćemo:
- evoluciju sistema,
- nasleđene sisteme,
- analizu uticaja,
- podmlađivanje softvera.

U prethodnim poglavljima smo istraživali kako da izgradimo sistem. Međutim, životni vek sistema ne završava se isporukom. U poglavlju 9 videli smo da je konačni sistem obično predmet stalnih izmena, čak i kada je izgrađen. Zato se sada okrećemo izazovu održavanja sistema koji stalno evoluira. Prvo, razmotrićemo one aspekte sistema koji su verovatno podložni izmenama. Zatim ćemo proučiti aktivnosti i osoblje koji su obuhvaćeni održavanjem sistema. Proces održavanja može biti težak; ispitaćemo probleme u vezi sa tim, uključujući prirodu troškova i kako oni rastu. Najzad, razmotrićemo alate i tehnike koje nam pomažu da poboljšamo kvalitet sistema, u toku njegove evolucije.

11.1 SISTEM KOJI SE MENJA

Razvoj sistema se završava kada se on uvede u eksploataciju, odnosno, kada ga korisnici koriste u stvarnom eksploatacionom okruženju. Sve aktivnosti vezane za izmenu sistema posle njegovog uvođenja u eksploataciju smatraju se **održavanjem**. Obično se o održavanju softvera razmišlja na isti način kao da se radi o održavanju hardvera: da treba zamenjivati ili sprečavati pojavu polomljenih ili neispravnih radnih delova. Međutim, održavanje softvera ne može da se posmatra na isti način. Da pogledamo zašto.

Jedan od ciljeva softverskog inženjerstva je razvijanje tehnika koje tačno definišu problem, projektuju sistem kao rešenje, implementiraju ispravan i efikasan skup programa i ispituju sistem na neispravnosti. Taj cilj je sličan i za projektante hardvera: izgradnja pouzdanog sistema bez neispravnosti u skladu sa specifikacijom. Održavanje hardvera u takvom sistemu usredsređuje se na zamenu delova koji su istrošeni ili upotrebu tehnika koje produžavaju životni vek sistema. Međutim, iskaz *whiledo* se ne troši posle 10 000 petlji, a znaci tačka zapeta ne otpadaju sa krajeva iskaza; za razliku od hardvera, softver se u operativnom radu ne degradira, niti zahteva periodično održavanje. Dakle, softverski

sistemi se razlikuju od hardverskih i na njih se ne može primeniti analogija s hardverom onako kako to činimo kada je reč o nekim drugim aspektima softverskog inženjerstva.

Tipovi sistema

Najveća razlika između hardverskih i softverskih sistema je u tome što su softverski sistemi izgrađeni tako da inkorporišu promene. Izuzev u najjednostavnijim slučajevima, sistemi koje razvijamo su evolucioni. To znači da se jedna ili više karakteristika koje definišu sistem menja u toku njegovog životnog veka. Lehman (1980) je opisao kategorizaciju programa na osnovu načina na koji ih je moguće menjati. U ovom odeljku razmotrićemo kako se njegova kategorizacija primenjuje na sisteme.

Softverski sistemi se mogu menjati ne samo na osnovu odluke kupca da u svoj domen rada unese neku izmenu već i zbog promenljive prirode samih sistema. Posmatrajmo na primer sistem koji služi za obračun zarada i proizvodi čekove za isplatu zarada. Sistem zavisi od poreskih zakona i lokalnih gradskih, državnih ili pokrajinskih propisa, kao i propisa u zemlji porekla date kompanije. Ako se poreski zakoni menjaju ili ako kompanija premesti sedište na drugu lokaciju, moguće su izmene unutar sistema. Otuda promene u sistemu mogu biti potrebne iako je u bliskoj prošlosti on radio ispravno.

Zašto su neki sistemi izloženiji promenama od drugih? U opštem slučaju sistem možemo opisati preko njegovog odnosa sa okruženjem u kome radi. Za razliku od programa koji funkcionišu u apstraktnom svetu, stvarni svet sadrži neodređenosti i koncepte koje ne razumemo u celosti. Što više sistemski zahtevi zavise od realnog sveta, veća je verovatnoća da će se on menjati.

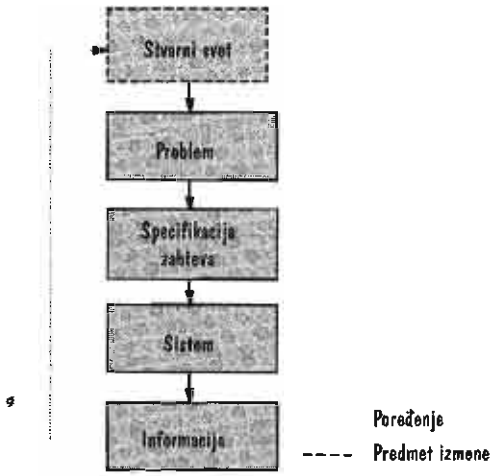
S-sistemi. Neki sistemi se formalno definišu u okviru specifikacije i iz nje se izvode. Kod ovih sistema razmatrani problem se iskazuje kroz potpun skup okolnosti na koje se on odnosi. Na primer, može se postaviti zahtev za realizaciju sistema koji izvršava sabiranje, množenje i inverziju matrica na datom skupu matrica sa zadatim ograničenjima na nivou performanse. Problem je potpuno definisan i za njega postoji jedno ili više ispravnih rešenja. Rešenje je dobro poznato, pa se projektant ne bavi ispravnošću rešenja već njegovom pravilnom implementacijom. Sistem koji nastaje na ovakav način zove se **S-sistem**. Takav sistem je statičan i ne prilagođava se lako promeni u domenu problema iz koga je nastao.

Kao što je prikazano na slici 11.1, problem koji se rešava pomoću S-sistema povezan je sa stvarnim svetom, a on je predmet izmena. Međutim, ako se svet promeni, javlja se potpuno nov problem koji se mora definisati.

P-sistemi. Stručnjaci iz oblasti računarskih nauka često definišu apstraktne probleme oslanjajući se na S-sisteme, a zatim razviju sisteme koji ih rešavaju. Međutim, nije uvek lako ili moguće potpuno opisati problem iz stvarnog sveta. U mnogim slučajevima, teoretski postoji rešenje problema, ali je njegova implementacija nepraktična ili nemoguća.

Na primer, zamislite sistem koji igra šah. Pošto su šahovska pravila potpuno definisana, moguće je potpuno specifikovati problem. U svakom koraku igre, rešenje bi moglo da obuhvati proračun svih mogućih poteza i njihovih posledica u cilju određivanja najboljeg sledećeg poteza. Međutim, implementiranje takvog rešenja u celosti nemoguće je oslanjanjem na današnje tehnologije. Broj poteza je suviše velik da bi se procenio u dovoljno

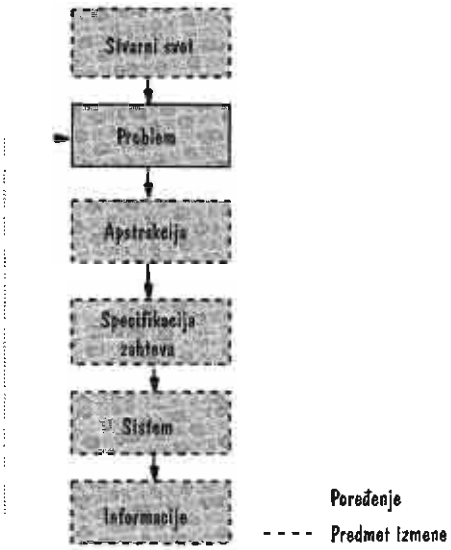
kratkom vremenu. Zbog toga se mora formulisati približno rešenje koje je praktičnije za realizaciju i korišćenje.



SLIKA 11.1 S-sistem

Da bismo razvili to rešenje, problem opisujemo na apstraktan način i zatim sa tog apstraktnog stanovišta formulišemo specifikaciju zahteva. Sistem koji se razvija na taj način zove se **P-sistem**, zato što se zasniva na praktičnoj apstrakciji problema, a ne na potpuno definisanoj specifikaciji. Kao što je prikazano na slici 11.2, P-sistem je dinamičniji od S-sistema. Rešenje proizvodi informaciju koja se poredi sa problemom; ako ona na bilo koji način nije odgovarajuća, apstrakcija se menja, a zahtevi modifikuju u cilju dobijanja realističnog rešenja.

SLIKA 11.2 P-sistem



Dakle, u P-sistemu zahtevi se zasnivaju na aproksimaciji. Rešenje delimično zavisi od interpretacije analitičara koji generiše zahteve. Čak uz pretpostavku da i postoji tačno rešenje, rešenje koje daje P-sistem svodi se na okruženje u kome nastaje. Kod S-sistema, rešenje je prihvatljivo ako su specifikacije tačne. Međutim, kod P-sistema rešenje je prihvatljivo ako rezultati imaju smisla u okruženju iz koga je problem izveden.

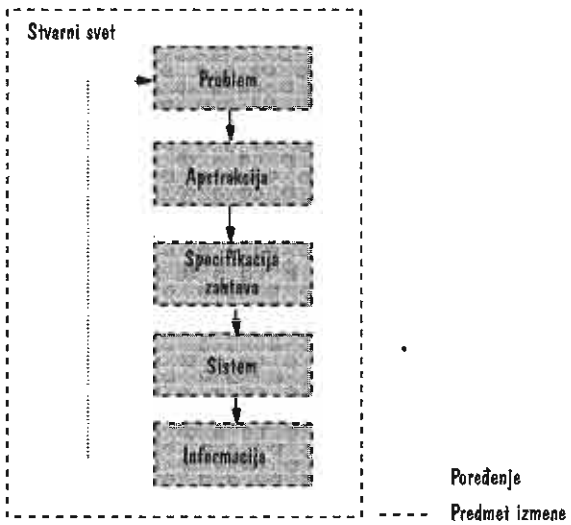
Kod P-sistema mogu da se menjaju mnogi elementi. Kada se izlazna informacija poredi sa stvarnim problemom, može da dođe ili do izmene apstrakcije, ili do drugačije formulacije zahteva, što na odgovarajući način može da utiče na implementaciju. Sistem koji je rezultat izmena ne može se smatrati novim rešenjem novog problema. To je pre modifikacija starog rešenja da bi se ono bolje uskladilo sa postojećim problemom.

E-sistemi. Prilikom razmatranja S i P-sistema, stvarno okruženje ostaje stabilno. Međutim, treća klasa sistema uzima u obzir promenljivu prirodu samog realnog sveta. **E-sistem** je onaj koji je ugrađen u realni svet i menja se u skladu sa njim. Rešenje je zasnovano na modelu razmatranog apstraktnog procesa. Otuda je sistem integralni deo realnog sveta koji modeluje.

Na primer, sistem koji služi za predviđanje stepena zdravlja privrede neke zemlje zasnovan je na funkcionalnom modelu te privrede. Promene se dešavaju u svetu kome problem pripada. Uz to, privreda nije u potpunosti shvatljiva, tako da se model može menjati kako se naša shvatanja menjaju. Na kraju se samo rešenje menja na osnovu izmena apstraktnog modela.

Na slici 11.3 ilustrovana je promenljivost E-sistema i njegova zavisnost od konteksta realnog sveta. Dok je verovatnoća izmena kod S-sistema mala, a P-sistemi su predmet postepenih izmena to su kod E-sistema izmene stalno prisutne. Štaviše, uspeh E-sistema u potpunosti zavisi od ocene koju o performansi sistema daje kupac. Kako se problem koji E-sistem rešava ne može u potpunosti specifikovati, o sistemu se može suditi isključivo na osnovu njegovog ponašanja u stvarnim radnim uslovima.

SLIKA 11.3 E-sistem



Ove kategorije ukazuju na elemente sistema koji su predmet izmena. Što je veći broj elemenata koji se menjaju, to je potreba za održavanjem sistema verovatnija. S obzirom na to da je priroda problema koji generiše neki E-sistem promenljiva, rešenje E-sistema će se verovatno stalno unapređivati.

Izmene u toku životnog ciklusa sistema

Analizom sistema u svetlu njegove kategorije (S, P ili E) uočavaju se mesta na kojima u toku razvoja može doći do izmena i na osnovu toga se određuje način na koji te izmene utiču na sistem. Po svojoj prirodi, problem koji se predstavlja u formi S-sistema potpuno je definisan i nije verovatno da će se menjati. Sličan problem se može rešiti izmenom S-sistema, ali je rezultat potpuno nov problem sa novim rešenjem. Ako S-sistem ne funkcioniše na prihvatljiv način, razlog je obično to što se rešava pogrešan problem. Odgovarajuća mera koju treba preduzeti nije izmena starog sistema, već redefinisanje problema i izrada nove specifikacije, a zatim razvoj potpuno novog rešenje.

P-sistem predstavlja aproksimativno rešenje problema, koje se menja ako se identifikuju neusklađenosti ili propusti. U stvari, poređenjem informacija koje generiše sistem sa stvarnom situacijom koja se modeluje menjamo P-sistem da bismo obezbedili njegovu ekonomičnost i efikasan rad.

U P-sistemu model aproksimira rešenje postavljenog problema, pa modifikacije mogu da se pojave u svim fazama razvoja. Prevashodno može da se promeni apstrakcija. Drugim rečima, možemo da promenimo opis problema, a zatim da na odgovarajući način menjamo specifikaciju zahteva. Nakon toga menjamo konstrukciju sistema, ponavljajući implementaciju i testiranje u cilju implementacije izmena. Najzad, menjamo sistemsku i programsku dokumentaciju aproksimativnog sistema što može da zahteva novu obuku.

E-sistemi koriste apstrakcije i modele za aproksimaciju situacije, pa su zato, u najmanju ruku, podložni izmenama koje mogu da dožive P-sistemi, ali je priroda e-sistema nestalnija jer se i sam problem može menjati. Budući da su ugrađeni u aktivnosti koje su veoma promenljive, E-sistemi zahtevaju da se svojstvo promenljivosti ugradi u sam sistem i time on učini prilagodljivim.

Primeri izmena u svim vrstama sistema navedeni su u tabeli 11.1. Na primer, izmena zahteva u fazi analize zahteva dovodi do izmena u specifikaciji. Izmena tehničkog dizajna može da zahteva izmenu u konstrukciji sistema, a možda i u prvobitnim zahtevima. Otuda izmena u bilo kojoj fazi razvoja može da utiče na rezultate, kako prethodnih, tako i narednih faza.

Predloženi principi softverskog inženjerstva koji se odnose na razvoj sistema olakšavaju menjanje sistema u fazi održavanja. Na primer, modularizacija konstrukcije i komponenti koda zajedno sa unakrsnim referenciranjem komponenti i zahteva, obezbeđuje da se lako prati kako izmena u zahtevima utiču na komponente koje treba menjati i na testove koje treba nakon izmene ponoviti. Slično tome, ako dođe do otkaza, moguće je identifikovati komponentu koja sadrži uzroke i zatim izvršiti izmene na svim nivoima (konstrukcije, koda i testa), a ne samo na nivou koda. Principi softverskog inženjerstva doprinose ne samo dobroj konstrukciji i ispravnom kodu, nego i sposobnosti da se brzo i lako naprave izmene.

РЕПУБЛИКА СРПСКА
Природно-математички
факултет Бања Лука
Ф И З И К А

TABELA 11.1 Primeri izmena u fazi razvoja softvera

Uzročnik početne izmene	Izmena je posledica
Analiza zahteva	Specifikacije zahteva
Konstrukcija sistema	Specifikacije arhitektonskog dizajna
	Specifikacije tehničkog dizajna
Konstrukcija programa	Specifikacije dizajna programa
Implementacija programa	Programskog kôda
	Programske dokumentacije
Jedinično testiranje	Planiranja testova
	Scenarija testova
Testiranje sistema	Planiranja testova
	Scenarija testova
Isporuka sistema	Korisničke dokumentacije
	Pomagala za obuku
	Operaterska dokumentacija
	Sistemskog vodiča
	Programerskog vodiča
	Časova obuke

Životni vek sistema

S obzirom na to da softverski inženjeri pokušavaju da isporuče proizvode koje je moguće održavati, osnovno pitanje koje se postavlja jeste da li je moguće iz prve izgraditi ispravan sistem. Drugim rečima, ako koristimo komponente veoma visokog stepena kohezije i niskim stepenom sprezanja, ako je dokumentacija kompletna i ažurna i ako je ceo sistem unakrsno referenciran, da li će održavanje biti uopšte potrebno? Nažalost, odgovor je potvrdan. Razlozi leže u prirodi samih sistema. Kao što smo videli, nema načina da se garantuje da P-sistemi ili E-sistemi neće zahtevati izmene. U stvari, moramo pretpostaviti da će se oni menjati i izgraditi ih tako da su njihove naknadne izmene jednostavne.

Naše sledeće pitanje mora da bude: Koja količina izmena može da se očekuje? Odgovor opet zavisi od prirode sistema. S-sistemi će se malo ili nimalo menjati. P-sistemi će se menjati više, a E-sistemi će se verovatno stalno menjati. Zato mnogi softverski inženjeri više vole da fazu održavanja u razvoju zovu **fazom evolucije**. Govorimo o **nasleđenim sistemima**, izgrađenim ranije, kada su naše potrebe i okruženje bili drugačiji. Kao što ćemo videti, nasleđene sisteme moramo procenjivati i pomagatu im da evoluiraju kada tehnologije i poslovni razlozi zahtevaju evoluciju. U nekom trenutku možemo odlučiti da nasleđeni sistem u celosti zamenimo novim ili da ga jednostavno povučemo iz upotrebe zato što se više ne koristi.

Vreme za razvoj prema vremenu za održavanje. Da bismo stekli predstavu o tome koliko dugačku fazu evolucije treba da očekujemo, možemo pogledati vremena razvoja i održavanja drugih projekata. Prema Parikhu i Zvegintzovu (1983), tipičan razvojni projekat uzima od jedne do dve godine, ali zahteva vreme održavanja od još pet do šest godina. Izraženo u uloženom radu, više od polovine programerskih resursa se potroši na održavanje. U pregledu koji su dali Fjeldstad i Hamlen (1979), izveštava se o sličnom odnosu: 25 organizacija za obradu podataka su izjavile da su u proseku uložile 39% rada u razvoj, a 61% u održavanje (ispravljanje grešaka, modifikacije i podrška korisnika). U skorašnjim pregledima se prikazuju slični rezultati, a mnogi projektanti primenjuju pravilo 80-20: 20% rada pripada razvoju, a 80% održavanju.

Evolucija sistema prema prestanku rada. Kada sistem zahteva značajne i stalne izmene, moramo da odlučimo da li je bolje da odbacimo stari sistem i zamenimo ga novim. Da bismo to utvrdili, moramo da postavimo nekoliko pitanja:

- Da li su troškovi održavanja previsoki?
- Da li je pouzdanost sistema neprihvatljiva?
- Da li sistem više ne može da se prilagodi daljim izmenama u razumnom vremenu?
- Da li je performansa sistema i dalje izvan propisanih ograničenja?
- Da li su sistemske funkcije nedovoljno korisne?
- Da li drugi sistemi mogu da urade isti posao bolje, brže ili jeftinije?
- Da li su troškovi održavanja hardvera dovoljno visoki da opravdaju njegovu zamenu jeftinijim, novijim hardverom?

Pozitivan odgovor na sva ili samo na neka od ovih pitanja može da znači da je došao trenutak da se razmotri zamena starog sistema novim. Suma svih troškova vezanih za razvoj i održavanje sistema, od njegovog nastanka do prestanka korišćenja, naziva se **cena životnog ciklusa**. Odluku da li da održavamo, ponovo izgradimo ili zamenimo, često zasnivamo na poređenju troškova životnog ciklusa starih, revidiranih ili novih sistema.

Zakoni evolucije softvera. Odluke koje se tiču održavanja podržane su razumevanjem onoga što se dešava sistemima tokom vremena. Od interesa su izmene veličine, složenost, potrebni resursi i lakoća održavanja. O evolucionim trendovima možemo mnogo da naučimo testiranjem izmena u sklopu velikih sistema. U podsetniku 11.1 opisana je evolucija velikog sistema u kompaniji Bell Atlantic.

U toku svoje karijere, Lehman je posmatrao ponašanje sistema tokom njihove evolucije. Svoja otkrića je sumirao u pet zakona evolucije programa (Lehman 1980):

1. *Stalna izmena.* Program koji se koristi stalno doživljava izmene, ili postaje progresivno sve manje upotrebljiv. Izmene ili proces propadanja se nastavljaju dok ne postane isplativije da se sistem zameni novom verzijom.
2. *Povećavanje složenosti.* Kako se program evolutivno menja, njegova struktura se remeti. To se ogleda u povećanju složenosti, sem ako se ne uloži rad u njeno održavanje ili smanjivanje.
3. *Fundamentalni zakon evolucije programa.* Evolucija programa je podložna dinamici koja proces programiranja, a otud i mere globalnog projekta i sistemskih atributa, čini autonomno regulativnim, sa statistički definisanim trendovima i invarijantama.
4. *Konzervacija organizacione stabilnosti (nepromenljiva brzina rada).* Tokom aktivnog života programa, globalna brzina aktivnosti u projektu programiranja je statistički nepromenljiva.
5. *Konzervacija bliskosti (uočena složenost).* Tokom aktivnog života programa, sadržaj (izmene, dodaci, izbacivanja) uzastopnih izdanja programa koji evoluira je statistički nepromenljiv.

PODSETNIK 11.1 BELL ATLANTIC MENJA TRI SISTEMA JEDNIM KOJI EVOLUIRA

Godine 1993. kompanija Bell Atlantic (sada poznata kao Verizon) uvela je SaleService Negotiation System (SSNS), kao zamenu za tri nasleđena sistema koji su podržavali operatere da preuzimaju porudžbine za nove telefonske usluge. Početni ciljevi sistema bili su minimizacija grešaka i smanjenje vremena koje predstavnici davaoca usluga provode u telefonskom razgovoru sa kupcima. Ali kako je komercijala koristila sistem, u rukovodstvu kompanije su uočili njegov veliki potencijal za promociju Bell Atlanticovih proizvoda koji bi mogli da zadovolje potrebe kupaca. Ciljevi su se promenili jer sistem više nije trebalo da služi za preuzimanje narudžbina, nego za prodaju zasnovanu na potrebama.

Proces naručivanja u sklopu SSNS-a veoma je sličan intervjuisanju od strane sistema. Kako kupac unosi sve više informacija o sebi, SSNS mu skreće pažnju na relevantne proizvode. Otuda je nužno da se SSNS na adekvatan način osnažuje kako Bell Atlantic proširuje svoju proizvodnu i uslužnu ponudu. SSNS je već bio proširen rukovanjem informacijama vezanim za izdavanje računa, proveru adrese i kreditne sposobnosti uz oslonac na udaljene baze podataka, automatskim informisanjem kupaca o potvrdi usluge i snabdevanjem službenika informacijama i pitanjima vezanim za usluge.

U sistemu su takođe arhaične komande zamenjene jednostavnim engleskim jezikom, a mnogo toga što se ranije nalazilo u priručniku od 20 tomova, sada je onlajn. Izvršeno je prilagođavanje sistema u nekim državama, zbog različitih propisa koji uređuju pružanje telefonskih usluga. Nadležne ustanove zahtevale su od Bell Atlantica da obelodani specifične informacije o proizvodima na početku intervjua, pa je sistem, gde je to bilo potrebno, tako i podešen.

Prvobitno napisan na jezicima C i C++, sistem je modifikovan oslanjajući se na programski jezik Java krajem 1990-ih godina, da bi se obezbedila intranet verzija dostupna mobilnim trgovačkim predstavnicima. Kako su pravilima, proizvodima, tehnologijama i poslovanju potrebne izmene, SSNS mora da evoluira sa njima (Field 1997a).

U prvom zakonu se kaže da veliki sistemi nisu nikada dovršeni; oni neprekidno evoluiraju. Sistemi se šire sa dodavanjem novih svojstava, primenom više ograničenja, sadejstvom sa drugim sistemima, podrškom za više korisnika itd. Oni se takođe menjaju zbog izmena u njihovim okruženjima: prenose se na druge platforme, ili preprogramiraju na drugim jezicima.

Drugi zakon nam kaže da kako evoluiraju, veliki sistemi postaju sve složeniji, sem ako ne preduzmemo mere da tu složenost smanjimo. Često se taj porast složenosti dešava zbog neophodnosti brzih popravki koje za cilj imaju otklanjanje problema. Jednostavno nemamo vremena da u kodu održavamo eleganciju dizajna ili konsistentnost pristupa.

Prema trećem zakonu, softverski sistemi pokazuju pravilnosti u ponašanju i trendovima koje možemo da merimo i predvidamo. Mnogi istraživači u oblasti softverskog inženjerstva posvetili su se pronalaženju tih „univerzalnih istina" razvoja i održavanja softvera, slično kao što fizičari traže *veliku objedinjavajuću teoriju*.

U četvrtom zakonu se kaže da ne postoje neobuzdane i velike fluktuacije u svojstvima organizacije kao što je na primer, produktivnost. Lehman ukazuje na Brooksova zapaža-

nja koja podržavaju ovaj zakon (Brooks 1975). To znači da, u nekom trenutku, resursi i izlaz dostižu optimalni nivo i da dodavanje više resursa ne menja značajno izlaz. Slično tome, peti zakon kaže da posle izvesnog vremena, efekti sledećih izdanja neznatno doprinose opštoj funkcionalnosti sistema.

11.2 PRIRODA ODRŽAVANJA

Kada razvijamo sisteme, u osnovi smo usredsređeni na izradu koda koji implementira zahteve i radi ispravno. U svakoj fazi razvoja, naš tim se stalno oslanja na rad obavljen u prethodnim fazama. Komponente dizajna se povezuju sa specifikacijom zahteva, komponente koda se unakrsno referenciraju i pregledaju radi usaglašavanja sa dizajnom, a testovi se zasnivaju na proveri da li funkcije i ograničenja deluju u skladu sa zahtevima i dizajnom. Dakle, razvoj obuhvata i osvrtanje unazad, na pažljiv i kontrolisan način.

Održavanje je drugačije. Ljudi iz održavanja vode računa o proizvodima procesa razvoja, ali i oa sadašnjosti uspostavljajući spregu sa korisnicima i operaterima da bi utvrdili kako su oni zadovoljni načinom rada sistema. Takođe moraju gledati i unapred da bi predvideli stvari koje mogu da krenu loše, razmotrili izmene funkcionalnosti koje su posledica izmena u poslovanju i sagledali izmene sistema koje su posledica izmene hardvera, softvera ili interfejsa. Dakle, održavanje ima širi opseg, sa više elemenata koje je potrebno pratiti i kontrolisati. Ispitajmo aktivnosti koje su neophodne za ravnomeran rad sistema i identifikujemo ko te aktivnosti treba da sprovodi.

Aktivnosti i uloge u održavanju

Aktivnosti u održavanju su slične aktivnostima razvoja: analiza zahteva, evaluacija dizajna sistema i programa, pisanje i revizija programskog koda, testiranje izmena i ažuriranje dokumentacije. Otuda ljudi koji se bave održavanjem - analitičari, programeri i projektanti - imaju slične uloge. Međutim, zbog toga što izmene često zahtevaju detaljno poznavanje strukture i sadržaja programskog koda programeri imaju mnogo značajniju ulogu u procesu održavanja nego što su je imali u procesu razvoja.

Održavanje se istovremeno usredsređuje na četiri glavna aspekta evolucije sistema:

1. održavanje kontrole nad svakodnevnim funkcijama sistema;
2. održavanje kontrole nad izmenama sistema;
3. usavršavanje postojećih prihvatljivih funkcija;
4. sprečavanje da se performansa sistema degradira do neprihvatljivog nivoa.

Korektivno održavanje. Radi kontrole svakodnevnog funkcionisanja sistema, tim za održavanje reaguje na probleme koji nastaju usled grešaka. Rešavanje tih problema je poznato kao **korektivno održavanje**. Tim za korektivno održavanje pronalazi uzrok otkaza i vrši korekcije i izmene u zahtevima, konstrukciji, kodu, testovima i dokumentaciji, već prema potrebi. Početna popravka je često privremena: nešto što održava sistem u radu, ali ne predstavlja najbolje rešenje. Trajne izmene mogu kasnije da se implementiraju, da bi rešile opštije probleme koji su posledica konstrukcije ili koda.

Na primer, korisnik može da pokaže uzorak izveštaja sa suviše mnogo odštampanih redova na stranici. Programeri utvrđuju da je problem rezultat neispravnosti u konstrukciji upravljačkog programa za štampač. Tim za korektivn održavanje postupa po hitnoj intervenciji i pokazuje korisniku kako da ponovo postavi broj redova po stranici podešavanjem parametra u meniju za izveštaj pre štampanja. Na kraju tim redizajnira, ponovo kodira i testira upravljački program štampača, da bi on radio ispravno bez intervencije korisnika.

Adaptivno održavanje. Izmena uvedena u jednom delu sistema ponekad zahteva izmene u njegovim drugim delovima. **Adaptivno održavanje** je implementacija tih sekundarnih izmena. Na primer, pretpostavimo da se postojeći sistem za upravljanje bazom podataka, kao deo većeg hardverskog i softverskog sistema, nadograđuje novom verzijom. U tom procesu, programeri otkrivaju da rutine za pristup disku zahtevaju dodatni parametar. Adaptivne izmene učinjene da bi se uveo dodatni parametar ne ispravljaju greške; one samo dozvoljavaju sistemu da evoluira i da se prilagođava tokom te evolucije.

Slično tome, pretpostavimo da pojačavamo kompajler dodavanjem programa za otkrivanje i ispravljanje grešaka. Potrebno je promeniti menije, ikone ili funkcijske tastere da bismo dozvolili korisnicima da biraju opciju programa za otkrivanje i otklanjanje grešaka.

Adaptivno održavanje može da se primeni nakon izmena hardvera ili okruženja. Ako se sistem, inicijalno projektovan za rad u suvom, stabilnom okruženju, premesti u rezervoar ili podmornicu, neophodno ga je prilagoditi na kretanje, magnetne uticaje i vlažnost.

Održavanje radi usavršavanja. Kada održavamo sistem, ispitujemo dokumente, konstrukciju, kôd i testove, tražeći moguća poboljšanja. Na primer, kada se dodaju nove funkcije, prvobitna, originalna konstrukcija zasnovana na tabelama odlučivanja može da postane konfuzna i teško se prati. Preprojektovanje na pristup na osnovu pravila može da poboljša proces održavanja u budućnosti i pojednostavi dodavanje novih funkcija. **Održavanje radi usavršavanja** obuhvata izmene koje poboljšavaju neki aspekt sistema, čak i kada one nisu posledica grešaka. Izmene u dokumentaciji radi pojašnjenja nekih delova, izmene u skupu testova da bi se poboljšali stepen obuhvatnosti testova i izmene koda i konstrukcije radi bolje čitljivosti su primeri održavanja radi usavršavanja.

Preventivno održavanje. Slično održavanju radi usavršavanja, **preventivno održavanje** obuhvata izmenu nekog aspekta sistema da bi se sprečili otkazi. Ono može da uključi dodavanje provere tipova, poboljšano rukovanje greškama, ili proširenje iskaza *case* iskazom „catch-all", da bismo obezbedili da sistem opsluži sve mogućnost. Preventivno održavanje obično nastaje kada programer ili analitičar koda pronađe stvarnu ili moguću grešku koja još nije dovela do otkaza i preduzme akciju da ispravi grešku pre nego što ona prouzrokuje štetu.

Ko sprovodi održavanje. Tim koji razvija sistem nije uvek onaj koji ga i održava nakon njegovog uvođenja u eksploataciju. Često se angažuje poseban tim za održavanje, čija je namena obezbeđenje ispravnog rada sistema. Razvojni tim je upoznat sa kodom, konstrukcijom i filozofijom na kojoj se on zasniva, kao i sa ključnim funkcijama sistema.

Ako su projektanti svesni da grade nešto što će oni održavati, tada će izgraditi sistem na način koji olakšava održavanje.

Međutim, ponekad projektanti imaju toliko poverenja u svoje razumevanje sistema da su skloni da ne ažuriraju dokumentaciju. Nebriga u pisanju i reviziji dokumentacije može da rezultira većim angažovanjem resursa i ljudi za rešavanje problema. Ta situacija dovodi do dugog intervala između trenutka kada se problem pojavi i trenutka njegovog rešavanja. Mnogi kupci neće tolerisati takvo kašnjenje.

Često se posebna grupa analitičara, programera i projektanata (ponekad uključujući jednog ili dva člana razvojnog tima) promoviše u tim za održavanje. Svež, novi tim može da bude objektivniji od prvobitnih projektanata. Poseban tim može da otkrije da je lakše razlikovati šta bi sistem trebalo da radi od toga kako to sistem radi. Ako znaju da će drugi raditi na osnovu njihove dokumentacije, projektanti su skloni da budu pažljiviji kada je u pitanju dokumentovanje i primena standarda programiranja.

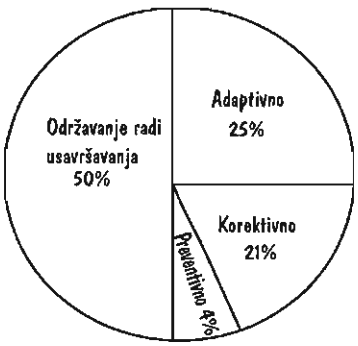
Odgovornosti tima. Održavanje sistema uključuje sve članove tima. Korisnici, operateri ili predstavnici kupca obično pristupaju timu za održavanje sa nekim komentarom ili problemom. Analitičari ili programeri određuju na koje delove koda se problem odnosi, njegov uticaj na dizajn i verovatne resurse (uključujući vreme i rad) neophodne za realizaciju izmene. Tim se bavi mnogim aktivnostima, kao što su:

1. razumevanje sistema;
2. pronalaženje informacija u dokumentaciji sistema;
3. održavanje ažurnosti dokumentacije sistema;
4. proširivanje postojećih funkcija da bi se prihvatili novi ili promenjeni zahtevi;
5. dodavanje novih funkcija sistemu;
6. pronalaženje uzroka otkaza ili problema u sistemu;
7. pronalaženje i ispravljanje grešaka;
8. odgovaranje na pitanja o načinu na koji sistem radi;
9. restrukturisanje konstrukcije i komponenti koda;
10. ponovo pisanje komponenti dizajna i koda;
11. brisanje komponenti konstrukcije koda koje više nisu korisne;
12. upravljanje izmenama sistema kada se one prave.

Pored toga, članovi tima za održavanje rade sa korisnicima, operaterima i kupcima. Prvo, oni pokušavaju da shvate problem iskazan jezikom korisnika. Nakon toga se problem transformiše u zahtev za izmenu. Zahtev za izmenu sadrži opis sadašnjeg načina rada, načina na koji korisnik želi da sistem radi i koje su izmene neophodne za tu izmenu. Kada se konstrukcija ili kôd jednom modifikuju i testiraju, tim za održavanje ponovo obučava korisnika, ako je to potrebno. Dakle, održavanje uključuje interakciju sa ljudima, kao i sa softverom i hardverom.

Raspodela rada na održavanju. Postoje različiti izveštaji o tome na koje vrste održavanja odlazi najviše serviserskog vremena. Lientz i Swanson (1981) su pratili rad izvršilaca u 487 organizacija za obradu podataka i otkrili raspodelu poslova po vrstama održavanja prikazanu na slici 11.4. Najveći rad ulaže se u adaptivno održavanje i održa-

vanje radi usavršavanja. Slične podele su pokazala i druga istraživanja istog tipa. Raspodela vremena za datu organizaciju zavisi od više činilaca, uključujući i da li je sistem tipa S, P ili E i koliko brzo se poslovanje menja.



SLIKA 11.4 Raspodela rada na održavanju

11.3 PROBLEMI ODRŽAVANJA

Održavanje sistema je teško. Imajući u vidu da je sistem već u eksploataciji, tim za održavanje uravnotežuje potrebu za izmenom sa potrebom da sistem ostane dostupan korisnicima. Na primer, nadogradnja sistema može da zahteva da on bude nedostupan korisnicima više sati. Međutim, ako je sistem kritičan sa aspekta poslovanja ili rada korisnika, može se desiti da vremenski okvir od nekoliko sati nedostupnosti jednostavno ne postoji. Na primer, sistem za održavanje života ne može da prekine vezu sa pacijentom da bi se obavilo održavanje softvera. Tim za održavanje mora da pronađe način kako da implementira izmene bez ugrožavanja korisnika.

Personalni problemi

Postoji mnogo personalnih i organizacionih razloga koji održavanje čine teškim. Osoblje mora da deluje kao posrednik između problema i njegovog rešenja, popravljajući i krojeći softver da bi obezbedilo da rešenje prati problem kako se on menja.

Ograničeno razumevanje. Pored uravnotežavanja korisničkih potreba sa potrebama softvera i hardvera, tim za održavanje ima posla i sa ograničenjima ljudskog razumevanja. Postoji granica brzine kojom neko može da proučava dokumentaciju i iz nje izvlači materijal relevantan za problem koji se rešava. Štaviše, mi često tražimo više ključeva nego što je stvarno potrebno za rešenje problema. Ako tome dodamo svakodnevne kancelarijske „zabave", dobili smo recept za ograničenu produktivnost.

Parikh i Zvegintzov (1983) izveštavaju da je 47% rada u održavanju softvera posvećeno razumevanju softvera koji treba da se modifikuje. Ova velika cifra je razumljiva kada

uzmemo u obzir interfejse koje treba proveriti kad god se menja komponenta. Na primer, ako sistem ima m komponenti i treba da promenim k od njih, postoji

$$k*(m-k)+k*(k-1)/2$$

interfejsa koji treba da se ocene zbog uticaja i ispravnosti (Gerlich i Denskat 1994). Dakle, čak i izmena samo jednog reda u sistemu može da zahteva stotine testova, da bismo se uverili da izmena nema direktnog ili indirektnog efekta na neki drugi deo sistema.

Korisničko razumevanje takođe predstavlja problem. Lientz i Swanson (1981) su pronašli da više od polovine problema programera koji se bave održavanjem nastaje kao posledica nedovoljne veštine ili stepena razumevanja kod korisnika. Na primer, ako korisnici ne razumeju kako sistem funkcioniše, oni mogu da daju nepotpune ili obmanjujuće podatke kada izveštavaju o efektima problema.

Ovi rezultati ilustruju značaj jasne i potpune dokumentacije i obuke. Timu za održavanje su takođe potrebne dobre „ljudske veštine". Kao što smo videli u poglavlju 2, postoji više različitih stilova rada. Tim za održavanje mora da razume kako ljudi sa različitim stilovima misle i rade, a članovi tima moraju da budu fleksibilni u komunikaciji.

Prioriteti menadžmenta. Tim za održavanje odmerava želje rukovodećih ljudi na strani kupca prema potrebama sistema. Prioriteti rukovodilaca često prevazilaze tehničke prioritete; menadžeri ponekad vide održavanje i poboljšanje kao važnije od izrade novih aplikacija. Drugim rečima, kompanije moraju ponekad da se usredsrede na poslovanje kao što je uobičajeno, umesto da istražuju nove alternative. Ali, dok menadžment ohrabruje servisere da popravljaju stari sistem, korisnici vape za novim funkcijama ili novim sistemom. Slično tome, žurba da proizvod dođe na tržište može da ohrabri projektante ili servisere da implementiraju brzu, neelegantnu, nedovoljno testiranu izmenu, a ne da odvoje vreme za sprovođenje dobre prakse softverskog inženjerstva. Rezultat je zakrpljeni proizvod koji se kasnije teško razume i popravlja.

Moral. Studije Lientza i Swansona (1981) pokazuju da je 11,9% problema u toku održavanja rezultat niskog morala i produktivnosti. Glavni razlog za nizak moral je drugorazredni status koji se često dodeljuje timu za održavanje. Programeri ponekad misle da je potrebno više veštine da bi se projektovao i razvio sistem, nego da se on održava u radu. Međutim, kao što smo videli, programeri iz održavanja bave se problemima koje projektanti nikada ne vide. Serviseri su obučeni ne samo za pisanje koda, nego i za rad sa korisnicima, predviđanje izmena i traganje. Potrebne su velika veština i istrajnost da bi se problem pratio do njegovog uzroka, da se razume unutrašnji rad velikog sistema i da se modifikuju njegova struktura, kôd i dokumentacija.

Neke grupe rotiraju programere između više projekata održavanja i razvoja, da bi im dali priliku da rade različite stvari. Ta rotacija pomaže da se izbegne „žigosanje" programiranja za potrebe održavanja. Međutim, od programera se često traži da istovremeno rade na više projekata. Zahtevi za vremenom programera rezultuju sukobom prioriteta. U toku održavanja, 8% problema su posledica činjenice da se programeri razvlače na više strana odjednom i da zato nisu u stanju da se dovoljno dugo usredsrede na jedan problem da bi ga rešili.

Tehnički problemi

Tehnički problemi takođe utiču na produktivnost održavanja. Ponekad su oni nasledstvo onoga što su projektanti i serviseri radili ranije. U drugim prilikama, oni su rezultat posebnih platformi ili procesa koji su usvojeni za implementaciju.

Artefakti i paradigme. Ako logika konstrukcije nije očigledna, tim ne može lako da odredi da li konstrukcija može da opsluži predložene izmene. Pogrešna ili nefleksibilna konstrukcija može da zahteva dodatno vreme za razumevanje, menjanje i testiranje. Na primer, projektanti su mogli da uključe komponentu za ulaz i izlaz koja opslužuje jedino traku; potrebno je učiniti velike izmene da se traka zameni diskom, zato što disk nije ograničen sekvencijalnim pristupom kao traka. Slično tome, projektanti su mogli da ne predvide izmene; veličine polja i tabele mogu da budu fiksne, što ih čini teškim za modifikaciju. „Problem 2000. godine", gde je većina projektanata predstavljala godinu sa samo dve cifre, dobar je primer kako jednostavne, ali ograničene odluke u projektu mogu da imaju veliki uticaj na proces održavanja.

I održavanje objektno orijentisanih programa može da bude problematično, zato što projekat često obuhvata komponente koje su jako spregnute složenim šemama nasleđivanja. Postepene izmene moraju da se prave sa velikom pažnjom, zato što modifikacije mogu da rezultiraju dugim lancima prikrivenih klasa ili klasa koje redefinišu objekte na konfliktan način. U podsetniku 11.2 opisani su posebni projektantski kompromisi koji moraju da se prave kada se održavaju objektno orijentisani sistemi.

Uopšte, na neodgovarajuće specifikovane projekte i programe slabog kvaliteta odlazi skoro 10% rada na održavanju. Slična količina rada troši se zbog hardverskih zahteva: dobijanje odgovarajućeg prostora za skladištenje i vremena obrade. Kao student, vi razumete frustraciju kada treba da rešite problem, ali nemate pristup radnoj stanici, ili kada morate uporno da birate broj jer nikako ne možete da dobijete udaljeni pristup. Problemi takođe nastaju kada su hardver, softver ili podaci nepouzdani.

Teškoće sa testiranjem. Testiranje može da bude problem ako je teško pronaći vreme za njegovo izvođenje. Na primer, sistem za rezervaciju avionskih karata mora da bude stalno na raspolaganju. Teško je ubediti korisnike da ne mogu da koriste sistem zato što se on testira sat ili dva.Sisteme koji izvršavaju kritične funkcije, kao što je kontrola vazdušnog saobraćaja ili praćenje stanja pacijenata, ponekad je nemoguće dovesti u neaktivno stanje radi testiranja. U takvim slučajevima, testovi se obično izvršavaju na rezervnim sistemima; testirane izmene se zatim prenose na proizvodni sistem.

Pored problema sa vremenom kada je sistem raspoloživ, može se dogoditi i da nema dobrih ili odgovarajućih podataka za testiranje izmena koje se treba učiniti. Na primer, ako treba da se izmeni sistem za predviđanje zemljotresa da bi prihvatio signale sa senzorskog uređaja koji jee tek razvija, podaci za testiranje moraju da se simuliraju. Generisanje tačnih podataka za testiranje tada nije baš lako jer naučnici još uvek ne znaju tačno kako dolazi do zemljotresa.

Što je najvažnije, testerima nije uvek lako da predvide kakav će biti uticaj izmena na konstrukciju ili kod i da se za njih pripreme. Takva nepredvidljivost postoji posebno onda kada različiti članovi tima za održavanje rade na različitim problemima u isto vreme. Ako Pat menja komponentu da bi rešila problem prekoračenja opsega podataka, dok Denis menja istu komponentu da bi rešio problem interfejsa, kombinacija tih izmena može da prouzrokuje novu grešku.

PODSETNIK 11.2 KORISTI I NEDOSTACI ODRŽAVANJA OBJEKTNO ORIJENTISANIH SISTEMA

Wilde, Matthews i Huitt (1993) su istraživali razlike između održavanja objektno orijentisanih i proceduralnih sistema. Oni su uočili prednosti objektne orijentacije:

- izmene usled održavanja na jednoj klasi ne utiču na ostatak programa;
- serviseri mogu lako ponovo da upotrebe objekte, pišući samo malu količinu novog koda.

Međutim, postoji više nedostataka:

- objektno orijentisane tehnike otežavaju razumevanje programa zbog mnoštva delova koji čine program; teško je razaznati prvobitnu nameru projektanta zbog delokalizacije planova programa sa distribuirani na više nesusednih programskih segmenata;
- iz istog razloga, više delova otežava razumevanje ponašanje celog sistema;
- nasleđivanje može da stvori zavisnosti koje je teško pratiti;
- dinamičko povezivanje onemogućava da se odredi koja od više metoda će se izvršiti, pa serviseri moraju da razmatraju sve mogućnosti.
- skrivanjem detalja o strukturi podataka, funkcija programa je često distribuirana na više klasa; tada je teško otkriti i dešifrovati klase su u međusobnoj interakciji.

Potreba za kompromisom

Tim za održavanje se uvek bavi uravnotežavanjem različitih ciljeva. Kao što smo videli, sukob može da nastane između raspoloživosti sistema za korisnike i implementacije izmena, ispravki i poboljšanja. Imajući u vidu da se otkazi pojavljuju u nepredvidljivim trenucima, osoblje za održavanje je stalno svesno tog sukoba.

Za profesionalce u računarstvu, jedan drugi sukob nastaje kad god je potrebna izmena. Principi softverskog inženjerstva su u suprotnosti sa ekspeditivnošću i troškovima. Problem često može da se reši na jedan od dva načina: brzim, ali neelegantnim rešenjem koje radi, ali ne odgovara dizajnu sistema ili strategiji kodiranja, ili zahtevnijim, ali elegantnijim načinom koji je konsistentan sa vodećim principima korišćenim za projektovanje ostatka sistema. Kao što smo ranije napomenuli, programeri mogu biti prinuđeni da prave kompromis između elegancije i principa projektovanja, zato što je izmena trenutno potrebna.

Kada se naprave takvi kompromisi, više sa njima povezanih događaja može da otežа održavanje u budućnosti. Prvo, žalbu serviserima obično podnose korisnik ili operater. Oni verovatno ne razumeju problem u kontekstu konstrukcije i koda, nego samo u kontekstu svakodnevnog rada. Drugo, rešavanje takvog problema se svodi samo na ispravku date greške. Ne dolazi do revizije sistemske ili programske konstrukcije da bi sistem u celini postao razumljiviji ili da bi izmena bila konsistentna sa ostalim komponentama sistema. Zbog ta dva činioca tim za održavanje ima ograničeni cilj - brzu popravku greške. Tim je primoran da usmeri svoje resurse na problem koji nedovoljno razume.

Tim mora da razreši još jedan sukob. Kada se razvija sistema za rešavanje početnog problema, njegovi projektanti ponekad njime pokušavaju da reše i druge slične probleme ne menjajući konstrukciju i kod. Takvi sistemi često rade sporo, zato što je njihov kôd opšte namene i mora da pregleda veliki broj slučajeva ili mogućnosti. Da bi se poboljšala performansa, treba praviti sistem specijalne namene, odnosno opštost se žrtvuje zarad brzine. Komponente specijalne namene su često manje jer ne moraju da razmatraju svaku mogućnost. Tako dobijeni sistem može lako da se menja, po ceni vremena koje je potrebno da bi se modifikovale ili poboljšale konstrukcije sistema ili programa. Tim mora da odmeri opštost prema brzini kada odlučuje kako i zašto pravi modifikaciju ili ispravku.

Ostali činioci koji mogu da utiču na pristup tima za održavanje su:

- vrsta otkaza;
- kritičnost ili ozbiljnost otkaza;
- težina potrebnih izmena;
- opseg potrebnih izmena;
- složenost komponenti koje se menjaju;
- broj fizičkih lokacija na kojima moraju da se izvrše izmene.

Svi činioci koje smo ovde opisali ukazuju na to da osoblje za održavanje radi dvostruki posao. Prvo, tim shvata dizajn sistema, kôd i filozofiju testiranja i strukture. Drugo, on razvija filozofiju održavanja i strukturisanja sistema. Uravnotežujući kratkoročne i dugoročne ciljeve, tim odlučuje kada da žrtvuje kvalitet radi brzine.

Troškovi održavanja

Svi problemi sa održavanjem sistema doprinose visokim troškovima održavanja softvera. U 1970-im godinama, veći deo budžeta softverskih sistema trošio se na razvoj. Odnos ulaganja u razvoj prema ulaganju u održavanje promenio se u 1980-im, a različita istraživanja pokazuju da na održavanje odlazi 40% do 60% ukupnih troškova životnog ciklusa sistema (odnosno, od razvoja, preko održavanja do konačnog povlačenja ili zamene). Trenutno postojeće procene nagoveštavaju da su se troškovi održavanja povećali do 80% troškova životnog ciklusa sistema u 2000-im godinama.

Činioci koji utiču na uloženi rad. Pored problema o kojima smo već govorili, postoje i mnogi drugi koji utiču na rad potreban za održavanje sistema. Ti činioci su obično sledeći:

- *Vrsta aplikacije.* Sisteme za rad u realnom vremenu i one koji su veoma sinhronizovani teže je održavati od onih gde tajming nije od suštinske važnosti za ispravno funkcionisanje. Moramo da posvetimo veliku pažnju da bismo obezbedili da izmena jedne komponente ne utiče na tajming drugih. Slično tome, izmene u programima sa kruto definisanim formatima podataka mogu da zahtevaju dodatne izmene u velikom broju rutina za pristup podacima.

- *Novost u sistemu.* Kada se u sistemu implementira nova aplikacija ili nov način izvođenja opštih funkcija (kao što je slučaj u sistemu opisanom u podsetniku 11.3), serviseri ne mogu tako lako da se oslone na svoje iskustvo i shvatanje kako da pronađu i isprave greške. Potrebno je više vremena da bi se razumela konstrukcija, odredilo mesto gde je uzrok problema i testirao ispravljeni kôd. U mnogim slučajevima, moraju da se naprave dodatni podaci za testiranja, kada takvi stari podaci ne postoje.
- *Promet i raspoloživost osoblja za održavanje.* Potrebno je dosta vremena da bi se o sistemu naučilo dovoljno da bi se on razumeo i menjao. Rad na održavanju se otežava ako se članovi tima za održavanje rutinski rotiraju, napuštaju organizaciju da bi radili na drugom projektu, ili se od njih očekuje da održavaju više različitih proizvoda u isto vreme.
- *Raspon životnog veka sistema.* Sistem koji je projektovan da traje više godina verovatno zahteva više održavanja od onog čiji je životni vek kratak. Brze popravke i nedostatak pažnje u ažuriranju dokumentacije verovatno su prihvatljivi za sistem kratkog veka trajanja; takve navike mogu da budu pogubne za dugoročne projekte jer mogu da otežaju ostalim članovima tima izmene u budućnosti.

PODSETNIK 11.3 URAVNOTEŽAVANJE TEHNIČKIH I RUKOVODEĆIH POTREBA U KOMPANIJI CHASE MANHATTAN

Chase Manhattan Middle Market Banking Group drži polovinu tržišta poslovnih bankarskih usluga za male i srednje kompanije u oblasti New York/New Jersey/Connecticut. Da bi se razumelo ko su kupci, koje usluge banke koriste i kako se oni mogu podsticati da koriste više takvih usluga u budućnosti, u kompaniji je razvijen sistem za upravljanje odnosima (*Relationship Management System*, RMS) koji daje ljudima iz prodaje jedinstven interfejs ka više vrsta podataka o kupcima sa tržišta srednje kategorije, kao što su dugovanja i stanje transakcij. Zajedno sa Chase Manhattanovim starijim aplikacijama sa PC/LAN/WAN tehnologijom, RMS treba da pomogne da zastupnici kupaca dobiju više vremena da upoznaju svoje kupce.

Sistem je započeo rad 1994. godine, kao aplikacija koju je razvila Chemical Bank. Kada se Chemical spojila sa Chaseom 1996. godine, u Chaseu su odlučili da modifikuju Chemicalov sistem za upotrebu u celom bančinom sistemu. Sistem RMS se razvijao u više faza. On je spojen sa drugim Chemicalovim sistemom za globalno upravljanje, a zatim kombinovan sa više drugih sistema da bi se izbeglo udvostručavanje i povezale hardverske platforme i poslovne kancelarije. Razvijen je RMS-ov grafički korisnički interfejs zasnovan na operativnom sistemu Windows i sistem je modifikovan da bi omogućio izvršavanje tabelarnih proračuna i štampanje izveštaja korišćenjem Microsoftovih proizvoda. Onda je sistem ugrađen u Lotus Notes, kako bi izmene u podacima mogle da se izvode samo kroz aplikaciju Notes. Neki delovi RMS-a su implementirani u drugim Chase Manhattanovim bankarskim jedinicama, a sistem je stavljen na intranet da bi se omogućio daljinski pristup mobilnom osoblju banke (Field 1997b).

- *Zavisnost od promenljivog okruženja.* S-sistem obično zahteva manje održavanja od P-sistema, kome je pak potrebno manje prilagođavanja i poboljšavanja nego E-sistemu. Određenije, u sistemu koji zavisi od karakteristika hardvera na kome radi biće neophodno mnogo izmena kada se hardver modifikuje ili zameni.
- *Hardverske karakteristike.* Nepouzdane hardverske komponente ili nepouzdana podrška prodavca mogu da otežaju praćenje problema do njegovog uzroka.
- *Kvalitet konstrukcije.* Ako sistem nije sastavljen od nezavisnih, kohezivnih komponenti, pronalaženje i popravljanje uzroka problema može stvoriti takvu mešavinu izmena koje na kraju dovode do nepredvidljivih efekata u drugim komponentama.
- *Kvalitet koda.* Ako kôd nije strukturisan, ili ne implementira vodeće principe svoje arhitekture, može biti teško da se otkriju greške. Pored toga, sam jezik može da oteža pronalaženje i ispravljanje grešaka; jezici visokog nivoa često poboljšavaju mogućnost održavanja.
- *Kvalitet dokumentacije.* Nedokumentovana konstrukcija ili kôd gotovo da onemogućavaju traženje rešenja problema. Slično tome, ako se dokumentacija teško čita, ili je čak netačna, serviseri lako mogu da odlutaju s pravog puta.
- *Kvalitet testiranja.* Ako se testovi izvršavaju sa nepotpunim podacima, ili ne predviđaju posledice izmene, modifikacije i poboljšanja mogu da stvore druge probleme u sistemu.

Modelovanje rada potrebnog za održavanje. Kao što se to radi kod razvoja, želimo da izračunamo i rad neophodan za održavanje softverskog sistema. Belady i Lehman (1972) su bili među prvim istraživačima koji su pokušali da rad na održavanju obuhvate modelom predviđanja. Oni su uzeli u obzir pogoršavanje koje se javlja u velikom sistemu tokom vremena. Niz opravki i poboljšanja obično dovodi do fragmentacije sistemskih aktivnosti i, uopšte, sistem narasta sa svakim ciklusom popravke.

U velikim sistemima, serviseri moraju da postanu stručnjaci za određene aspekte sistema. To znači da svaki član tima mora da se specijalizuje u određenoj oblasti funkcionisanja sistema (performansi): na primer, za bazu podataka, korisnički interfejs ili mrežni softver. Specijalizacija servisera može da proizvede tim u kome nema opštih stručnjaka, odnosno osobe koja poseduje znanja o tome kako bi sistem trebalo da funkcioniše i kako se on odnosi prema zahtevima. Više ljudi je potrebno da bi se izašlo na kraj sa rastućim sistemom, a mašine i vreme moraju da im se stave na raspolaganje da bi ih podržali. Potrebno je i više komunikacije između članova tima, da bi se dvostruko proverilo razumevanje načina na koji rade komponente ili funkcije sistema.

U isto vreme, sistem postaje složeniji, što je rezultat dve stvari. Prvo, kada se ispravi jedna greška, popravka sama po sebi može da unese nove greške u sistem. Drugo, kada se vrše ispravke, menja se struktura sistema. Sprezanje i kohezija komponenti, kao i nasleđena struktura objektno orijentisanog sistema često se kvare zato što se radi mnogo popravki sa ograničenom ciljem rešavanja nekog posebnog problema.

Belady i Lehman su te efekte obuhvatili sledećom jednačinom:

$$M = p + K^{c-d}$$

gde je M ukupan rad na održavanju koji se ulaže za sistem, a p ukupan proizvodni rad: na analizi, proceni, projektovanju, kodiranju i testiranju; c je složenost prouzrokovana

nedostatkom strukturisanog dizajna i dokumentacije; ona se smanjuje pomoću *d*, stepena u kome je tim za održavanje upoznat sa softverom. Najzad, *K* je konstanta koja se određuje poređenjem tog modela sa odnosima rada na stvarnim projektima; ona se zove **empirijska konstanta**, zato što njena vrednost zavisi od okruženja.

Belady-Lehmanova jednačina izražava veoma značajan odnos između činilaca koji određuju rad na održavanju. Ako se sistem razvija bez principa softverskog inženjerstva, vrednost *c* će biti velika. Ako se, pored toga, on održava bez razumevanja samog softvera, vrednost *d* će biti mala. Rezultat je da se troškovi održavanja povećavaju eksponencijalno. Dakle, da bismo uštedeli na održavanju, najbolji pristup je da izgradimo sistem koristeći dobru praksu softverskog inženjerstva i damo serviserima vremena da upoznaju softver.

Savremeni modeli za predviđanje troškova održavanja koriste mnoge iste činioce koje su predložili Belady i Lehman. Na primer, COCOMO II proračunava rad na održavanju koristeći promenljivu veličine koja se izračunava na sledeći način (Boehm i dr. 1995):

Veličina = ASLOC(AA + SU + 0,4DM + 0,3CM + 0,3IM)/100

Promenljiva *ASLOC* meri broj redova koda koji treba da se adaptira, *DM* je procenat konstrukcije koji treba da se modifikuje, *CM* je procenat koda koji treba da se promeni, a *IM* je procenat spoljašnjeg koda (kao što je višekratno upotrebljivi kôd) koji treba da se integriše (ako ga ima). *SU* je mera koja predstavlja meru neophodne razumljivosti softvera, kao što je prikazano u tabeli 11.2. Na primer, ako je softver dobro strukturisan, jasan i samodokumentovan, onda su kazneni poeni za nerazumljivost samo 10%; ako je nedokumentovan, zamršeni kôd, to se kažnjava sa 50%.

TABELA 11.2 Rangiranje razumljivosti softvera po COCOMO II

	Vrlo nisko	Nisko	Nominalno	Visoko	Vrlo visoko
Struktura	Vrlo mala kohezija, jako sprezanje, zamršen kôd	Umereno mala kohezija, jako sprezanje	Razumno dobro strukturisan; neke slabe oblasti	Velika kohezija, malo sprezanje	Jaka modularnost, skrivanje informacija u podacima i upravljačkim strukturama
Jasnost aplikacije	Neusaglašenost programa i pogleda na sve aplikacije	Neka korelacija između programa i aplikacije	Umerena korelacija između programa i aplikacije	Dobra korelacija između programa i aplikacije	Jasna podudarnost između pogleda na svet programa i aplikacije
Samodokumentovanje	Nejasan kôd, nedostaje dokumentacija, nejasna je ili zastarela	Nešto komentara koda; nešto korisne dokumentacije	Umeren nivo komentara koda, zaglavlja i dokumentacije	Dobri komentari koda i zaglavlja; korisna dokumentacija; neke slabe oblasti	Samoopisiv kôd; ažurna dokumentacija, dobro organizovana sa obrazloženje projekta
Povećanje SU	50	40	30	20	10

TABELA 11.3 Rangiranje rada potrebnog za procenu i asimilaciju po COCOMO II

Inkrement procene i asimilacije	Rad na proceni i asimilaciji
0	Nikakav
2	Pretraživanje osnovnih komponenti i dokumentacije
4	Testiranje nekih komponenti i evaluacija dokumentacije
6	Prilično testiranje komponenti i evaluacija dokumentacije
8	Obimno testiranje komponenti i evaluacija dokumentacije

COCOMO II takođe uključuje rangiranje rada na proceni koda i pravljenju izmena, kao što je prikazano u tabeli 11.3. Što je više testiranja i dokumentacije potrebno, to je potreban veći napor.

Mnoga pitanja u pogledu procene vremena potrebnog za razvoj podjednako se odnose i na procene u vezi sa održavanjem. Najbolje procene se prave ako su zasnovane na iscrpnim istorijatima sličnih projekata u prošlosti. Pored toga, procene moraju ponovo da se proračunaju kada se menjaju atributi projekta i proizvoda; s obzirom na to da nasleđeni sistemi stalno evoluiraju, procene moraju da se rade redovno.

11.4 MERENJE KARAKTERISTIKA ODRŽAVANJA

Govorili smo o više svojstava softvera koja olakšavaju (ili ne olakšavaju) njegovo razumevanje, poboljšanje i ispravljanje. Koristeći ta svojstva kao softverske mere prilikom isporuke, možemo da predvidimo izvesnost da će njegovo održavanje biti lako. Tokom održavanja u radu možemo da se rukovodimo merenjima koja nam pomažu da procenimo uticaj izmena ili ocenimo relativne koristi od nekoliko predloženih izmena ili pristupa.

Mogućnost održavanja nije ograničena jedino na kôd; ona opisuje mnoge produkte softverskog procesa uključujući specifikaciju zahteva, specifikaciju dizajna i plan testiranja. Otuda je mere održljivosti potrebno imati za sve produkte koje je potrebno održavati.

O održljivosti možemo da razmišljamo na dva načina, koja odražavaju spoljašnje i unutrašnje viđenje softvera. Održljivost, onako kako je u ovoj knjizi definisana, predstavlja spoljašnji atribut softvera, zato što ne zavisi samo od proizvoda, nego i od ljudi koji sprovode održavanje, od dokumentacije, alata i predloženog načina korišćenja softvera. To znači da održljivost ne možemo meriti ako ne pratimo kako se softver ponaša u datom okruženju.

S druge strane, želimo da merimo održljivost pre stvarne isporuke softvera , da bismo stekli osećaj o resursima neophodnim za prevazilaženje svih problema koji mogu nastati. Ova merenja se oslanjaju na unutrašnje atribute softvera (na primer, one koji se odnose na strukturu) i na njima zasnivamo predviđanje spoljašnjih mera. S obzirom na to da ovaj pristup ne predstavlja direktno merenje, moramo da odmerimo praktičnost indirektnog u odnosu na tačnost spoljašnjeg pristupa.

Spoljašnje viđenje održljivosti

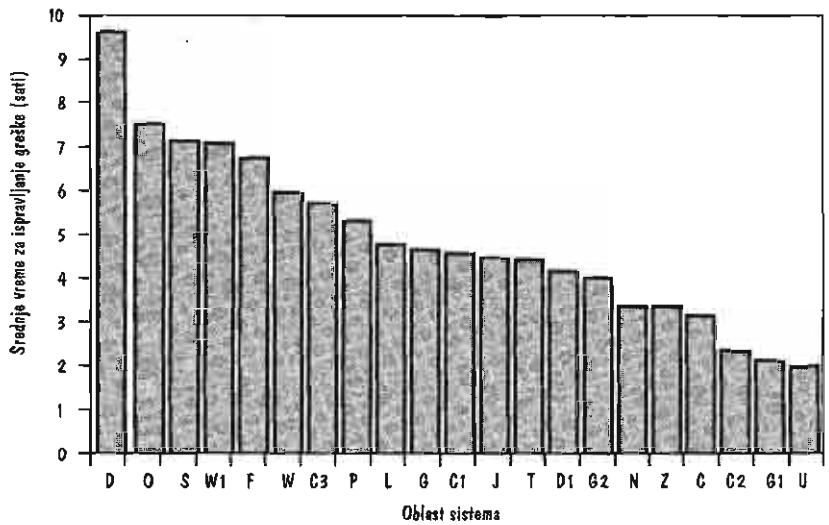
Da bismo održljivost merili kao srednje vreme popravke (kao što smo videli u poglavlju 9), za svaki problem su nam potrebni precizni zapisi o:

- trenutku izveštavanja o problemu;
- vremenu izgubljenom zbog administrativnog kašnjenja;
- vremenu potrebnom za analizu problema;
- vremenu potrebnom za određivanje neophodnih izmena;
- vremenu potrebnom za sprovođenje izmena;
- vremenu potrebnom za testiranje izmena;
- vremenu potrebnom za dokumentovanje izmena.

Na slici 11.5 prikazano je srednje vreme popravke različitih podsistema za softver u jednoj velikoj britanskoj firmi. Te informacije su bile korisne prilikom identifikovanja podsistema koji prouzrokuju najviše problema i planiranje aktivnosti preventivnog održavanja (Pfleeger, Fenton i Page 1994). Praćenje srednjeg vremena popravke pomoću takvih grafikona pokazuje da održljivost sistema raste ili opada.

Druge mere (zavisne od okruženja) mogu takođe da budu korisne, ako su na raspolaganju:

- odnos između ukupnog vremena implementacije izmene i ukupnog broja implementiranih izmena;
- broj nerešenih problema;
- vreme provedeno na nerešenim problemima;
- procenat izmena koje unose nove greške;
- broj komponenti koje je bilo potrebno modifikovati da bi se implementirala izmena.



SLIKA 11.5 Srednje vreme za ispravljanje grešaka po oblastima primene sistema (Pfleeger, Fenton i Page 1994)©1996 IEEE

Sve zajedno, ove mere stvaraju sliku stepena aktivnosti i efikasnosti procesa održavanja.

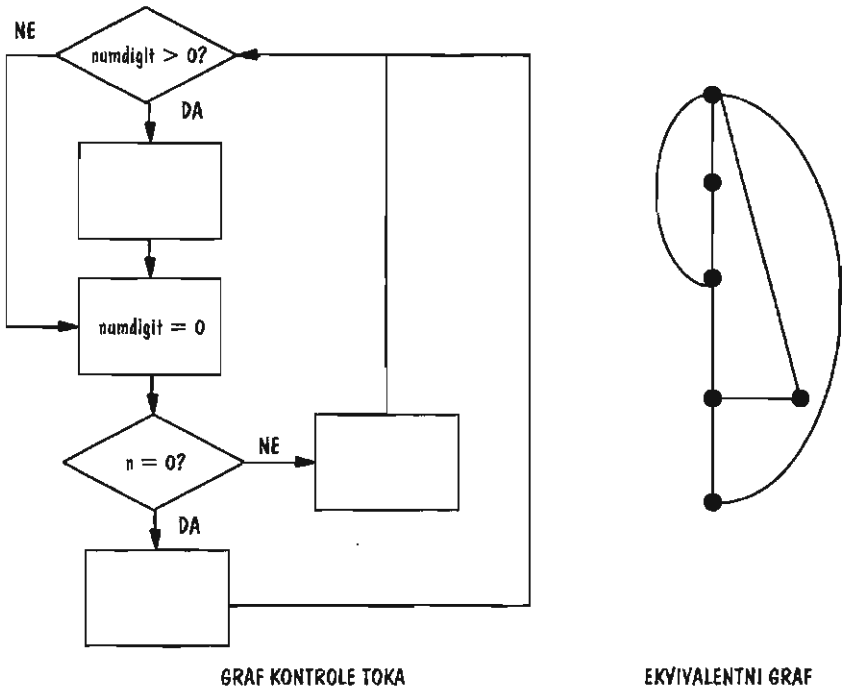
Unutrašnji atributi koji utiču na mogućnost održavanja

Mnogi istraživači su predložili mere unutrašnjih atributa koji su u vezi sa održljivošću. Na primer, mere složenosti, opisane u prethodnim poglavljima, često su u korelaciji s radom koji je ulažen u održavanje; što je kôd složeniji, potreban je veći rad za njegovo održavanje. Važno je zapamtiti da korelacija nije isto što i merenje, ali postoji jasna i intuitivna veza između loše strukturisanih i dokumentovanih proizvoda, i njihove održljivosti.

Ciklomatski broj. Jedna od najčešće korišćenih mera u postupku održavanja je ciklomatski broj, koga je prvi put definisao McCabe (1976). **Ciklomatski broj** predstavlja meru strukturne složenost izvornog koda iskazanu brojem linearno nezavisnih putanja kroz kôd. Zasnovan na konceptima teorije grafova, on se proračunava transformacijom programskog koda u ekvivalentni graf kontrole toka i primenom osobina grafova za određivanje vrednosti.

U cilju ilustracije načina određivanja ciklomatskog broja, posmatrajmo C++ kôd iz reference Lee i Tepfenhart (1997):

```
Scoreboard::drawscore(int n)
{   while(numdigits-> 0) {
          score[numdigits]->erase();
    }
    // build newe score in loop, each time update position
    numdigits = 0;
    // if score is 0, just display "0"
    if (n==0) {
      delete score [numdigits];
      score[numdigits] = new Displayable(digits[0]);
      score[numdigits]->move(Point((700-numdigits*18),40));
      score[numdigits]->draw();
      numdigits++;
}
while (n){
      int rem = n % 10;
      delete score[numdigits];
      score[numdigits] = new Displayable(digits[rem]);
      score[numdigits]->move(Point((700-numdigits*18),40));
      score[numdigits]->draw();
      n /=10;
      numdigits++;
    }
}
```



SLIKA 11.6 Primer za izračunavanje ciklomatskog broja

Graf kontrole toka prikazan je na levoj strani slike 11.6. Možemo da precrtamo taj graf dodeljujući čvor svakom rombu ili pravougaoniku i zatim povezujući čvorove granama, kao što je urađeno na prvobitnom grafu. Rezultat je graf sa n čvorova i e grana; u našem primeru, n je 6, a e 8. Na osnovu teorije grafova možemo utvrditi da je broj linearno nezavisnih putanja kroz graf jednak:

$$e - n + 2$$

odnosno 4 u našem primeru. McCabe je dokazao da je ciklomatski broj za jedan veći od broja iskaza odlučivanja u kodu. Ako pogledamo prethodni fragment programskog koda, uočavamo dva *while* i jedan *if* iskaz, pa je ciklomatski broj jednak 4. Jednostavan način izračunavanja ciklomatskog broja na osnovu grafa je da se posmatra način na koji graf segmentira ravan. U našem primeru, graf na desnoj strani deli stranicu na tri dela (trougao, polukrug i nepravilan oblik desno od trougla) plus još jedan deo za ostatak stranice. Dakle, graf deli ravan na četiri dela, što odgovara ciklomatskom broju.

Sličan proračun može da se napravi na bazi konstrukcije komponente, pa se ciklomatski broj često koristi za procenu više varijanti konstrukcije čak i pre početka samog kodiranja. Ciklomatski broj je takode koristan za razne druge procene. On ukazuje koliko je nezavisnih putanja potrebno testirati da bismo dobili potpuno pokrivanje putanja, pa se često meri i koristi pri izboru strategije testiranja.

U procesu održavanja, broj nezavisnih putanja ili, što je ekvivalentno, broj odluka plus jedan, ukazuje na stepen razumevanja i praćenja potreban pri ispitivanju ili menjanju komponente. Zato mnogi istraživači i praktičari nalaze da je korisno razmotriti uticaj izmene ili ispravke na ciklomatski broj razmatranog sistema ili komponente; ako izmena ili ispravka rezultira dramatičnim povećanjem ciklomatskog broja, neophodno je ponovo razmotriti predloženu izmenu ili ispravku. Zaista, Lehmanov drugi zakon evolucije softvera predviđa da će se ciklomatski broj (i druge mere složenosti) povećavati tokom evolucije sistema.

Neophodan je izvestan oprez pri korišćenju ove ili bilo koje druge mere za predstavljanje ukupne složenosti softvera. Istina je da povećanje broja odluka ili putanja obično otežava razumevanje koda, ali postoje i drugi atributi koji doprinose složenosti, a nisu obuhvaćeni strukturom koda. Na primer, hijerarhija nasleđivanja objektno orijentisanih programa može da bude izuzetno složena i da ima grananja koja nisu očigledna kada se proučava samo jedna komponenta izvan konteksta. Istraživači i dalje tragaju za boljim metodama za definisanje i merenje složenosti, da bi nam pomogli u potrazi za jednostavnim sistemima koji su laki za održavanje.

Ostale mere za proizvode

Postoji više atributa proizvoda koji pomažu da razumemo mogućnost održavanja i predvidimo verovatne izvore problema. Neke organizacije koriste pravila zasnovana na merama za komponente, kao što je veličina. Na primer, Möller i Paulish (1993) su pokazali da su, u Siemensu, manje komponente (izraženo u redovima koda) imale veću gustinu grešaka od većih (pročitajte podsetnik 11.4). Drugi istraživači su koristili informacije o dubini ugnežđivanja, broju operatora i operanada, broju ulaza i broju izlaza, da bi predvideli kvalitet održavanja. U podsetniku 11.5 opisan je pristup u kompaniji Hewlett-Packarda za pravljenje indeksa održljivosti.

PODSETNIK 11.4 MODELI POGREŠNOG PONAŠANJA

Hatton i Hopkins (1989) su proučavali NAG biblioteku naučnih Fortran potprograma koja se sastojala od 1 600 rutina sa ukupno 250 000 izvršnih linija koda. Biblioteka je kroz 15 izdanja, prisutna više od 20 godina, pa je za istraživanje postojala obimna istorija održavanja. Bili su zapanjeni kada su otkrili da su manje komponente sadržale srazmerno više grešaka od onih većih (gde je veličina iskazivana brojem statičkih putanja).

Hatton je išao i dalje, tražeći sličan dokaz od drugih istraživača. On je zapazio da su Möller i Paulish (1993) izvestili o istoj pojavi kod Siemensa, gde je veličina merena redovima koda. Withrow (1990) opisuje slično ponašanje Ada programskog koda kod Unisysa, kao i Basili i Perricone (1984) za Fortran proizvode kod NASA Goddarda.

Međutim, Rosenberg (1998) ističe da su ti izveštaji zasnovani na poređenju veličine sa gustinom grešaka. Kako se gustina meri brojem grešaka podeljenim sa veličinom, veličina je deo oba činioca koji se porede. Dakle, može postojati jaka negativna korelacija između dva činioca, koja maskira stvarni odnos između grešaka i veličine. Rosenberg upozorava da dobro obratimo pažnju na razumljivost definicije mera pre nego što na njih primenimo statističke tehnike.

PODSETNIK 11.5 MERE ODRŽAVANJA KOD HEWLETT-PACKARDA

Oman i Hagemeister (1992) su pretpostavili da se održivost može modelovati korišćenjem tri dimenzije: upravljačke strukture, informacione strukture i tipografije, imenovanja i komentara sistema o čijem se održavanju radi. Oni definišu metrike za svaku dimenziju i kombinuju ih u indeks održivosti sistema.

Taj indeks je upotrebio Coleman sa saradnicima u Hewlett-Packardu (HP) da proceni održivost različitih softverskih sistema. Prvo, indeks je bio kalibrisan na osnovu brojnih metrika i izračunatog namenskog polinomijalnog indeksa na osnovu proširenog ciklomatskog broja, redova koda, broja komentara i Helsteadove mere rada (1977). Nakon toga je polinom primenjen na 741 komponentu sa 236 000 redova C koda razvijenih od strane nezavisnih proizvođača. Analiza mogućnosti održavanja je rezultirala rangiranjem komponenti, na osnovu čega je kompanija HP izolovala one koje su bile teške za održavanje. Taj rezultat se poklapao sa „predosećajem" HP-ovih servisera o teškoćama prilikom održavanja.

Polinomi su takođe korišćeni za poređenje dva softverska sistema koji su bili slični po veličini, broju modula, platformi i jeziku. Rezultat je opet potvrdio osećaj HP inženjera. I u više potonjih analiza, rezultat primene polinoma se poklapao sa intuicijom servisera. Merenja su pružila dodatne informacije neophodne u procesu donošenja odluke da li da se radi vlastiti razvoj ili da se kupuju gotove komponente, izdvojila komponente za preventivno održavanje i unapređenje i procenu efekata reinženjeringa.

Porter i Selby su koristili statističku tehniku poznatu kao analiza stabla klasifikacije za identifikaciju mera proizvoda koje su najbolje za predviđanje grešaka u sklopu interfejsa na koje se verovatno može naići tokom održavanja. Stablo odlučivanja koje predstavlja rezultat iz analize klasifikacionih podataka ukazuje na merljiva ograničenja zasnovana na istorijatu:

- Između četiri i osam revizija u fazi projektovanja i najmanje 15 povezivanja podataka ukazuju da su greške u interfejsu verovatne.
- Problemi u sklopu interfejsa su verovatni kod komponente čija je osnovna funkcija upravljanje datotekama i gde je bilo najmanje devet revizija u toku projektovanja.

Ove napomene su specifične za rad sa skupovima podataka i nemaju opštu namenu. Međutim, tehnika može da se primeni na bilo koju bazu podataka koja sadrži informacije o merenjima.

Za tekstualne proizvode, čitljivost utiče na mogućnost održavanja. Najpoznatija mera čitljivosti je Gunningov indeks zamagljivanja (**Fog Index**), F, definisan kao

$$F = 0,4 \times \left(\frac{\text{broj reči}}{\text{broj rečenica}} + \frac{\text{procenat reči}}{\text{sa tri ili više slogova}} \right)$$

Mera grubo odgovara broju godina školovanja potrebnom da bi neko mogao da pročita odlomak sa lakoćom i razumevanjem. Za velike dokumente, mera se obično računa iz uzorka teksta (Gunning 1968).

Druge mere čitljivosti su specifične za softverske proizvode. De Young i Kampen (1979) definišu čitljivost R izvornog koda kao

$$R = 0{,}295a - 0{,}499b + 0{,}13c$$

gde je a prosečna normalizovana dužina promenljivih (dužina promenljive je broj znakova u nazivu promenljive), b broj redova koji sadrže iskaze, a c McCabeov ciklomatski broj. Formula je izvedena upotrebom regresione analize podataka o subjektivnim procenama čitljivosti.

11.5 TEHNIKE I ALATI ZA ODRŽAVANJE

Jedan od načina da se smanji rad u procesu održavanja jeste da se kvalitet ugradi od samog početka. Pokušaj da se dobra konstrukcija i struktura sprovede na već izgrađenom sistemu manje je uspešan od pravilne gradnje sistema od početka. Međutim, pored dobre prakse, postoji više drugih tehnika koje podižu razumljivost i kvalitet.

Upravljanje konfiguracijom

Praćenje izmena i njihovih efekata na druge komponente sistema nije lak zadatak. Što je sistem složeniji, izmena utiče na više komponenti. Iz tog razloga, upravljanje konfiguracijom, koje je važno u toku razvoja, postaje kritično u vreme održavanja.

Odbor za upravljanje konfiguracijom. S obzirom na to da kupci i korisnici iniciraju mnogobrojne izmene u fazi održavanja (kako se pojavljuju otkazi ili zahtevaju poboljšanja), ustanovljava se **odbor za upravljanje konfiguracijom** sa ciljem da nadzire unošenje izmena. Odbor čine predstavnici svih zainteresovanih strana, uključujući kupce, projektante i korisnike. Svakom problemu se pristupa na sledeći način:

1. Problem otkriva korisnik, kupac, ili projektant, koji evidentira simptome na propisanom obrascu za upravljanje izmenom. Alternativno, kupac, korisnik, ili projektant zahteva poboljšanje: novu funkciju, varijaciju stare funkcije, ili uklanjanje postojeće funkcije. Obrazac, sličan izveštaju o otkazu koji smo proučili u poglavlju 9, mora da sadrži informacije o sadašnjem načinu rada sistema, prirodi problema ili poboljšanja i budućem načinu rada sistema.
2. O predloženoj promeni izveštava se odbor za upravljanje konfiguracijom.
3. Odbor za upravljanje konfiguracijom se sastaje da bi raspravljao o problemu. Prvo, on određuje da li predlog predstavlja otkaz u sklopu zahteva, ili se radi o zahtevu za poboljšanje. Ta odluka obično utiče na to ko će da finansira resurse potrebne za implementaciju izmena.
4. Za otkaz o kome je izvešten, odbor za upravljanje konfiguracijom raspravlja o verovatnom uzroku problema. Za zahtevano poboljšanje, odbor raspravlja o delovima sistema na koje će verovatno uticati izmena. U oba slučaja, programeri i analitičari mogu da opišu opseg svake neophodne izmene i očekivano vreme njihove imple-

mentacije. Upravni odbor dodeljuje prioritet ili nivo ozbiljnosti zahtevu, a programera ili analitičara imenuje odgovorno lice za realizaciju izmene u sistemu.

5. Imenovani analitičar ili porgramer pronalazi odakle potiče problem ili pronalazi komponente na koje se odnosi zahtev i zatim identifikuje neophodne izmene. Radeći na testnom primerku a ne na operativnoj verziji sistema, programer ili analitičar implementira i testira izmene da bi bio siguran da one rade.
6. Imenovani analitičar ili programer radi sa programskim bibliotekarem u cilju kontrolisanja implementacije izmena u operativnom sistemu. Sva relevantna dokumentacija se ažurira.
7. Programer ili analitičar pravi izveštaj o promeni u kome detaljno opisuje sve izmene.

Upravljanje izmenom. Najkritičniji korak u procesu je onaj pod brojem 6. U svakom trenutku, tim za upravljanje konfiguracijom mora da zna stanje svake komponente ili dokumenta u sistemu. Shodno tome, upravljanje konfiguracijom bi trebalo da pojača komunikaciju između onih čije akcije utiču na sistem. Cashman i Holt (1980) sugerišu da je neophodno uvek znati odgovore na sledeća pitanja:

- *Sinhronizacija:* Kada je napravljena izmena?
- *Identifikacija:* Ko je napravio izmenu?
- *Imenovanje:* Koje komponente sistema su izmenjene?
- *Provera autentičnosti:* Da li je izmena napravljena ispravno?
- *Autorizovanje:* Ko je odobrio da se napravi izmena?
- *Rutiranje:* Ko je bio obavešten o izmeni?
- *Obustavljanje:* Ko može da obustavi zahtev za izmenom?
- *Delegiranje:* Ko je odgovoran za izmenu?
- *Vrednovanje:* Koji je prioritet izmene?

Imajte u vidu da su to pitanja rukovođenja, a ne tehnička pitanja. Da bismo pažljivo upravljali izmenom, moramo da koristimo procedure.

Upravljanje izmenom može da se potpomogne sprovođenjem više konvencija. Prvo, svakoj radnoj verziji se dodeljuje identifikacioni kôd ili broj. Kada se verzija menja, revizioni kôd ili broj se dodeljuje svakoj promenjenoj komponenti. Pravimo zapis o svakoj verziji komponente i statusu, kao i istorijat svih izmena. Potom, u svakom trenutku životnog veka sistema, tim za upravljanje konfiguracijom može da identifikuje sadašnju verziju radnog sistema i broj revizije svake komponente u upotrebi. Tim takođe može da otkrije kako se različite verzije razlikuju, ko je pravio izmene i zašto su one napravljene.

Iz vaše studentske perspektive, ove konvencije upravljanja konfiguracijom verovatno izgledaju nepotrebne. Vašim školskim projektima verovatno upravljaju pojedinci ili male grupe programera, koristeći verbalnu komunikaciju da bi se pratile izmene i poboljšanja. Međutim, zamislite haos koji bi nastao zbog korišćenja tih istih tehnika u razvoju i održavanju sistema od 200 komponenti. Veliki sistem često razvija više nezavisnih grupa koje istovremeno rade na različitim aspektima sistema; ponekad se te grupe nalaze u različitim delovima grada, ili čak i u različitim gradovima. Kada loša komunikacija dovede do otkaza, tim za upravljanje konfiguracijom mora da bude u stanju da ponovo vrati sistem

u njegovo prethodno stabilno stanje; taj korak može da se preduzme samo kada tim tačno zna ko je napravio kakve izmene, na kojim komponentama i kada.

Analiza uticaja

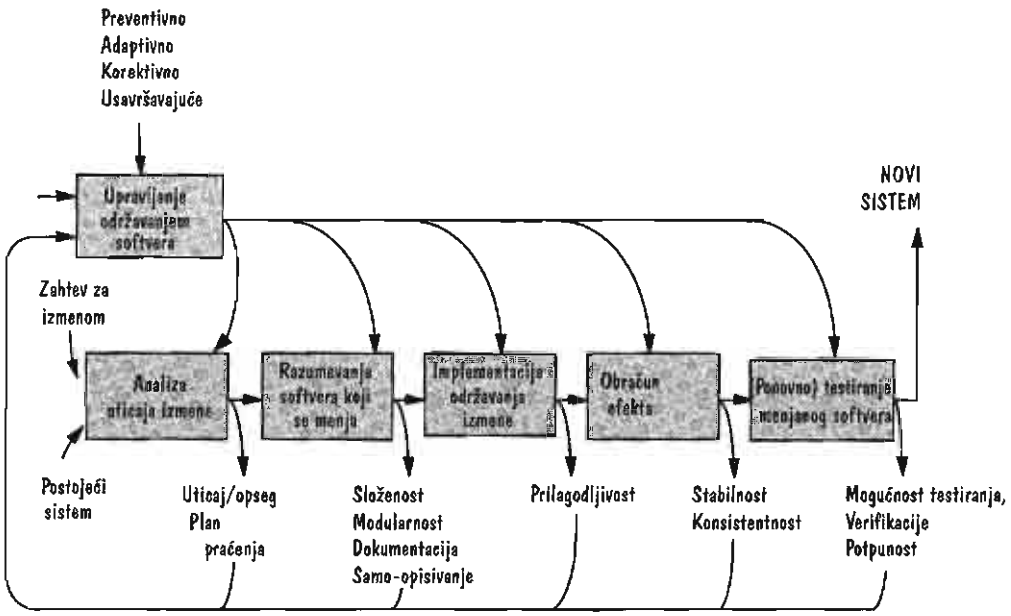
Prema tradicionalnom shvatanju životnog veka softvera, održavanje počinje po završetku razvoja softvera. Međutim, održavanje softvera zavisi od korisničkih zahteva i počinje još sa njima. Prema tome, principi razvoja dobrog softvera primenjuju se i na proces razvoja i na proces održavanja. S obzirom na to da razvoj dobrog softvera podržava njegovu izmenu, o njoj treba razmišljati u toku celog životnog veka softverskog proizvoda. Štaviše, naizgled minorna izmena je često obimnija (i zato skuplja za implementaciju) nego što se to očekuje. **Analiza uticaja** je procena stepena rizika u vezi sa promenom, uključujući procenu uticaja na resurse, rad i vremenske rokove.

Efekti mnogostrukih izmena u sistemu mogu da se vide u neodgovarajućoj ili zastareloj dokumentaciji koja nastaje u nepravilno ili nepotpuno popravljanom softveru, loše strukturisanom dizajnu ili kodu, artefaktima koji ne podležu standardima i još mnogo čemu. Problem je mešavina usložnjavanja, dužeg vremena potrebnog da bi projektanti razumeli kôd koji je pretrpeo izmenei više sporednih uticaja koje izmene mogu da imaju na druge delove sistema. Ti problemi povećavaju troškove održavanja, a rukovodstvo bi želelo da te troškove drži pod kontrolom. Analizu uticaja možemo da koristimo da bismo podržali upravljanje troškovima održavanja.

Pfleeger i Bohner (1990) su istraživali načine za merenje uticaja koji predložena izmena ima na razne aspekte softvera da bi odredili rizike i odmerili različite opcije. Oni opisuju model održavanja softvera koji uključuje merenje povratne sprege. Na dijagramu na slici 11.7 ilustrovane su aktivnosti koje se izvode kada se zahteva izmena, a natpisi uz strelice u donjoj polovini dijagrama predstavljaju merenja koja daju informacije koje menadžeri mogu da koriste pri donošenju odluke kada i kako da se napravi izmena.

Produkt je svaki artefakt razvoja čija izmena je značajna. Dakle, zahtevi, konstrukcija i komponente koda, test slučajevi i dokumentacija su proizvodi rada; kvalitet jednog od njih može da utiče na kvalitet ostalih, pa njihova izmena može da ima značajne posledice. Uticaj izmene možemo da procenimo za sve produkte. Za svaki od njih, **vertikalna pratljivost** izražava odnose između delova proizvoda. Na primer, vertikalna pratljivost opisuje međusobne zavisnosti zahteva i sistema. **Horizontalna pratljivost** bavi se odnosima komponenti u kolekciji produkata. Na primer, svaka komponenta konstrukcije prati se do nivoa komponenti koda koje implementiraju taj deo konstrukcije. Obe vrste pratljivosti su potrebne za razumevanje potpunog skupa odnosa u postupku analize uticaja.

I horizontalnu i vertikalnu pogodnost za praćenje možemo da prikažemo koristeći usmerene grafove. **Usmereni graf** je jednostavno zbirka objekata koji se zovu **čvorovi** i pridružena zbirka uređenih parova čvorova, koji se zovu **grane**. Prvi čvor grane zove se **čvor izvor**, a drugi je **čvor odredište**. Čvorovi predstavljaju informacije koje se nalaze u dokumentima, člancima i drugim artefaktima. Svaki artefakt sadrži čvor za svaku komponentu. Na primer, možemo da predstavimo projekat kao zbirku čvorova, sa po jednim čvorom za svaku komponentu konstrukcije, a specifikaciju zahteva tako da ima po jedan čvor za svaki zahtev. Usmerene grane predstavljaju odnose unutar jednog produkta i između produkata.



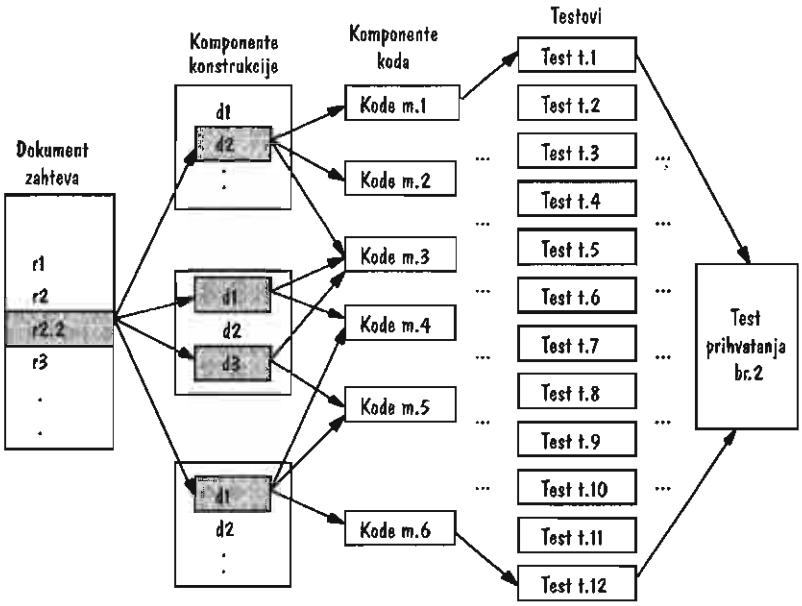
SLIKA 11.7 Aktivnosti održavanja softvera

Na slici 11.8 ilustrovano je kako se određuju grafički odnosi i pratljive veze između odgovarajućih produkata. Ispitujemo svaki zahtev i crtamo vezu između zahteva i komponenti konstrukcije koje ga implementiraju. Zatim, povezujemo svaku komponentu konstrukcije sa komponentama koda koje je implementiraju. Najzad, povezujemo svaki modul koda sa skupom slučajeva za testiranje. Povezanost koja se tako dobije formira graf koji prikazuje odnose između produkata.

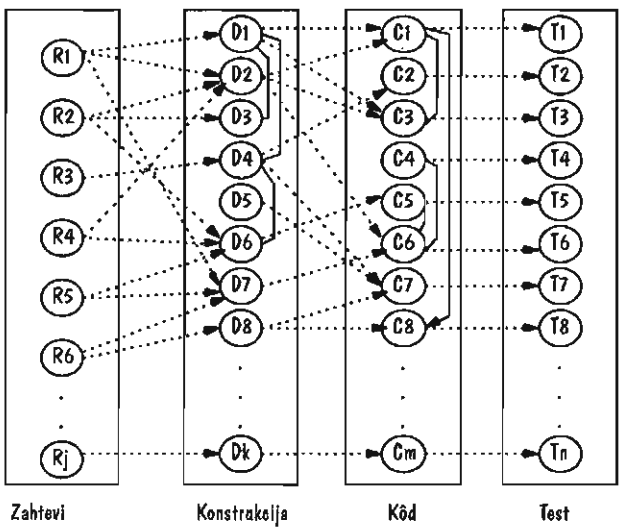
Na slici 11.9 prikazano je kako bi mogao da izgleda graf ukupne pratljivosti. Svaki artefakt glavnog procesa (zahtevi, konstrukcija, kôd i test) prikazan je kao pravougaonik oko konstitutivnih čvorova. Pune grane unutar svakog pravougaonika su odnosi vertikalne pratljivosti komponenti u pravougaoniku. Isprekidane grane između pravougaonika prikazuju veze horizontalne pratljivosti u sistemu. U podsetniku 11.6 opisuje se kako je ovaj pristup primenjen u kompaniji Ericsson.

Postoji više dokaza da su neke mere složenosti dobri indikatori potrebnog rada i učestalosti grešaka (Card i Glass 1990.) Ti pojmovi mogu da se prošire na karakteristike grafa pratljivosti, da bi se procenio uticaj predložene izmene. Uzmimo, na primer, graf vertikalne pratljivosti unutar svakog pravougaonika na slici 11.9. Ukupan broj čvorova, broj grana čije je odredište u čvoru (takozvani **ulazni stepen** čvora) i grana čiji je izvor u čvoru (**izlazni stepen**), zatim mere kao što je ciklomatski broj, mogu da se izračunaju pre i posle izmene. Ako izgleda da se veličina i složenost grafa sa izmenom povećavaju, verovatno je da će se veličina i složenost odgovarajućih produkata takođe povećavati. Koristeći tu informaciju, odbor za upravljanje konfiguracijom može da odluči da implementira izmenu na drugačiji način, ili da to uopšte ne radi. Čak i ako rukovodstvo odluči

da uvede predloženu izmenu, rizik koji ide uz nju može bolje da se razume pomoću ove slike zasnovane na merenjima.



SLIKA 11.8 Horizontalna pratljivost u softverskim proizvodima rada



SLIKA 11.9 Prateći graf održavanja

PODSETNIK 11.6 PRIMENA PRATLJIVOSTI NA SISTEME IZ STVARNOG OKRUŽENJA

Lindvall i Sandhal (1996) primenili su Pfleegerovu i Bohnerovu pogodnost za praćenje na jedan objektno orijentisani projekat u kompaniji Ericsson Radio Systems. Kao rezultat, oni su konstruisali dvodimenzionalni okvir pratljivosti. Prva dimenzija obuhvata stavke koje se prate. Na primer, oni su implementirali pet vrsta pratljivosti:

- između objekata;
- između asocijacija;
- između slučaja korišćenja;
- između slučaja korišćenja i objekata;
- između dvodimenzionalnih objekata (uključujući i nasleđivanje).

Druga dimenzija pokazuje kako se praćenje obavlja:

- korišćenjem eksplicitnih veza;
- korišćenjem tekstualnih referenci na različite dokumente;
- korišćenjem imena i koncepata koji su isti i slični;
- korišćenjem znanja o sistemu i domenu.

Proces formiranja veza doveo je do otkrića i rešavanja problema, kao i do razjašnjenja značenja mnogih aspekata sistema. Oni su iz svog istraživanja zaključili da „ako je pogodnost za praćenje naglašen činilac kvaliteta od samog početka projekta, dokumentacija će biti jasnija i doslednija, postići će se bolje razumevanje među projektantima, ulazi u projekat će biti bolje usmereni, a održavanje proizvoda će manje zavisiti od pojedinih stručnjaka". Međutim, neki od njihovih radova u oblasti pratljivosti zahtevali su dosta rada. Lindvall i Sandhal nagoveštavaju da postoje najmanje dve situacije koje zahtevaju značajan rad:

- praćenje stavki bez alata za praćenje veza (kao kod pratljivosti između asocijacija);
- praćenje u modelima koji su delimično nedosledni ili nedovoljno dokumentovani.

Mere vertikalne pratljivostisu one mere proizvoda koje pokazuju dejstvo izmene na svaki produkt koji se održava. Izmerene karakteristike na grafu horizontalne pratljivosti daju procesni pogled na izmenu. Za svaki par produkata, možemo da formiramo podgrafove sledećih odnosa: prvi podgraf sa odnosima između odgovarajućeg zahteva i konstrukcije, drugi sa odnosima između odgovarajuće konstrukcije i koda, i treći koji pokazuje odnose između koda i test slučajeva. Nakon toga merimo veličinu i složenost od-

nosa da bismo odredili veličinu štetnosti uticaja. Štaviše, možemo da sagledamo ukupnu horizontalnu pratljivost da bismo videli da li je ukupna pratljivost posle izmene postaje otežana ili olakšana. Pfleeger i Bohner (1990) su tražili minimalni skup putanja koje obuhvata taj graf; ako se broj putanja povećava posle izmene, onda će sistem verovatno biti nezgrapniji i teži za održavanje. Slično tome, ako se ulazni i izlazni stepeni značajno povećaju, sistem će se u budućnosti teže održavati.

Automatizovani alati za održavanje

Praćenje statusa svih komponenti i testova je ogroman posao. Na sreću, postoji više automatizovanih alata koji nam pomažu u održavanju softvera. Ovde opisujemo neke vrste takvih alata; na veb stranici koja prati ovu knjigu postoje pokazivači na lokacije proizvođača i prikaze alata.

Editori teksta. Editori teksta su na više načina korisni za održavanje. Prvo, editor može da kopira kôd ili dokumentaciju sa jednog mesta na drugo, sprečavajući greške koje se javljaju kada sami pravimo duplikat teksta. Drugo, kao što smo videli u poglavlju 9, neki editori teksta u posebnim datotekama prate izmene koje se odnose na osnovnu datoteku. Mnogi od tih editora označavaju vreme i podatke za svaki unos i obezbeđuju način da se vrati sa trenutne verzije datoteke na prethodnu, ako je to potrebno.

Komparatori datoteka. **Komparator datoteka** je koristan alat za vreme održavanja, koji poredi dve datoteke i izveštava o njihovim razlikama. Često ga koristimo kako bismo se uverili da su dva sistema ili programa stvarno istovetni, ako radimo s takvom pretpostavkom. Program čita obe datoteke i ukazuje na nesaglasnosti.

Kompajleri i linkeri. Kompajleri i linkeri često imaju svojstva koja pojednostavljuju održavanje i upravljanje konfiguracijom. Kompajler proverava sintaksne greške u kodu, ukazujući u mnogim slučajevima na mesto i vrstu greške. Kompajleri nekih jezika, kao što su Modula-2 i Ada, takođe proveravaju doslednost preko posebno prevedenih komponenti.

Kada se kôd ispravno prevede, linker (povezivač ili editor veza) povezuje kôd sa drugim komponentama koje su potrebne za izvršavanje programa. Na primer, linker povezuje datoteku *filename.h* sa odgovarajućom datotekom *filename.c* u jeziku C. Ili, linker može da naznači poziv potprograma, biblioteke ili makroa, automatski donoseći potrebne datoteke da bi napravio celinu koju je moguće prevesti. Neki linkeri takođe prate brojeve verzija svake od zahtevanih komponenti, tako da se povezuju samo odgovarajuće verzije. Ta tehnika pomaže da se odstrane problemi čiji su uzrok pogrešne kopije sistema ili podsistema kada se ispituje izmena.

Alati za pronalaženje i uklanjanje grešaka. Alati za pronalaženje i uklanjanje grešaka pomažu pri postepenom praćenju logike programa, ispitujući sadržaj registara i memorijskih područja i postavljajući indikatore i pokazivače.

Generatori unakrsnih referenci. O značaju pratljivosti govorili smo ranije u ovom poglavlju. Automatizovani sistemi generišu i skladište unakrsne reference da bi obezbedili i razvojnom timu i timu za održavanje bolje upravljanje izmenama u sistemu. Na primer,

neki alati za unakrsne reference deluju kao skladišta sistemskih zahteva, a takođe sadrže i veze ka drugim sistemskim dokumentima i kodu koji se odnose na pojedinačne zahteve. Kada se predloži izmena zahteva, možemo da upotrebimo alat da bi nam on saopštio na koje druge zahteve i delove dizajna i koda će one uticati.

Neki alati za unakrsno referenciranje sadrže skup logičkih formula koje se zovu uslovi za verifikaciju; ako svaka od formula daje vrednost „istinito", onda kôd zadovoljava specifikaciju na osnovu koje je nastao. To svojstvo je posebno korisno za vreme održavanja, da bi se obezbedilo da promenjeni kôd bude i dalje u saglasnosti sa svojim specifikacijama.

Statički analizatori koda. Statički analizatori koda proračunavaju informacije o strukturnim atributima koda, kao što su dubina ugnežđivanja, broj obuhvatnih putanja, ciklomatski broj, broj redova koda i nedostižnih iskaza. Te informacije možemo da izračunamo kada pravimo nove verzije sistema koje održavamo, da bismo videli da li one postaju veće, složenije i teže za održavanje. Merenja nam takođe pomažu da odaberemo jednu varijantu dizajna među onima koje su nam ponuđene, posebno kada ponovo projektujemo delove postojećeg koda.

Spremišta upravljanja konfiguracijom. Upravljanje konfiguracijom bilo bi nemoguće bez biblioteka informacija koje kontrolišu proces izmene. Ta spremišta mogu da sadrže izveštaje o problemima, uključujući informacije o svakom od njih, o organizaciji koja je o njemu izvestila i organizaciji koja ga rešava. Neka spremišta dozvoljavaju korisnicima da čuvaju beleške o najavljenim problemima u sistemima koje koriste. Druga, kao što je alat opisan u podsetniku 11.7, izvršavaju jednu verziju kontrole i unakrsnog referenciranja.

PODSETNIK 11.7 PANVALET

Panvalet je popularan alat koji se koristi na IBM-ovim velikim računarima. On uključuje izvorni kôd, objektni kôd, upravljački jezik i datoteke podataka koji su potrebni da bi sistem radio. Datotekama su dodeljeni tipovi, a različite datoteke se mogu spajati. Ta osobina dozvoljava projektantu da menja niz u jednoj datoteci, u svim datotekama datog tipa ili u čitavoj biblioteci datoteka.

Panvalet kontroliše više od jedne verzije sistema, pa datoteka može da ima više verzija. Jedna verzija je imenovana kao proizvodna verzija i nikom nije dozvoljeno da je menja. Da bi izmenio tu datoteku, projektant mora da napravi novu verziju i onda da menja novu datoteku.

Datoteke su organizovane u hijerarhiji i međusobno su unakrsno referencirane. Svakoj verziji datoteke pridružen je katalog informacija o verziji: njen status s obzirom na proizvodnu verziju, datumi poslednjeg pristupa i poslednjeg ažuriranja, broj iskaza u datoteci i vrsta poslednje preduzete akcije nad datotekom. Kada se datoteka prevede, Panvalet automatski postavlja broj verzije i datum poslednje izmene na listingu kompajlera i objektnom modulu.

Panvalet takođe ima i svojstva za izveštavanje, pravljenje rezervnih kopija i oporavak, plus tri nivoa bezbednog pristupa. Kada se datoteke ne koriste u dužem periodu, Panvalet može da ih arhivira.

11.6 PODMLAĐIVANJE SOFTVERA

U mnogim organizacijama koje koriste glomazan softver održavanje tih sistema predstavlja pravi izazov. Da biste videli zašto, zamislite osiguravajuće društvo koje nudi novu vrstu životnog osiguranja. Da bi podržalo taj svoj proizvod, društvo razvija softver za rad sa polisama osiguranja, informacijama o vlasnicima polisa i finansijskim i računovodstvenim informacijama. Takve polise treba da se podržavaju desetinama godina; ponekad softver ne može da se povuče dok ne umre i poslednji nosilac polise i dok se ne srede sva potraživanja. Rezultat svega toga je da osiguravajuće društvo verovatno treba da podržava mnogo različitih aplikacija na različitim platformama, sa velikim brojem implementacionih jezika. Organizacije koje su u takvoj situaciji moraju da donose teške odluke o tome kako da učine svoje sisteme pogodnijim za održavanje. Njihov izbor se kreće u opsegu od poboljšanja do potpune zamene novom tehnologijom; svaki izbor treba da sačuva ili poveća kvalitet softvera, uz istovremeno zadržavanje troškova na što je moguće nižem nivou.

Podmlađivanje softvera odgovara na taj izazov održavanja pokušavajući da poveća ukupan kvalitet postojećeg sistema. Ono se vraća produktima rada sistema pokušavajući da dobije dodatne informacije ili da ih reformatira na razumljiviji način. Postoji više aspekata softverskog podmlađivanja, uključujući:

- redokumentovanje;
- restrukturisanje;
- reverzni inženjering;
- reinženjering.

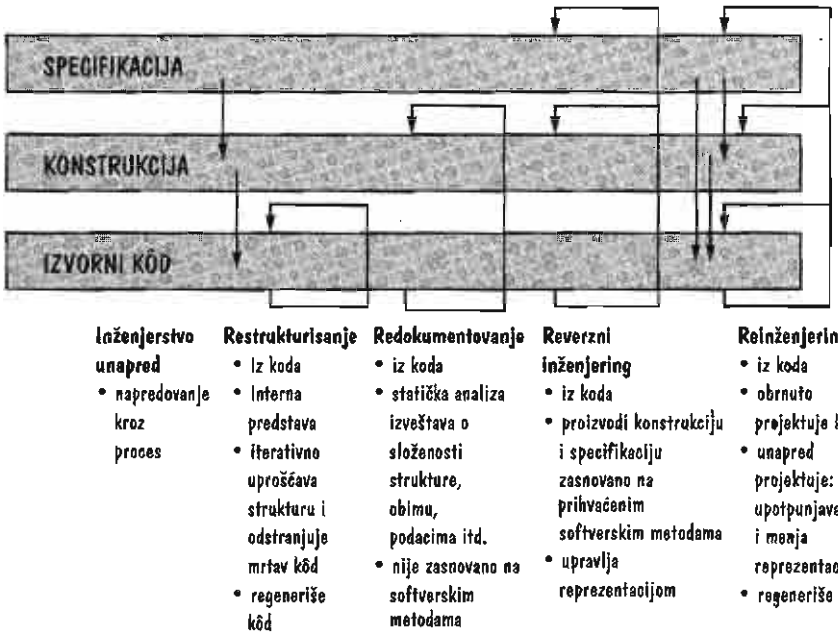
Kada **redokumentujemo** sistem, izvodimo statičku analizu izvornog koda proizvodeći dodatne informacije da bismo pomogli serviserima u razumevanju i referenciranju koda. Analiza ne čini ništa da bi transformisala aktuelni kôd; ona samo izvodi informacije. Međutim, kada **restrukturišemo**, stvarno menjamo kôd, transformišući loše strukturisani kôd u dobro strukturisani. Obe ove tehnike bave se samo izvornim kodom. Da bismo izvršili **reverzni inženjering** sistema, vraćamo se od izvornog koda ka proizvodima koji su mu prethodili, ponovo stvarajući informacije o konstrukciji i specifikaciji iz koda. **Reinženjering** je još obuhvatniji jer se obavlja reverzni inženjering postojećeg sistema koji se zatim „projektuje unapred" radi izmena u specifikaciji i konstrukciji koje upotpunjuju logički model; zatim se stvara novi sistem na osnovu revidirane specifikacije i konstrukcije. Na slici 11.10 ilustrovani su odnosi između četiri vrste podmlađivanja.

Naravno, nemoguće je očekivati da se ponovo stvore svi radni međuproizvodi iz dela izvornog koda. Takav zadatak je uporediv sa ponovnim stvaranjem slike deteta iz slike odrasle osobe. Svejedno, neka suštinska svojstva nekih produkata rada mogu da se poboljšaju ili doteraju. Stepen u kome informacija može da se izdvoji iz finalnog proizvoda zavisi od više činilaca (Bohner 1990):

- jezika koji se koriste;
- interfejsa baze podataka;
- korisničkih interfejs;
- interfejsa ka sistemskim uslugama;

- interfejsa ka drugim jezicima;
- zrelosti i stabilnosti domena;
- raspoloživih alata.

Sposobnost servisera, njihovo znanje i iskustvo takođe igraju važnu ulogu kada je reč o stepenu u kome informacije mogu uspešno da se interpretiraju i koriste.



SLIKA 11.10 Taksonomija podmlađivanja softvera (Bohner 1990)

Redokumentovanje

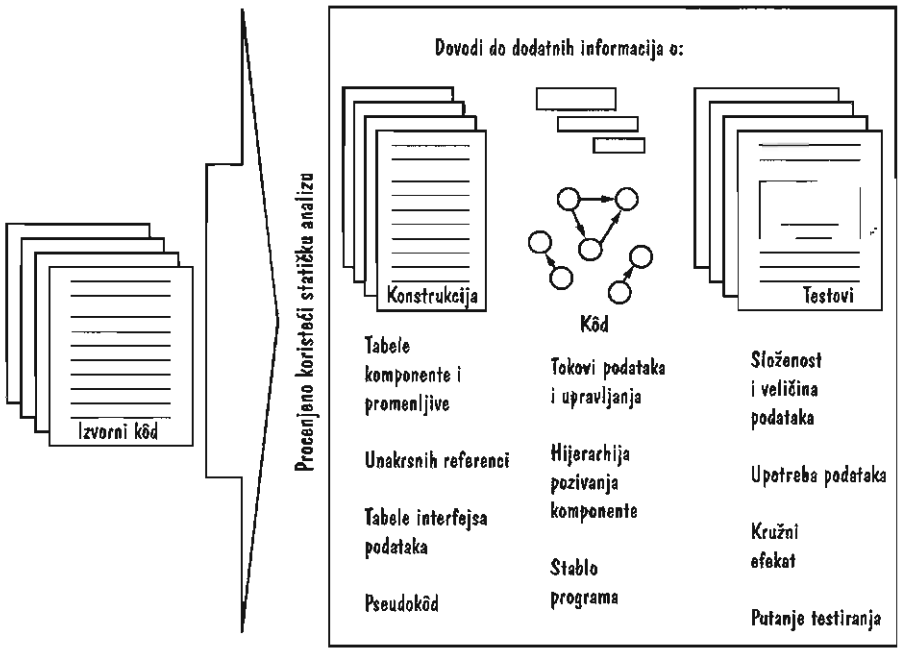
Redokumentovanje obuhvata statičku analizu izvornog koda da bi se proizvela dokumentacija sistema. Možemo da ispitamo upotrebu promenljivih, pozive komponenti, putanje upravljanja, veličinu komponenti, pozivajuće parametre, putanje testiranja i druge odnosne mere koje nam pomažu da razumemo šta i kako kôd radi. Informacije koje proizvede statička analiza koda mogu da budu grafičke ili tekstualne.

Na slici 11.11 ilustrovan je proces redokumentovanja. Uobičajeno je da serviser počinje redokumentovanje analizirajući kôd analitičkim alatom. Izlaz može da uključi:

- pozivne odnose između komponenti;
- hijerarhije klasa;
- interfejsne tabele za podatke;
- informacije o rečniku u vezi s podacima;
- tabele ili dijagrame toka podataka;

- tabele ili dijagrame toka upravljanja;
- pseudokôd;
- putanje testiranja;
- unakrsno referenciranje komponenti i promenljivih.

Grafičke, tekstualne i tabelarne informacije mogu da se upotrebe za procenu da li sistem zahteva ili ne zahteva restrukturisanje. Međutim, kako nema preslikavanja između specifikacije i restrukturisanog koda, rezultujuća dokumentacija odražava ono što jeste, a ne ono što bi trebalo da bude.

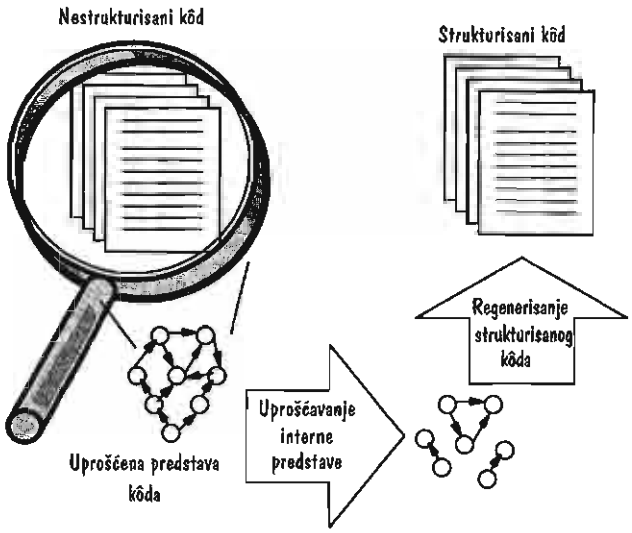


SLIKA 11.11 Redokumentovanje (prilagođeno od Bohnera 1990)

Restrukturisanje

Rekonstrukcija softvera je potrebna da bi se omogućilo njegovo lakše razumevanje i lakši rad na izmenama. Alati koji tome služe interpretiraju izvorni kôd i predstavljaju ga interno. Zatim se pomoću pravila za transformaciju uprošćava interna predstava i rezultati prerađuju u strukturisani kôd. Mada neki alati proizvode samo izvorni kôd, drugi imaju funkcije podrške za generisanje informacija o strukturi, obimu, složenosti i sl. Ta merenja se zatim koriste da bi se odredila održljivost koda i procenili efekti restrukturisanja. Na primer, posle restrukturisanja merenja bi trebalo da pokažu smanjenje složenosti.

Na slici 11.12 ilustrovane su tri glavne aktivnosti u restrukturisanju. Prvo, statička analiza daje informacije koje koristimo da bismo predstavili kôd kao semantičku mrežu ili usmereni graf. Predstava ne mora da bude lako čitljiva za ljude; nju obično koristi samo automatizovani alat.



SLIKA 11.12 Restrukturisanje (prilagođeno iz Bohnera 1990)

Zatim se predstava prečišćava kroz uzastopna uprošćavanja zasnovana na transformacionim tehnikama. Najzad, prečišćena predstava se interpretira i koristi da bi se generisalo ekvivalentno strukturisano telo koda sistema (obično isti kompajler).

Reverzni inženjering

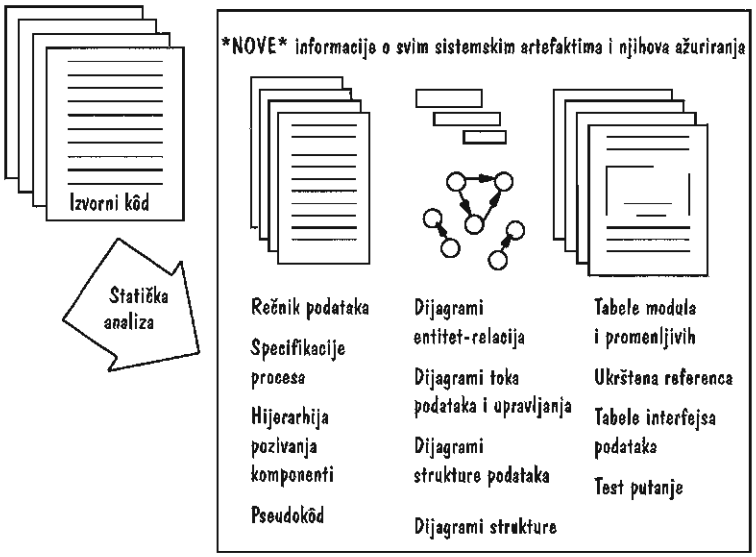
Reverzni inženjering, kao i redokumentovanje, obezbeđuje specifikacione i konstrukcione informacije o softveru iz njegovog izvornog koda. Međutim, reverzni inženjering ide dalje, pokušavajući da obnovi inženjerske informacije na osnovu metoda za specifikaciju i inženjering softvera; te informacije se zatim skladište u obliku koji nam dozvoljava da njima manipulišemo. Izdvojena informacija nije obavezno potpuna, zato što su mnoge komponente izvornog koda pridružene ponekad jednoj komponenti konstrukcije, a ponekad većem broju tih komponenti. Zato sistem koji se dobija reverznim inženjeringom može zaista da ima manje informacija od prvobitnog sistema.

Zahvaljujući grafičkim radnim stanicama i alatima za upravljanje memorijskim uređajima, veliki deo reverznog inženjeringa može da se automatizuje. Možemo da prikazujemo grafičke konstrukcije i manipulišemo njima i da upravljamo skladištem podataka prikupljenih pomoću alata koji koristimo.

Na slici 11.13 prikazan je proces reverznog inženjeringa. Prvo, izvorni kôd se predaje alatu za reverzni inženjering, koji interpretira strukturu i informacije o imenovanju i konstruiše izlaze na vrlo sličan način kao i redokumentovanje. Standardna strukturna analiza i metode projektovanja služe kao dobri komunikacioni mehanizmi za artikulisanje informacija dobijenih reverznim inženjeringom, kao što su rečnici podataka, tok podataka, tok upravljanja i dijagrami entiteti-veze-atributi.

Ključ za reverzni inženjering je njegova sposobnost da apstrahuje informacije iz detaljne implementacije izvornog koda. Međutim, reverzni inženjering nailazi i na značajne prepreke koje onemogućuju njegovo univerzalno korišćenje. Sistemi za rad u realnom vremenu predstavljaju problem; preslikavanje između implementacije i konstrukcije je retko, zbog čestih optimizacija performanse. Drugi problem se javlja kada se izuzetno

složeni sistemi implementiraju pomoću sažetih ili nerazumljivih konvencija imenovanja. Kada se alati za reverzni inženjering primene na takve vrste sistema, informacije koje se dobiju o modelovanju su od ograničene vrednosti.



SLIKA 11.13 Reverzni inženjering (prilagođeno iz Bohnera 1990)

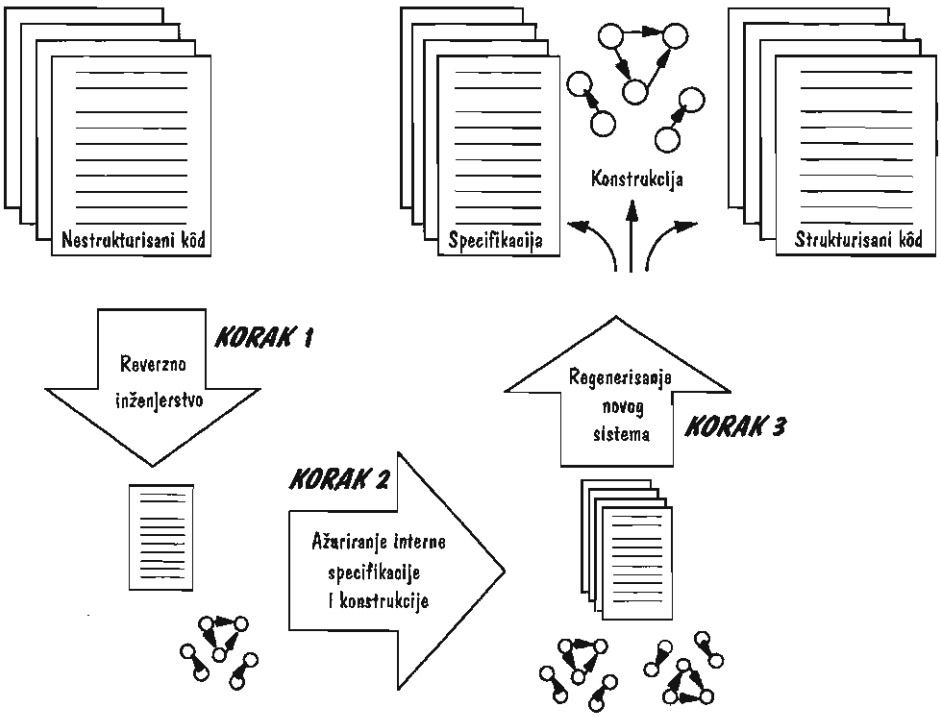
Reverzni inženjering je uspešan kada očekivanja nisu velika. To znači da alati rade dobro prilikom određivanja svih srodnih podataka i poziva određene komponente. Oni mogu da prikažu strukturu složenog sistema i prepoznaju nedoslednosti ili odstupanja od projektnih standarda.

Reinženjering

Reinženjering je proširenje reverznog inženjeringa. Dok reverzni inženjering apstrahuje informacije, reinženjering proizvodi izvorni kôd novog softvera bez menjanja ukupne funkcije sistema. Na slici 11.14 ilustrovani su koraci u tom procesu. Prvo, sistem se podvrgava reverzom inženjeringu i interno se predstavlja na način pogodan za unošenje izmena od strane ljudi ili računara na osnovu postojećih specifikacionih i projektantskih metoda. Zatim se model softverskog sistema ispravlja ili dopunjava. Najzad, nov sistem se generiše na osnovu nove specifikacije ili dizajna.

Ulazi za proces reinženjeringa uključuju datoteke izvornog koda, datoteke baze podataka, datoteke za generisanje ekrana i slične datoteke koje čine sistem. Kada se proces završi, on stvara celokupnu sistemsku dokumentaciju, uključujući specifikaciju i dizajn, kao i nov izvorni kôd.

Kako potpuna automatizacija reinženjeringa nije izvesna u bliskoj budućnosti, proces mora da obuhvati kombinaciju transformacije i ljudske interakcije. Nepotpune predstave možemo da upotpunimo ručno, a iskusan projektant može da poboljša konstrukciju pre nego što se sistem realizuje. U podsetniku 11.8 govori se o radu potrebnom da se izvrši reinženjering sistema i koliko toga mora da se uradi ručno.



SLIKA 11.14 Reinženjering (prilagodeno iz Bohnera 1990)

PODSETNIK 11.8 RAD ULOŽEN U REINŽENJERING

U američkom Nacionalnom institutu za standarde i tehnologiju (NIST) proučavani su rezultati reinženjeringa 13 131 reda izvornih iskaza na jeziku COBOL. Sistem, koji je obuhvatao paketsku obradu bez upotrebe komercijalnih gotovih proizvoda, podvrgnut je reinženjeringu korišćenjem automatske translacije. Ruhl i Gunn (1991) su izvestili da je ukupan rad na reinženjeringu iznosio 35 meseci.

Boehm i saradnici (1995) ističu da je prvobitni COCOMO model izračunao da će biti potrebno 152 meseca rada za reinženjering sistema istog tipa sistema, što je neprihvatljiv nivo preciznosti. Zato je COCOMO II preraden na osnovu drugih studija reinženjeringa, da bi uključio faktor automatske translacije; model proračunava da se automatska translacija događa pri brzini od 2400 izvornih iskaza po mesecu ljudskog rada.

Studija NIST napominje da se količina koda koju je moguće automatski translirati menja sa vrstom aplikacije. Na primer, 96% paketske obrade može da se translira automatski, ali samo 88% aplikacija paketske obrade koje koriste sisteme za upravljanje bazom podataka. Suprotno tome, Ruhl i Gunn (1991) su pronašli da samo polovina interaktivne aplikacije može da se translira automatski.

Budućnost podmlađivanja

S obzirom na to da održavanje softvera nije uvek tako privlačno za praktičare kao što je razvoj novog softvera, pitanja kao što je podmlađivanje ne dobijaju toliku pažnju kao ona koja su u vezi sa novim razvojem. Međutim, neka dostignuća su unela više svetla u podmlađivanje. Komercijalni alati za reverzni inženjering delimično obnavljaju konstrukciju sistema; oni mogu da identifikuju, predstave i analiziraju informacije iz izvornog koda, ali ne mogu da rekonstruišu, snime i izraze konstruktivne apstrakcije koje nisu eksplicitno predstavljene u izvornom kodu.

Informacije iz izvornog koda ne sadrže mnogo podataka o prvobitnoj konstrukciji, pa ostatak mora da se rekonstruiše zaključivanjem. Prema tome, većina uspešnih ishoda u reverznom inženjeringu potiče iz jasno razumljivih stabilnih domena, kao što su informacioni sistemi. Tipičan sistem je tu standardan, jezik (obično COBOL) relativno jednostavan i dobro strukturisan, a postoji i mnogo stručnjaka iz te oblasti.

U drugim oblastima, obnavljanje konstrukcije moguće je samo sa informacijom iz koda, dokumentacijom postojeće konstrukcije, ličnim iskustvom i opštim znanjem o oblasti primene. Potrebni su neformalno lingvističko znanje o domenu problema i idiomima aplikacije da bi se razumeo i rekonstruisao kompletan dizajn. Dakle, podmlađivanje softvera će napredovati kada tehnologije i metodi budu mogli da obuhvate pravila, politike, konstruktivne odluke, terminologiju, konvencije imenovanja i druge neformalne informacije. Kao što ćemo videti u poglavlju 12, analize po ukidanju sistema nam pomažu da zabeležimo te informacije.

U isto vreme, formalizacija konstruktivnih notacija i uvođenje modela za domene proširiće raspoložive informacije donoseći bolje razumevanje i održavanje softverskog sistema. Možemo da očekujemo poboljšanja u transformacionim tehnologijama koje će moći da se primene na više oblasti , a potpunije reprezentacije dozvoliće viši stepen automatizacije u procesu reinženjeringa.

11.7 PRIMER IZ INFORMACIONIH SISTEMA

Ispitajmo odnos softvera Piccadilly prema stvarnom okruženju, da bismo odredili da li je Piccadilly S, P ili E-sistem. Ako bi Piccadilly bio S-sistem, njegov problem bi bio u potpunosti određen; takav sistem je statičan i ne prihvata lako izmenu. Međutim, jasno je da problem sam po sebi može dramatično da se promeni. Na primer, pravila oglašavanja koja nameće britanska vlada mogu da se izmene kada se donesu novi i povuku stari zakoni. Ili može da se promeni strategija formiranja cena u televizijskoj kompaniji, mogu da se sprovedu specijalne promocije da bi se privukli novi oglašivači i sl. Na taj način, softver ne može da bude S-sistem; njegova fleksibilnost bi zahtevala potpuno nov sistem svaki put kada evoluiraju ograničenja stvarnog okruženja.

Ako bi Piccadilly bio P-sistem, njegovo rešenje bi se zasnivalo na apstrakciji problema. Zaista, svaki metod za formiranje cene oglasnih termina zasniva se na modelu različitih karakteristika televizijskog programa, uključujući doba dana, dan u nedelji i broj dodatnih prodatih reklamnih termina. Međutim, P-sistem zahteva stabilnu apstrakciju. To znači da se model ne menja; menjaju se samo informacije o modelu. Jasno je da naš model može da se menja, kada se predlažu nove strategije oglašavanja i kada konkurencija menja svoje modele.

U S i P-sistemima, situacija stvarnog okruženja ostaje stabilna, ali se E-sistem menja kako to čini i stvarni svet; sistem je integralni deo sveta koga modeluje. To je jasno u slučaju softvera Piccadilly, zato što uspeh posebne strategije oglašavanja može u stvari da utiče na sam model. Na primer, pretpostavimo da Piccadilly sledi dosledno svoju prvobitnu strategiju prodaje oglasnog prostora kojom konkurentima otima deo dobiti od oglašavanja petkom od 9 do 11 sati uveče. Direktori televizijske kompanije odlučuju da produže to vreme jer je jasno da tada oglašivači ne odlaze konkurenciji nego oglasne termine kupuju od Piccadillyja. U skladu s tim, kompanija menja svoje cene i otvara promotivnu prodaju oglasnog prostora, što sadašnji model ne podržava. Na primer, daju se posebno povoljne cene oglašavičima koji žele da se njihove komercijalne poruke emituju u 9 sati uveče subotom, kao i u 8 sati uveče petkom. Dakle, postoji stalna interakcija između stvarnog okruženja i apstrakcije; Piccadilly mora da bude E-sistem.

Implikacije za održavanje su posebno jasne u pogledu konstrukcije. Početna konstrukcija Piccadillyja mora da bude veoma fleksibilna i ne sme da se pogoršava sa evolucijom sistema. Štaviše, direktori televizijske kompanije mogu da osnaže Piccadillyjev softver ugrađujući simulacije kako bi se sagledali verovatni efekti predloženih izmena u strategiji.

11.8 PRIMER IZ PRAKSE

Održljivost se često meri pomoću srednjeg vremena između otkaza, pa bi sprečavanje ili ublažavanje otkaza trebalo da bude glavni cilj tima za održavanje. Jedan od načina da se taj cilj postigne jeste da se pažljivo pregleda strategija sistema u pogledu otkaza. To znači da moramo sami da se upitamo koje pretpostavke su napravljene o najboljem načinu rukovanja otkazima i zatim da vidimo da li možemo da promenimo te pretpostavke, da bismo smanjili učestalost otkaza.

Izveštaj istrage posle eksplozije Ariane-5 ističe da su se projektanti usredsredili na ublažavanje posledica slučajnog otkaza. Odnosno, smernica koju je projektantima dao inercijalni referentni sistem (kao ceo softver) bila je da se zaustavi procesor ako se otkrije bilo koji izuzetak. Kada je otkazao inercijalni referentni sistem, to se dogodilo zbog greške u projektovanju a ne zbog slučajnog otkaza, pa je u stvari prestanak obrade bio ispravan postupak sa aspekta specifikacije, ali neispravan sa aspekta misije rakete.

Istražna komisija ističe tu činjenicu u svom izveštaju. Rečeno je sledeće:

> Otkriven je izuzetak, ali je nepravilno opslužen, zato što se zauzelo gledište da bi softver trebalo da se smatra ispravnim sve dok se ne pokaže neispravnim. Komisija ima razloga da veruje da je takvo gledište takođe prihvaćeno i u drugim oblastima projekta softvera Ariane-5. Komisija daje prednost suprotnom gledištu, da bi softver trebalo smatrati neispravnim sve dok primena postojećih prihvaćenih najboljih metoda ne pokaže da je on ispravan (Lions i saradnici, 1996).

Prema tome, kritičan sledeći korak za Arianin softver jeste da se promeni strategija otkaza i implementira niz preventivnih poboljšanja. Zaista, odbor eksplicitno opisuje taj zadatak:

> To znači da softver, koji je kritičan u smislu da njegov otkaz dovodi misiju u rizik, mora da se identifikuje na veoma detaljnom nivou, da izuzetno ponašanje mora da se ograniči i da razumna politika rezervisanja mora da uzme u obzir otkaze softvera (Lions i saradnici, 1996).

Slučaj Ariane-5 nam takođe daje dobar primer poteškoća u upravljanju testiranjem i izmenama. Jasno je da Evropska kosmička agencija ne može da šalje novu raketu svaki put kada treba da se ispita izmena u softveru, pa testiranje izmena tokom održavanja obuhvata složen niz simulacija da bi se procenili efekti novog ili promenjenog koda. Slično tome, imajući u vidu da ima softvera, kao što je inercijalni referentni sistem, koji postoji u više verzija (odnosno, jedan u Ariane-4, a drugi u Ariane-5), moraju se sprovoditi kontrole izmena i upravljanja konfiguracijom, kako bi se obezbedilo da uspešne izmene jedne verzije ne pogoršaju funkcionalnost ili performansu druge verzije. Postojeći inercijalni referentni sistem za Ariane-4 radi kako treba; izmene potrebne za SRI za Ariane-5 mogu da neodgovaraju za Ariane-4.

11.9 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju su vam predstavljeni glavni problemi u održavanju softvera. Videli smo kako održavanje obuhvata i tehničke i ljudske probleme. Serviseri moraju da razumeju ne samo softver, onakav kakav je sada, nego i kakav je bio u prošlosti i kako će evoluirati u budućnosti. Glavne lekcije koje smo razmatrali su sledeće:

- Što je sistem više povezan sa stvarnim okruženjem, to je verovatnije da će se menjati i da će se teže održavati.
- U odnosu na projektante, serviseri imaju više dodatnih poslova. Oni su stalno u interakciji sa kupcima i korisnicima i moraju da razumeju potrebe poslovanja kao i razvoj softvera. Oni treba takođe da budu dobri detektivi, koji detaljno ispituju softver i traže izvore otkaza.
- Merenje održljivosti je teško. Da bismo dobili pravu meru održljivosti, moramo da procenjujemo spoljašnje ponašanje sistema i da pratimo srednje vreme između otkaza. Ali, kasno je čekati da sistem otkaže, pa koristimo unutrašnje atribute koda, kao što su veličina i struktura, da bismo predvideli koji delovi softverskog sistema će, na osnovu istorijskih podataka, verovatno otkazati. Koristimo statičke analizatore koda da nam pomognu u tom procesu.
- Analiza uticaja gradi i prati veze između zahteva, konstrukcije, koda i test slučajeva. Ona nam pomaže da procenimo efekte izmene u jednoj komponenti na druge komponente.
- Tehnike za podmlađivanje softvera su redokumentovanje, restrukturisanje, reverzni inženjering i reinženjering. Opšti cilj je da se skrivene informacije učine eksplicitnim, tako da možemo da ih koristimo da bismo poboljšali dizajn i strukturu koda. Mada potpuno podmlađivanje nije verovatno u bliskoj budućnosti, ono se uspešno koristi u domenima koji su zreli i dobro shvaćeni, kao što je informaciona tehnologija.

11.10 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Održavanje je definitivno timska aktivnost. Mora se obezbediti mnogo koordinacije kada se komponenta proverava, menja i ispituje i kada se prerađena komponenta vraća u sistem koji radi. Štaviše, mnogi otkazi su rezultat složenih interakcija između komponenti,

tako da morate da komunicirate sa članovima svog tima da biste dobili ukupnu sliku kako softver sarađuje sa svojim okruženjem.

Znanja vaših ljudi su posebno važna za vreme održavanja. Kada tražite uzrok problema, morate da razgovarate sa svojim kolegama, korisnicima i kupcima, a svaki od njih će imati različit stil rada (kao što smo videli u poglavlju 2). Zato morate da naučite kako da izdvojite informaciju koja vam je potrebna iz dokumenata i od ljudi na najefikasnije moguće načine, koristeći svoje razumevanje stilova rada.

11.11 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Održavanje je zrelo područje za istraživanje. Mnoge aktivnosti održavanja bi mogle da se učine lakšim ili efikasnijim ako bismo bili bolji u predviđanju verovatnih izvora neispravnosti. Istraživači tragaju za boljim načinima merenja održljivosti, na osnovu informacija o proizvodu; oni razvijaju nove modele da bi nam pokazali međusobne veze između proizvoda, procesa i resursa. Slično tome, modeli će nam pomoći da saznamo koliko rada je potrebno uložiti da bi se održavao sistem i kada je celishodno povući ili podmladiti nasleđeni sistem.

Rad se nastavlja izgradnjom alata koji nam pomažu u procesu održavanja. Alati za reinženjering, kontrolu izmena, repozitorijume upravljanja konfiguracijom i baze podataka sa istorijom konkretnog projekta verovatno će postati usavršenije kada istraživači izgrade prototipove zasnovane na empirijskim podacima.

Najzad, istraživači će nastaviti da tragaju za opštim zakonima održavanja, kao što je nagovestio Lehman (1980). Oni su željni da saznaju da li teorija softverskog inženjerstva može da potvrdi ono na što empirijska posmatranja ukazuju u praksi: naime, da je evolucija softverskih sistema dosledna i predvidljiva.

11.12 SEMINARSKI RAD

Ispitajte sve svoje artefakte (zahteve, konstrukciju, kôd, planove testiranja, dokumentaciju) za *Aranžer pozajmica*. Koliko su oni pogodni za održavanje? Ako biste opet morali da projektujete i implementirate *Aranžer pozajmica*, šta biste uradili drugačije da bi projekat bilo lakše održavati?

11.13 KLJUČNE REFERENCE

Nema mnogo savremenih knjiga o održavanju softvera; većina informacija se nalazi u časopisima i zbornicima radova sa konferencija. Izdanje *IEEE Software* iz januara 1990. godine za temu ima održavanje, reverzni inženjering i obnavljanje konstrukcije; izdanje iz januara 1995. godine je usredsređeno na nasleđene sisteme, a u izdanju iz januara 1993. godine postoji dobar članak koji su napisali Wilde, Matthews i Huitt o specijalnim problemima održavanja objektno orijentisanih sistema. Izdanje časopisa *Communications of the ACM* iz maja 1994. godine je specijalan broj posvećen reverznom inženjeringu. *Software Maintenace: Research and Practice* je časopis potpuno posvećen pitanjima održavanja.

IEEE Computer Society Press nudi nekoliko dobrih uputstava o temama koje se odnose na održavanje, uključujući jedno o reinženjeringu softvera od Arnolda (1993) i drugo o analizi uticaja, od Bohnera i Arnolda (1996).

Samuleson (1990) istražuje legalne implikacije reverznog inženjeringa, i postavlja pitanje da li je takva praksa ekvivalentna krađi tuđih ideja.

Međunarodna konferencija o održavanju softvera održava se svake godine pod sponzorstvom organizacija IEEE i ACM. Zbornike radova sa prethodnih konferencija možete da naručite od IEEE Computer Society Press, a informacije o sledećoj konferenciji na temu održavanja potražite na veb lokaciji Computer Society.

11.14 VEŽBE

1. Kategorišite sledeće sisteme kao S-, P- ili E-sisteme. Za svaki od njih, objasnite zašto pripada toj kategoriji. Identifikujte one aspekte sistema koji mogu da se menjaju.
   (a) sistem za kontrolu vazdušnog saobraćaja
   (b) operativni sistem za mikroračunar
   (c) sistem za ubrzavanje aritmetike u pokretnom zarezu
   (d) sistem za upravljanje bazom podataka
   (e) sistem za pronalaženje prostih činilaca brojeva
   (f) sistem za pronalaženje prvog prostog broja većeg od datog broja
2. Objasnite zašto visok stepen sprezanja između komponenti može da veoma oteža održavanje.
3. Objasnite zašto uspeh sistema mnogo zavisi od kvaliteta dokumentacije koja se napravi u toku razvoja sistema.
4. Neka od predavanja iz računarskih nauka obuhvataju izradu seminarskog rada koji počinje kao mali sistem i stalno se poboljšava dok se ne dobije potpun rezultat. Ako ste radili na takvom projektu, pregledajte svoje beleške. Koliko vremena je potrošeno na definisanje i razumevanje problema? Koliko vremena je potrošeno za implementiranje koda? Uporedite kategorije iz tabele 11.2 sa vremenom koje ste dobili za vaš projekat i komentarišite da li su razlike dobre ili loše.
5. Objasnite zašto programiranje za održavanje može da bude izazovnije od novog razvoja. Zašto dobar programer održavanja mora da bude dobar u radu s ljudima? Koje su druge poželjne karakteristike programera za održavanje?
6. Ispitajte veliki program iz jednog od vaših školskih projekata. Kako morate da mu dodate dokumentaciju da bi neko drugi mogao da ga održava? Raspravljajte o razlozima u korist i protiv pisanja dodatne dokumentacije za gotov program.
7. Pozajmite kopiju velikog programa (više od 1 000 redova koda) od prijatelja. Pokušajte da izaberete program koji uopšte ne poznajete. Koliko je korisna dokumentacija? Uporedite kôd sa dokumentacijom; koliko je tačna dokumentacija? Ako bi vam bilo dodeljeno da održavate taj program, koju dodatnu dokumentaciju biste voleli da vidite? Kako veličina programa utiče na vašu sposobnost da ga održavate?
8. Kao u prethodnom problemu, ispitajte prijateljev program. Pretpostavimo da želite da napravite izmenu u kodu i da morate da izvedete regresiono testiranje rezultata kako biste se uverili da program i dalje ispravno radi. Da li su vam na raspolaganju podaci testiranja i scenariji testiranja? Raspravljajte o potrebi čuvanja podataka iz formalnih testova i scenarija za potrebe održavanja.

9. Objasnite zašto testiranje komponenti sa jednim ulazom i jednim izlazom olakšava testiranje u procesu održavanja.
10. Pregledajte karakteristike dobrog projektovanja softvera. Za svaku od njih objasnite da li će pomoći ili otežati podmlađivanje softvera.
11. Da li je softver Ariane-5 S-, P- ili E-sistem?
12. Da li McCabeov ciklomatski broj dozvoljava da rangiranje komponenti prema kvalitetu? Odnosno, možemo li uvek da kažemo da je jedna komponenta složenija od druge? Navedite neke aspekte složenosti softvera koji nisu obuhvaćeni ciklomatskim brojem.
13. Pretpostavite da održavate veliki softverski sistem koji je veoma važan za bezbednost. Koristite, na primer, Porterov i Selbyjev model da biste predvideli koja od komponenti će najverovatnije da otkaže. Zatim pažljivo ispitujete te identifikovane komponente i sprovodite usavršavajuće i preventivno održavanje svake od njih. Uskoro sistem doživljava katastrofalan otkaz, sa ozbiljnim posledicama po život i imovinu. Ispostavilo se da je uzrok otkaza komponenta koju vaš model nije identifikovao. Da li ste krivi što niste pregledali i druge komponente?
14. Sledi lista funkcionalnih kriterijuma verzije i kontrole konfiguracije za alate za upravljanje konfiguracijom agencije British. Objasnite kako svaki od dole navedneih činilaca doprinosi lakšem održavanju.
    (a) Zapisivanje verzije ili reference verzija.
    (b) Izvlačenje svake verzije po zahtevu.
    (c) Zapisivanje odnosa.
    (d) Zapisivanje odnosa između verzija do kojih alat kontroliše pristup i onih za koje to ne čini.
    (e) Kontrola bezbednosti i evidentiranje ovlašćenja.
    (f) Zapisivanje izmene datoteke.
    (g) Zapisivanje statusa verzije.
    (h) Pomoć u konfigurisanju verzije.
    (i) Odnos sa alatom za kontrolu projekta.
    (j) Proizvodnja izveštaja.
    (k) Kontrola izdanja.
    (l) Kontrola samog sebe.
    (m)Arhiviranje i izvlačenje retko korišćenih datoteka.

12 Procenjivanje proizvoda, procesa i resursa



U ovom poglavlju obradivaćemo:
- analizu svojstava, studije slučajeva, ankete i eksperimente;
- merenje i vrednovanje;
- zrelost sposobnosti, ISO 9000 i druge modele procesa;
- zrelost ljudi;
- procenu produkata razvoja;
- povraćaj ulaganja.

U prethodnim poglavljima, govorili smo o različitim aktivnostima vezanim za razvoj i održavanje sistema zasnovanih na softveru. Primeri iz računarstva i državne uprave pokazali su nam da projektanti softvera koriste vrlo različite metode i alate da bi otkrili i definisali zahteve, projektovali i implementirali sisteme, ispitivali i održavali ih u toku njihove evolucije. Kako doneti odluku o izboru tehnike ili alata? Kako proceniti delotvornost i efikasnost postojećeg načina rada, da bismo mogli da utvrdimo da poboljšavamo sistem? Kada je, u datoj situaciji, jedna tehnika bolja od neke druge? Kako pokazujemo da naši proizvodi, procesi i resursi poseduju željene karakteristike (kao što je kvalitet)? U ovom poglavlju, istražujemo tehnike za procenu proizvoda, procesa i resursa; u sledećem poglavlju prikazaćemo dokumentovane primere poboljšanja, zasnovanog na primeni tih tehnika.

12.1 PRISTUPI PROCENI

Kao profesionalci, skloni smo da procenjujemo vlastite proizvode i načine njihove izrade. Tehnike za procenu koje koristimo slične su onima u drugim disciplinama: merimo ključne aspekte proizvoda, procesa i resursa i te informacije koristimo za određivanje ispunjenosti postavljenih ciljeva u pogledu produktivnosti, performanse, kvaliteta i drugih

poželjnih atributa. Ali, postoji više vrsta izučavanja i važno je da se razume koja od njih su najprikladnija za ono što želimo saznati.

Tehniku za procenu možemo da svrstamo u jednu od sledeće četiri kategorije:

1. analiza svojstava
2. anketa
3. studija slučaja
4. formalni eksperiment

Analiza svojstava

Najjednostavnija vrsta procene je **analiza svojstava**, koja se koristi za procenu i rangiranje atributa različitih proizvoda s ciljem da se izabere adekvatan alat ili metod. Na primer, može da nas zanima kupovina alata za projektovanje i zato formiramo listu pet ključnih atributa koje bi naš alat trebalo da poseduje:

1. dobar korisnički interfejs
2. podrška objektno orijentisanom dizajnu
3. provera konsistentnosti
4. podrška slučajevima korišćenja
5. radi pod operativnim sistemom UNIX

Nakon toga, identifikujemo tri moguća alata i svaki od kriterijuma ocenjujemo od 1 (ne zadovoljava) do 5 (zadovoljava u potpunosti). Zatim analiziramo rezultate, možda stvarajući ukupan rezultat na osnovu značaja pojedinačnih kriterijuma, kao što je prikazano u tabeli 12.1. (Množimo koeficijent važnosti sa ocenom kriterijuma za svaki pojedinačni kriterijum, a zatim sabiramo.) Najzad, na osnovu rezultata, biramo t-OO-1 kao alat za projektovanje.

Analiza svojstava je uvek veoma subjektivna, a ocene odražavaju sklonosti ocenjivača. Ona je korisna za sužavanje izbora mogućih alata, ali u stvari ne procenjuje ponašanje u smislu uzroka i posledice. Na primer, analiza svojstava uopšte nije korisna pri izboru najpogodnije tehnike projektovanja koja treba da pomogne u izgradnji potpunih i konsistentnih projektnih rešenja; u tom slučaju radimo kontrolisane studije da bismo mogli da razumemo uzrok i posledicu.

TABELA 12.1 Rangiranje projektantskih alata

Svojstvo	Alat 1: t-OO-1	Alat 2: ObjecTool	Alat 3: EasyDesign	Značaj
Dobar korisnički interfejs	4	5	4	3
Podrška objektno orijentisanom dizajnu	5	5	5	5
Provera konsistentnosti	5	3	1	3
Podrška slučajevima korišćenja	4	4	4	2
Radi pod operativnim sistemom UNIX	5	4	5	5
Rezultat	85	77	73	

Ankete

Anketa je retrospektivna studija koja pokušava da dokumentuje odnose i ishode u razmatranoj situaciji. Ankete se često koriste u društvenim naukama, gde se ispituju stavovi da bi se odredilo kakav je odnos stanovništva u pogledu nekog posebnog skupa pitanja, ili demografske ankete stanovništva sa ciljem utvrđivanja trendova i odnosa. Ankete u softverskom inženjerstvu su slične po tome što evidentiramo podatke da bismo odredili kako su učesnici u projektu reagovali na određeni metod, alat, ili tehniku, ili da bismo odredili trendove i odnose. Možemo takođe da registrujemo informacije vezane za proizvode ili projekte, dokumentujemo veličinu komponenti, broj grešaka, utrošen rad itd. Na primer, možemo da poredimo podatke iz anketa objektno orijentisanih projekata sa podacima iz proceduralnih projekata, da bismo videli da li postoje značajne razlike.

Kada sprovodimo ankete obično nemamo kontrolu nad posmatranom situacijom. S obzirom na to da je anketa retrospektivna studija, zapisujemo informacije o situaciji i poredimo ih sa sličnima, ali ne možemo da manipulišemo promenljivima; za to su nam potrebne studije slučajeva i eksperimenti.

Studije slučajeva

Ni studije slučajeva ni formalni eksperimenti obično nisu retrospektivni. Unapred odlučujemo šta želimo da istražujemo i zatim planiramo kako da dođemo do podataka da bismo podržali istraživanje. U **studiji slučaja**, identifikujemo ključne činioce koji mogu da utiču na ishod neke aktivnosti i zatim ih dokumentujemo: ulaze, ograničenja, resurse i izlaze. Suprotno od toga, **formalni eksperiment** je strogo kontrolisano istraživanje u kome se ključni činioci aktivnosti identifikuju i njima se manipuliše da bi se dokumentovali njihovi uticaji na ishod.

I studija slučaja i eksperiment se sastoje iz niza koraka: koncepcije, postavljanja hipoteze, projekta, pripreme, izvršenja, analize, dostavljanja rezultata i odlučivanja. Postavljanje hipoteze je posebno značajno, budući da usmerava objekte merenja i način analiziranja rezultata. Projekti koje izdvajamo radi uključenja u studiju moraju da se pažljivo biraju da bi ilustrovali ono što je tipično za organizaciju ili kompaniju.

U studiji slučaja obično se poredi jedna situacija s drugom: na primer, rezultati korišćenja jedne metode ili alata s rezultatima upotrebe drugih. Da bismo izbegli pristrasnost i obezbedili da ispitujemo odnos koji smo postavili u hipotezi, možemo da organizujemo studiju na jedan od tri načina: kao sestrinski (bliski) projekat, baznu liniju ili slučajan izbor.

Da biste shvatili razliku između ove tri vrste studija slučajeva, posmatrajmo sledeći primer. Odlučujete da izvedete studiju slučaja da biste procenili efekte upotrebe nove tehnike inspekcije. Za izvođenje takve studije, birate dva projekta koji se zovu **sestrinski projekti**, od kojih je svaki tipičan za organizaciju i ima slične vrednosti nezavisno promenljivih koje su predmet merenja. Na primer, projekti mogu da budu slični u smislu domena primene, implementacionog jezika, tehnike specifikacije i metoda projektovanja. Zatim, vršite inspekcije na postojeći način u prvom projektu i novi način u drugom projektu. Izborom što je moguće sličnijih projekata kontrolišete najviše što možete. Ta situacija vam dozvoljava da dovodite u vezu svaku razliku u rezultatu sa razlikom u tehnici inspekcije.

Međutim, ako ne možete da pronađete dva dovoljno slična projekta da budu sestrinski projekti, možete da poredite svoju novu tehniku inspekcije sa opštom **baznom linijom**. Ovde vaša kompanija ili organizacija sakuplja podatke iz različitih projekata bez obzira

na njihove sličnosti ili razlike. Pored informacija o promenljivama koje smo ranije pomenuli, podaci mogu da uključe i opisne mere, kao što su veličina proizvoda, uloženi rad, broj otkrivenih grešaka itd. Zatim možete da proračunate mere centralne tendencije i rasipanja podataka u bazi podataka, tako da dobijete uvid u „prosečnu" situaciji koja je tipična za vašu kompaniju. Vaša studija slučaja obuhvata upotpunjavanje projekta najpre korišćenjem nove tehnike inspekcije i zatim poređenjem rezultata sa baznom linijom. U nekim slučajevima, možete biti u stanju da iz baze podataka organizacije izaberete podskup projekata koji su slični onom koji koristi novu tehniku inspekcije; podskup i ovde dodaje izvestan stepen kontrole vašoj studiji, dajući vam viši stepen poverenja u to da je svaka razlika u rezultatu posledica razlike u tehnici inspekcije.

Nemamo uvek taj luksuz da pronađemo dva više projekata za proučavanje, posebno kada ispitujemo prvu upotrebu nove tehnike ili alata. U tom slučaju, možemo da upotrebimo **slučajan izbor** da bismo izdelili jedan projekat na delove, gde jedan deo koristi novu tehniku, a drugi ne. Ovde studija slučaja zahteva dosta kontrole, zato što pri analizi koristimo prednost slučajnosti i replikacije. Međutim, to nije formalni eksperiment, zato što projekat nije slučajno odabran između ostalih u kompaniji ili organizaciji. Ovde jedino komponente koda slučajno podvrgavamo bilo staroj bilo novoj tehnici inspekcije. Slučajnost nam pomaže da smanjimo eksperimentalnu grešku i uravnotežimo naše zavisne činioce (to jest, one činioce čiji rezultati utiču jedni na druge).

Slučajan izbor je posebno koristan u situacijama gde metoda koja se proučava može da poprimi više različitih vrednosti. Na primer, možemo poželeti da odredimo da li vreme pripreme utiče na efikasnost inspekcije. Evidentiramo vreme pripreme kao i veličinu komponente i otkrivene greške. Na osnovu toga možemo da ispitujemo da li produžena priprema rezultira većom brzinom otkrivanja grešaka.

Formalni eksperimenti

Formalni eksperimenti su najkontrolisanija vrsta studije i o njima se detaljno govori u radu Fentona i Pfleegera (1997). U **formalnom eksperimentu**, manipulišemo vrednostima nezavisno promenljivih i posmatramo promene u zavisnim promenljivim da bismo odredili kako promene ulaza utiču na izlaz. Na primer, možemo da ispitujemo efekat alata ili tehnike na kvalitet proizvoda ili produktivnost programera; ili možemo da pokušamo da otkrijemo odnos između vremena pripreme i efikasnosti inspekcije.

U formalnom eksperimentu koristi se više metoda da bi se eliminisala međudejstva i odstranili zavisni činioci, tako da uzrok i posledica mogu da se procene sa izvesnom sigurnošću. Na primer, slučajnost se koristi da bi se osiguralo da izbor subjekata eksperimenata ni na koji način nije uslovljen tehnikom izbora. Često merimo ponovljene primere aktivnosti, tako da višestruki skupovi podataka povećavaju naše poverenje u dobijeni rezultat. Drugim rečima, kada vidimo da nešto prouzrokuje isti efekat više puta umesto samo jednom, sigurniji smo da je efekat posledica uzroka, a ne slučajnosti.

Formalni eksperimenti se pažljivo projektuju da bi razmatrani primeri bili što reprezentativniji. Na primer, ako na projektu imamo početnika, poznavaoca i stručnjaka i poredimo dve tehnike, eksperiment projektujemo tako da sve tri vrste programera koriste svaku od te dve tehnike; nakon toga poredimo rezultate svih šest kombinacija programera i tehnike.

Priprema za procenu

Bez obzira na izbor vrste procene, postoji više ključnih koraka koje treba sprovesti da bismo se uverili da smo fokusirani i da možemo identifikovati odgovarajuće promenljive.

Postavljanje hipoteze. Odluku o tome šta želimo da istražujemo izražavamo kao hipotezu koju želimo da ispitamo. To znači da moramo tačno odrediti šta je to što želimo da saznamo. **Hipoteza** je probna teorija ili pretpostavka za koju mislimo da objašnjava ponašanje koje želimo da istražimo. Na primer, naša hipoteza može da bude sledeći iskaz:

Upotreba metode čiste sobe proizvodi softver boljeg kvaliteta nego upotreba metode SSADM.

Bilo da ispitujemo zapise iz prošlosti da bismo procenili šta se događalo kada je određena grupa koristila svaku od metoda (anketa), procenjujemo „snimak" naše organizacije kada koristi čistu sobu (studija slučaja) ili da vršimo pažljivo kontrolisano poređenje onih koji koriste čistu sobu sa onima koji koriste SSADM (formalni eksperiment), u stvari želimo da vidimo da li će podaci koje prikupljamo potvrditi ili opovrgnuti postavljenu hipotezu.

Gde god je to moguće, hipotezu postavljamo koristeći izraze koji se mogu kvantifikovati, tako da se može lako reći da li je hipoteza potvrđena ili odbačena. Na primer, „kvalitet" možemo da definišemo pomoću broja otkrivenih grešaka i da ponovo postavimo hipotezu poput ove:

Kôd koji se proizvodi korišćenjem metode čiste sobe ima manji broj grešaka na hiljadu redova koda od onog koji se proizvodi upotrebom metode SSADM.

Kvanitifikovanje hipoteze često dovodi do upotrebe metoda zamene. To znači, da bismo identifikovali činilac ili aspekt koji želimo da merimo (na primer kvalitet), moramo ga meriti indirektno, koristeći nešto što je pridruženo tom činiocu (na primer greške). S obzirom na to da je zamena indirektna mera, postoji opasnost da promena u zameni nije ista kao promena u prvobitnom činiocu. Na primer, greške (ili njihovo nepostojanje) mogu da ne odražavaju tačno kvalitet softvera: pronalaženje velikog broja grešaka za vreme ispitivanja može da znači da je test bio vrlo iscrpan i da je rezultujući proizvod skoro bez grešaka, ili može da znači da je razvoj bio nemaran i da je u proizvodu verovatno ostalo jošmnogo grešaka. Slično tome, isporučen broj redova koda ne mora uvek tačno da odražava rad uložen u proizvod, zato što ova mera ne uzima u obzir korišćenje postojećeg koda ili izradu prototipova. Stoga uz hipotezu koja se može kvantifikovati, dokumentujemo i odnos između mera i činilaca koje one odražavaju. Posebno, koristimo one kvantitativne termine koji su što je moguće direktniji i nedvosmisleniji.

Održavanje kontrole nad promenljivama. Jednom kada imamo eksplicitnu hipotezu, moramo da odlučimo koje promenljive mogu da utiču na njenu istinitost. Zatim, za svaku identifikovanu promenljivu, odlučujemo koliko možemo da je kontrolišemo. Na primer, ako istražujemo efekat metode projektovanja na kvalitet rezultujućeg softvera, ali nemamo nikakvu kontrolu nad onim ko koristi metodu projektovanja, tada pravimo studiju slučaja da bismo dokumentovali rezultate. Eksperimenti se izvode samo ako možemo direktno da manipulišemo ponašanjem, precizno i sistematski. Dakle, ako možemo da kontrolišemo ko koristi metodu čiste sobe, ko koristi SSADM i kada i gde se one koriste, onda možemo da izvedemo eksperiment. Ova vrsta manipulacije može da se uradi u formi „igre", gde se organizuju događaji da bi se simuliralo njihovo pojavljivanje u stvarnom okruženju, ili situaciji „na terenu", gde se događaji nadgledaju kada se stvarno dešavaju.

U eksperimentu uzorkujemo nezavisno promenljive, tako da možemo da predstavimo sve moguće slučajeve. Ali, u studiji slučaja, uzorkujemo iz promenljivih, birajući vrednosti koje su tipične za razmatranu organizaciju i njene projekte. Na primer, u eksperimentu koji odražava efekte jezika, izabrali bismo skup projekata sa ciljem da obuhvatimo što je moguće više jezika; suprotno tome, studija slučaja bi mogla da obuhvati izbor jezika koji se najviše koristi u projektima u sklopu organizacije.

Davanje smisla istraživanju. Postoji mnogo oblasti softverskog inženjerstva koje mogu da se analiziraju korišćenjem anketa, studija slučajeva i eksperimenata. Jedan od ključnih motiva za korišćenje formalnog eksperimenta, a ne studije slučaja ili ankete je taj što se rezultati eksperimenta obično mogu više uopštavati. Odnosno, ako koristimo anketu ili studiju slučaja da bismo razumeli šta se dešava u određenoj organizaciji, rezultati mogu da se primene samo na tu organizaciju (i možda na one organizacije koje su joj veoma slične). Ali, imajući u vidu da se formalni eksperiment pažljivo kontroliše i da se tu upoređuju različite vrednosti kontrolisanih promenljivih, njegovi rezultati su opšte primenljivi na širu zajednicu i više organizacija. Važno je zapamtiti da ne možemo sve da kontrolišemo; eksperimenti u softverskom inženjerstvu nisu kao eksperimenti u biologiji ili hemiji. Moramo da uzmemo u obzir ograničenja i nedostatak kontrole kada odlučujemo da li su rezultati studije primenljivi u novoj situaciji.

12.2 IZBOR TEHNIKE ZA PROCENU

Kitchenham, Pickard i Pfleeger (1995) napominju da se razlike između metoda istraživanja često odražavaju na njihov opseg. Kako zahtevaju dosta kontrole, formalni eksperimenti po svojoj prirodi teže da budu mali, obuhvatajući mali broj ljudi ili događaja. O eksperimentima možemo da razmišljamo kao o „istraživanju u malom". Studije slučajeva obično pre traže tipičan projekat, nego što pokušavaju da dođu do informacija o svim mogućim slučajevima; o njima može da se misli kao o „istraživanju tipičnog". Ankete pokušavaju da prebroje šta se dešava na širokom planu u velikim grupama projekata: to je „istraživanje u velikom".

Ključni činioci izbora

Nekoliko opštih smernica može da nam pomogne pri odlučivanju da li da sprovedemo anketu, studiju slučaja ili formalni eksperiment. Kao što smo videli, kontrola je ključni element za našu odluku. Ako imamo visok nivo kontrole nad promenljivim koje mogu da utiču na ishod, onda razmatramo eksperiment. Ako nemamo takvu kontrolu, tehnika studije slučaja je bolji izbor. Ali, nivo kontrole zadovoljava samo tehnički aspekt; moramo takođe da razmatramo i praktična pitanja. Veoma teško se može izvesti da se kontrolišu promenljive, s jedne strane zbog velikih troškova koje bi tako nešto proizvelo, a s druge zbog stepena rizika. Na primer, sistemi kritično po bezbednost mogu da nose visok rizik pri eksperimentisanju, a studija slučaja da bude izvodljivija.

Kitchenham, Pickard i Pfleeger (1995) ističu da je formalni eksperiment posebno koristan u istraživanju alternativnih metoda izvođenja pojedinačnog samostalnog zadatka; na primer, možete da izvedete eksperiment da biste odredili da li je VDM bolji od dijagrama stanja za specifikovanje skupa zahteva. Ovde samostalan zadatak može da se izoluje od ostatka procesa razvoja, ali se zadatak i dalje ugrađuje u uobičajen način na

koji se razvija kôd. Takođe, samostalan zadatak može odmah da se proceni, tako da eksperiment ne usporava završetak projekta. S druge strane, studija slučaja može da bude bolji izbor od formalnog eksperimenta, ako su promene procesa prouzrokovane nezavisnim promenljivim u širokom opsegu, što zahteva da se efekti mere na visokom nivou i preko mnogo zavisnih promenljivih koje treba da se kontrolišu i mere.

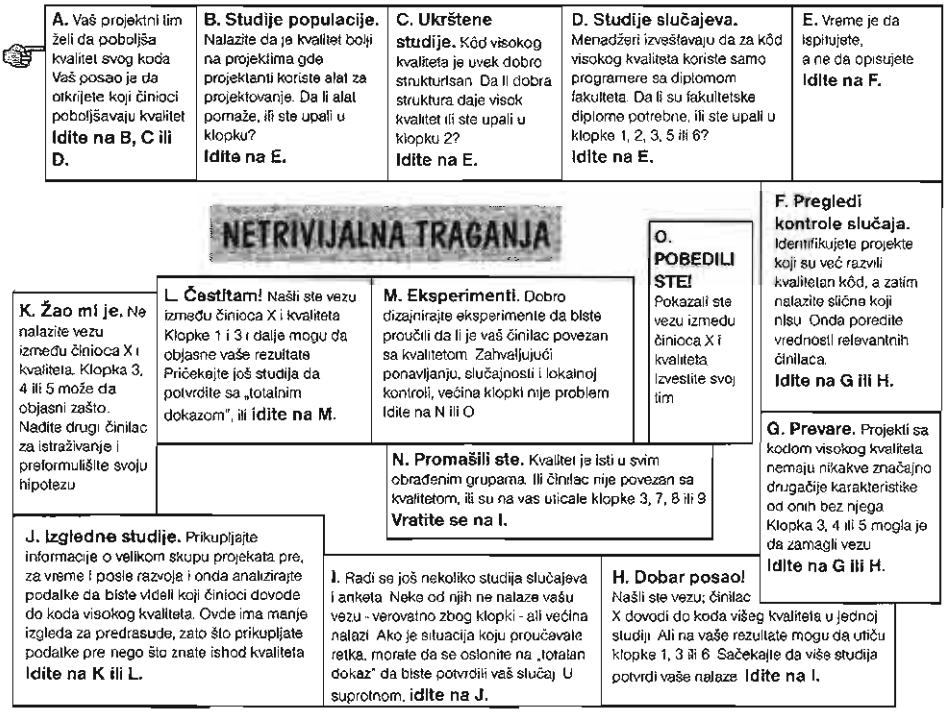
Sledeće o čemu treba razmišljati jeste stepen u kome možemo da ponavljamo osnovnu situaciju koju istražujemo. Na primer, pretpostavimo da želimo da istražimo efekte jezika na rezultujući softver. Možemo li da razvijemo isti projekat više puta, koristeći svaki put drugačiji jezik? Ako ponavljanje nije moguće, tada ne možemo da napravimo formalni eksperiment. Međutim, čak i kada je ponavljanje moguće, ako je skupo nećemo ga ponavljati. Na primer, ako ponavljanje u studiji koju želimo da izvedemo nije skupo, onda je podesnije raditi eksperiment nego studiju slučaja. Slično tome, ako nemamo nikakvu kontrolu (odnosno, veoma ju je teško uspostaviti), onda bi trebalo da razmotrimo studiju slučaja.

U šta da verujemo

Izveštaji o istraživanju sadrže zaključke studija slučaja, anketa i formalnih eksperimenata. Ali, nije uvek lako utvrditi koji rezultati su primenljivi na vaše okolnosti. Na primer, razmotrite odluku da pređete sa COBOL-a na jezik četvrte generacije (4GL), onu koja nije uvek tako jasna kao što izgleda. U 1980-im godinama, u više zanimljivih studija, izvršeno je poređenje upotrebe COBOL-a sa više 4GL-ova radi implementiranja relativno jednostavnih aplikacija u poslovnim sistemima: Misra i Jalics (1988), Verner i Tate (1988) i Matos i Jalics (1989). Rezultati tih studija bili su zadivljujući, ali kontradiktorni. Neki su pokazali poboljšanja produktivnosti sa 4GL-om od 4 do 5 puta, dok je kod drugih poboljšanje bilo samo 20% do 39%. Neki su pokazali da se performansa objektnog koda pogoršala za 15 do 174 puta za 4GL, ali u nekim slučajevima se dogodilo suprotno: 4GL jezici su proizveli kôd koji je bio šest puta brži od ekvivalentnog na COBOL-u!

Kako da znamo kojoj studiji da verujemo kada su rezultati kontradiktorni? Možemo da postavimo niz pitanja koristeći tablu za igru na slici 12.1, da bismo shvatili kako da razvrstavamo te studije. Odgovori na pitanja vam kažu kada imate dovoljno informacija da izvučete valjan zaključak o odnosu između činilaca. Da biste počeli, pretpostavite da se vaš projektni tim zanima za poboljšanje kvaliteta koda koji proizvodi. Želite da odredite koji činioci poboljšavaju kvalitet, kako bi vaš tim mogao da upotrebi odgovarajuće tehnike ili alate da bi proizvodio bolji kôd. Prvo, odlučujete da merite kvalitet brojeći greške na hiljadu redova koda; zatim, odlučujete da je vrlo kvalitetan sistem onaj koji ima manje od pet grešaka na hiljadu redova koda. Potom pokušavate da pronađete šta utiče na kvalitet koda analizirajući rezultate ispitivanja karakteristika velikih populacija projektanata da biste promenljive doveli u korelaciju. Na primer, čitate o anketi u drugoj organizaciji, gde se izveštava da se kvalitet koda poboljšava kada projektanti koriste projektantske alate. Kako znate da je ovaj rezultat validan? Moguće je da je studija pala u klopku 1, kao što je prikazano u tabeli 12.2.

Klopka 1 je zbunjujuća zato što nije moguće reći koji od dva činioca prouzrokuje rezultate koji se posmatraju. Na primer, ako novi programeri nikada ne koristi projektantske alate, a iskusni programeri to uvek čine, nije moguće reći da li kôd iskusnih programera uvek ima manje grešaka na terenu zbog korišćenja alata ili zbog njihovog iskustva. Dakle, kada su činioci međuzavisni, moguće je da je onaj drugi taj koji poboljšava kvalitet. Ako ne možete da prevladate međuzavisnost, morate da uradite pažljiviju studiju, što je na slici 12.1 predstavljeno u polju E.



SLIKA 12.1 Istraživanje i procena (prilagođeno sa dozvolom Liebmana 1994)

Međutim, pretpostavimo sada da analizirate ukrštenu studiju (polje C); to znači da birate reprezentativan uzorak projekata ili proizvoda za ispitivanje. Na primer, razmatrajući ukrštanje koda koji je napravila vaša organizacija, nalazite da je visokokvalitetan kôd uvek dobro strukturisan. Da li je dobra struktura uzrok dobrog kvaliteta, ili ste pomešali uzrok sa posledicom (klopka 2)? U stvari, dobra struktura bi mogla da bude rezultat neke druge akcije (kao što je upotreba projektantskih alata ili editora), a ne uzrok sam po sebi. Dakle, morate da uradite pažljiviju studiju, okrećući se od opisnih istraživanja ka onima koja ispituju hipoteze.

TABELA 12.2 Uobičajene klopke u proceni (prilagođeno sa dozvolom Liebmana 1994)

Klopka	Opis
1. Međudejstva	Drugi činilac prouzrokuje efekat.
2. Uzrok ili posledica?	Činilac bi mogao da bude rezultat, a ne uzrok postupka.
3. Slučajnost	Postoji uvek mala mogućnost da se vaš rezultat dogodio slučajno.
4. Homogenost	Ne nalazite vezu zato što su svi subjekti imali isti nivo činioca.
5. Pogrešna klasifikacija	Ne nalazite vezu zato što ne možete tačno da klasifikujete nivo činioca svakog subjekta.
6. Polarizacija	Procedure izbora ili administracija studije nehotično polarizuju rezultat.
7. Suviše kratko	Kratkoročni efekti su različiti od dugoročnih.
8. Pogrešna količina	Činilac bi imao efekta, ali ne u količini upotrebljenoj u studiji.
9. Pogrešna situacija	Činilac ima željeni efekat, ali ne u situaciji koja se proučava.

Alternativno, pretpostavite da ste u literaturi pronašli studiju slučaja koja nagoveštava da se na projektima sa visokokvalitetnim kodom koriste samo programeri sa fakultetskim diplomama. Pre nego što zaključite da je fakultetska diploma potrebna u vašoj organizaciji, morate da odlučite da li je taj rezultat posledica klopke 1, 2, 3, 5 ili 6. To znači, da bi visok kvalitet mogao da bude prouzrokovan povezanim činiocem, mešajući uzrok sa posledicom, ili jednostavno čistom slučajnošću. Ili bi kôd visokog kvaliteta mogao da bude rezultat pogrešne klasifikacije; možda svi projektanti u kompaniji imaju fakultetske diplome, pa će posedovanje akademskog stepena biti u korelaciji sa svakim drugim činiocem. Najzad, studija bi mogla da bude polarizovana; programeri izabrani za studiju bi mogli biti odabrani baš zbog toga što imaju diplome (odnosno, programeri bez diploma bi bili ostavljeni izvan studije), pa ne možemo da izvučemo validan zaključak o odnosu akademskog stepena i visokog kvaliteta.

Dakle, okrećemo se studijama sa mnogo višim nivoom kontrole. Pretpostavite, kao što je napomenuto u polju F, da identifikujete projekte koji su već razvili visokokvalitetan kôd i onda nalazite projekte sa sličnim karakteristikama koji su proizveli kôd slabijeg kvaliteta. Zatim, poredite vrednosti relevantnih činilaca da biste videli da li jedan činilac ili više njih izdvajaju projekat sa visokokvalitetnim kodom od projekata nižeg kvaliteta. Ako projekti sa visokokvalitetnim kodom nemaju značajno različite karakteristike od onih bez takvog koda, onda su možda klopke 3, 4 ili 5 zamaglile vezu. Odnosno, postoji mala mogućnost da je slučajnost dovela do toga da nema razlike u kodu, da su svi subjekti imali isti nivo svakog činioca, ili da ste pogrešno klasifikovali neke činioce.

S druge strane, možete da otkrijete da je činilac X doveo do visokokvalitetnog koda. Ako ste sigurni da međudejstva, slučajnost ili polarizacija nisu uticali na studiju, onda ćete možda poželeti da sačekate još studija koje bi potvrdile vaše nalaze.

Pretpostavite da uradite mnogo više studija slučajeva i anketa, ili da ste ih pronašli u literaturi. Neke od njih ne prikazuju vezu između činioca X i kvaliteta (verovatno zbog klopki), ali većina to čini. Ako je situacija u kojoj proučavate te činioce retka (odnosno, domen je nov ili se pristup rešavanju problema retko koristi), onda morate da se oslonite na „totalan dokaz" da biste dokazali vaš slučaj. To znači da morate da pretpostavite da se u najvećem delu vremena možete pouzdati u to da će činilac X poboljšati kvalitet koda.

S druge strane, ako situacija nije retka, onda možete da uradite izgledne studije. Prikupljate informacije o širokom skupu projekata pre, za vreme i posle razvoja. Zatim, analizirate podatke da biste videli koji činioci dovode do koda visokog kvaliteta. U studiji ove vrste ima mnogo manje izgleda za polarizaciju, zato što prikupljate podatke o činiocu pre nego što saznate o rezultujućem kvalitetu. Ako ne pronađete nikakvu vezu između činioca X i kvaliteta, slučajnost, homogenost ili pogrešna klasifikacija (klopke 3, 4 i 5) mogle bi objasniti zašto. U ovoj tački, ako ste sigurni da klopke na vas nisu uticale, vreme je da izvršite reviziju svojih hipoteza.

Alternativno, pretpostavite da ste pronašli vezu između činioca X i kvaliteta. Veza i dalje može da bude rezultat međudejstva ili slučajnosti, a vi možete poželeti da sačekate još studija da biste potvrdili nalaz pomoću „totalnog dokaza". Ali, ako želite jači dokaz da X dovodi do kvaliteta, možete da izvršite formalni eksperiment. Koristeći dobar plan eksperimenata i pažljivo kontrolišući činioce, možete da proučavate da li je činilac X

povezan sa kvalitetom. Upotrebom tehnika kao što su ponavljanje, uvođenje slučajnosti i lokalna kontrola, možete da izbegnete većinu klopki iz tabele 12.2.

Kada se vaš eksperiment završi, možete otkriti da je kvalitet isti u svim obrađenim grupama; dakle, ili činilac nije povezan sa kvalitetom, ili su na vas uticali slučajnost, ili klopke 7, 8 ili 9. Verovatno treba da sprovedete još studija. Ako vaš eksperiment zaista pokazuje vezu između činioca X i kvaliteta, možete da izvestite projektni tim o svom otkriću.

Dakle, svaka od različitih vrsta empirijskih istraživanja igra ulogu u saznavanju kako razni činioci procesa, proizvoda i resursa utiču na kvalitet softvera. Vrsta i broj studija zavisi od vremena, troškova, praktičnosti i potrebe. Posebno, što je više kontrole, to je studija jača. Da biste razumeli razliku između zbirke studija slučajeva i zbirke eksperimenata, razmotrite razliku između porota za građanske i kriminalne sudske procese u SAD. Za građanske parnice, porota mora da odredi da li premoć u dokazima podržava tužioca ili optuženog. Dakle, ako je 51% dokaza u korist tužioca, onda porota mora da donese odluku u njegovu korist. Međutim, u kriminalnom sudskom postupku, porota mora da pronađe dokaz izvan razumne sumnje – odnosno, porota mora da bude razumno sigurna da je optuženi kriv. Slično tome, ako ste zadovoljni što vas ukupnost dokaza dovoljno ubeđuje da pređete na novu tehniku ili alat, onda zbirka studija slučajeva može da podrži vašu odluku. Međutim, ako pitanja troškova ili kvaliteta (kao što su ona u izgradnji bezbedno ili poslovno kritičnih sistema) zahtevaju da imate dokaz van svake senke sumnje, onda bi verovatno trebalo da potražite eksperimente koji bi potvrdili vašu odluku, ili čak da izvedete sopstvene eksperimente.

Ova igra na tabli i njoj pridružene klopke nude vam smernice za donošenje odluka o softverskoj tehnologiji. Ali softversko inženjerstvo ne predstavlja uvek kontrolisane situacije, pa morate da budete realni i praktični. Istraživači kontrolišu ono što mogu i pri tom sasvim dobro poznaju moguće efekte nekontrolisanih činilaca tokom istraživanja delotvornosti akcija. Možete da procenite njihove objavljene rezultate i primenite ih na sopstveno okruženje. Studije koje čitate ili sami izvodite mogu da otkriju takve odnose u razvoju softvera koji će vam pomoći da donosite odluke na osnovu informacija i da izgradite bolje proizvode.

12.3 PROCENA U ODNOSU NA PREDVIĐANJE

Procena uvek obuhvata merenja. Informacije skupljamo da bismo izdvajali različite vrednosti zavisno i nezavisno promenljivih i da bismo manipulisali informacijama i postigli bolje razumevanje. Pored toga, merenje nam pomaže da razdvojimo tipične situacije od neuobičajenih, ili da definišemo osnove i skup ciljeva.

Formalno, **mera** je preslikavanje skupa entiteta i atributa u stvarnom empirijskom svetu na predstavu ili model u matematičkom svetu. Na primer, razmatramo skup ljudi i njihovih atributa, kao što su visina, masa i boja kose; zatim možemo da definišemo preslikavanja koja hvataju svojstva ljudi i čuvaju odnose. Na primer, možemo da kažemo da je Hughie visok 2 metra, Dewey 2,4 metra, a Louie 2,5 metara. Onda možemo da ma-

nipulišemo brojevima ili simbolima u matematičkom svetu da bismo dobili više informacija i razumevanja o stvarnom svetu. U našem primeru, naučili smo da je prosečna visina te trojice ljudi 2,3, pa su dvojica iz našeg skupa iznad proseka po visini.

Slično tome, možemo da preslikamo boju kose na skup {smeđa, plava, crvena, crna}; simboli ne moraju da budu brojevi. Možemo da koristimo preslikavanje boje kose da bismo generisali raspodele, koje nam govore kako 67% našeg uzorka ima smeđu kosu, a 33% crnu. Formalni okvir i opis teorije merenja mogu da se pronađu u radu Fentona i Pfleegera (1977).

Veliki broj softverskih mera evidentira informacije o atributima proizvoda, procesa ili resursa. Fenton i Pfleeger opisuju izvođenje i primenu mnogih takvih mera. Pronalaženje najbolje mere za vaše potrebe nije uvek lako jer sve mere koje uzmete u obzir služe za merenje ili predviđanje istih atributa (odnosno, cenu, veličinu ili složenost) na veoma različite načine. Na primer, u poglavlju 5 smo videli da postoje desetine načina za merenje složenosti konstrukcije. Takvo merenje može da nas zbuni jer postoje različite mere za istu stvar, a ponekad neka mera može da ima suprotno značenje od druge! Izvor zabune često leži u nepostojanju validacije softverskih merenja. To znači da se može dogoditi da merenjima i ne dobijemo informaciju o atributu koju tražimo.

Da biste razumeli validaciju softverskog merenja, pogledajmo dve vrste sistema (Fenton i Pfleeger 1997):

1. *Sisteme za merenje* koji se koriste za procenu postojećeg entiteta, numerički karakterišući jedan ili više njegovih atributa;
2. *Sisteme za predviđanje* koji se koriste da predvide neke atribute budućeg entiteta i čiji je sastavni deo i matematički model sa odgovarajućim procedurama za predviđanje.

Neformalno, kažemo da je **mera validna** ako tačno karakteriše atribut koji meri. S druge strane, **sistem za predviđanje je validan** ako daje tačne prognoze. Na taj način, ne samo da sistemi za merenja nisu isto što i sistemi za predviđanje, nego je takođe i pojam validnosti sadržinski drgačiji za svakog od njih.

Validacija sistema za predviđanje

Sistem za predviđanje validiramo u datom okruženju ustanovljavajući njegovu tačnost empirijskim sredstvima; odnosno, poredimo performansu modela sa poznatim podacima u datom okruženju. Postavljamo hipotezu, a zatim analiziramo podatke da bismo videli da li oni podržavaju hipotezu ili je opovrgavaju. Uzmimo, na primer, da želimo da utvrdimo da li je COCOMO validan za datu vrstu razvojnog projekta. Na osnovu podataka koji predstavljaju tu vrstu projekta procenjujemo tačnost s kojom COCOMO predviđa rad i trajanja. Ta vrsta validacije je dobro prihvaćena u softverskom inženjerstvu. Na primer, u podsetniku 12.1 opisuje se procena pouzdanosti modela za predviđanje.

Kada se potvrđuje model, prihvatljiva tačnost zavisi od više činilaca, uključujući i to ko vrši procenu. Početničke procene mogu da ne budu jednako tačne kao one koje prave iskusni stručnjaci. Razmatramo takođe razliku između **determinističkih** sistema za predviđanje (dobijamo uvek isti izlaz za dati ulaz) i **stohastičkih** sistema za predviđanje (izlaz će se probabilistički menjati za dati ulaz), s obzirom na dati model.

PODSETNIK 12.1 POREĐENJE PREDVIĐANJA POUZDANOSTI SOFTVERA

U poglavlju 9 opisano je predviđanje pouzdanosti i predstavljeno više tehnika i modela koji pomažu projektantima da predvide verovatnu pouzdanost sistema na terenu. U člancima iz oblasti softverskog inženjerstva opisano je više tehnika i modela i opisano je koliko je primena modela bila uspešna na datom projektu ili u određenom domenu.

Međutim, ljudi se nisu baš često bavili širim pitanjem modela koji bi bio najbolji u datim okolnostima. Lanubile (1996) opisuje rad izveden u saradnji sa Visaggiom na Univerzitetu u Bariju (Italija), gde su replicirane studije iz prošlosti i primenjene različite tehnike na 27 programa na programskom jeziku Pascal, razvijenih po istoj specifikaciji informacionog sistema; u programi su sadržali 118 komponenti za proučavanje. Lanubile i Visaggio su definisali da je komponenta visoko rizična ako su njene greške otkrivene prilikom testiranja, a nisko rizična ako nije otkrivena nijedna greška. Zatim su, koristeći nezavisne promenljive kao što su broj ulaza, broj izlaza, informacioni tok, ciklomatski broj, redovi koda i gustina komentara, ispitali sedam tehnika za predviđanje, izražemo u njihovim negativnim greškama (pogrešno klasifikovanje visokorizične komponente kao komponente niskog rizika), pozitivnim greškama (pogrešno klasifikovanje niskorizične komponente kao komponente visokog rizika), potpunosti (procenat komponenti visokog rizika koje su stvarno ispravno klasifikovane) i odbačenoj inspekciji (procenat identifikovanih komponenti koje su pogrešno klasifikovane).

Od 118 komponenti, dve trećine su upotrebljene da bi se definisali i podesili modeli, a preostala trećina je iskorišćena za validaciju. To je uobičajen pristup; podaci koji se koriste da bi se izgradio model zovu se **podaci za podešavanje**, a preostali podaci koji se koriste za ispitivanje modela zovu se **podaci za ispitivanje**. Rezultati, prikazani u tabeli 12.3, pokazuju da nijedna od tehnika nije bila dobra u odvajanju komponenti sa greškama od onih bez grešaka. U stvari, Lanubile i Visaggio su poredili rezultate sa onima koji su dobijeni prosto bacanjem novčića za svaku komponentu („glava" je visok rizik, a „pismo" nizak rizik) i otkrili su da nijedan model nije bio značajno bolji u pronalaženju komponenti sa visokim rizikom bez velikog rasipanja rada. Oni napominju da „objavljivanje samo empirijskih studija sa pozitivnim nalazima može praktičarima da pruži nerealna očekivanja koja brzo sledi isto tako nerealno razočaranje". Dalje, oni kažu da „model za predviđanje, od najjednostavnijeg do najsloženijeg, vredi samo ako se koristi sa lokalnim procesom da se izabere metrika koja je valid na kao predskazivač"

U stohastičkom modelu, dozvoljavamo prozor greške sa varijabilnom širinom oko stvarne vrednosti. Sistemi za predviđanje cene softvera, potrebnog rada, vremenskih rokova i pouzdanosti imaju veliku marginu greške; kažemo da su oni veoma stohastični. Na primer, možete da otkrijete da je, pod određenim okolnostima, predviđanje pouzdanosti vaše organizacije tačno do 20%; to znači da će predviđeno vreme do sledećeg otkaza biti u okviru 20% stvarnog vremena. Taj okvir opisujemo kao **opseg prihvatljivosti**: iskaz o maksimalnoj razlici između predviđanja i stvarne vrednosti. Dakle, 20% je opseg prihvatljivosti vašeg modela. Zavisno od okolnosti, možda zaključite da je okvir suviše velik da bi bio koristan; na primer, okvir može biti suviše širok za efikasno planiranje održavanja. Drugim menadžerima velik opseg prihvatljivosti može sasvim da odgovara, imajući u vidu neizvesnosti razvoja softvera. Uvek morate unapred da odredite prihvatljiv opseg, pa tek onda da koristite sistem za predviđanje.

TABELA 12.3 Rezultati poređenja modela za predviđanje (Lanubile 1996) 1996 IEEE

Tehnika modelovanja	Predvidljiva validnost	Proporcija negativnih grešaka (%)	Proporcija pozitivnih grešaka (%)	Proporcija pogrešnih klasifikacija (%)	Potpunost (%)	Ukupna inspekcija (%)	Odbačena inspekcija (%)
Diskriminantna analiza	p = 0,627	28	26	54	42	46	56
Analiza glavne komponente plus diskriminantna analiza	p = 0,408	15	41	56	68	74	55
Logistička regresija	p = 0,491	28	28	56	42	49	58
Analiza glavne komponente plus diskriminantna analiza	p = 0,184	13	46	59	74	82	56
Model logističke klasifikacije	p = 0,643	26	21	46	47	44	47
Slojevita neuralna mreža	p = 0,421	28	28	56	42	49	58
Holografička mreža	p = 0,634	26	28	54	47	51	55
Glava-pismo?	p = 1,000	25	25	50	50	50	50

Modeli predstavljaju posebno težak problem kada projektujemo eksperiment ili studiju slučaja, zato što njihova predviđanja mogu da utiču na ishod. To znači da predviđanja postaju ciljevi, a projektanti se bore da ih postignu, namerno ili nenamerno. Taj efekat je uobičajen kada se koriste modeli troškova i rasporeda, a menadžeri projekta pretvore predviđanja u ciljeve radi njegovog završetka. Iz tog razloga, eksperimenti za procenjivanje modela se ponekad projektuju kao „dvostruki slepi" eksperimenti, gde učesnici ne znaju za predviđanja dok se eksperiment ne završi. S druge strane, neki modeli, kao što su modeli pouzdanosti, ne utiču na ishod, zato što pouzdanost, merena kao srednje vreme do otkaza, ne može da se proceni dok softver ne bude spreman za upotrebu na terenu. Prema tome, vremenom između uzastopnih otkaza ne može da se „upravlja" onako kako bi se upravljalo rokovima i budžetima nekog projekta.

Sistemi za predviđanje ne moraju da budu složeni da bi bili korisni. Na primer, Fuchs izveštava da izveštaji o vremenskoj prognozi u Austriji imaju tačnost od 67% kada su zasnovani na vremenskim uslovima od prethodnog dana. Upotreba usavršenih računarskih modela povećava tu tačnost samo za 3% (Fenton i Pfleeger 1997)!

Validacija mera

Validacija softverskih merenja je veoma različita od validacije sistema za predviđanje. Želimo da budemo sigurni da mera „hvata" ona svojstva atributa koja bi trebalo. Na primer, da li ciklomatski broj meri veličinu ili složenost? **Uslov predstavljanja** kaže da odnosi između numeričkih vrednosti merenja odgovaraju odnosima atributa koje zapažamo u stvarnom okruženju. Dakle, ako definišemo meru za visinu, onda moramo obezbediti da je mera visine veća za Džejmsa nego za Suzanu, ako je Džejms viši od Suzane. Da bismo validirali meru, neophodno je demonstrirati da uslov predstavljanja važi za meru i njen odgovarajući atribut.

Na primer, pretpostavite da istraživač definiše meru *m* za koju se tvrdi da meri dužinu programa. Da bi se validirao *m*, moramo da izgradimo formalni model koji opisuje programe i funkciju koja čuva intuitivno poimanje dužine u vezama koje opisuju programe. Možemo izvršiti proveru sa ciljem da utvrdimo da li se *m* ponaša na očekivane načine. Na primer, pretpostavite da spojimo dva programa *P1* i *P2* da bismo dobili program čija je dužina kombinovana od dužina *P1* i *P2*. Tada bi *m* trebalo da zadovolji jednačinu

$$m(P_1, P_2) = m(P_1) + m(P_2)$$

Slično tome, ako je program *P1* duži od *P2*, onda je

$$m(P_1) > m(P_2)$$

Mnoge mere za dužinu mogu se validirati na ovaj način, uključujući dužinu redova koda, broj operatora i operanada i broj znakova tačka-zapeta. Vežba validacije nas uverava da su mere ispravno definisane i da su usklađene sa ponašanjem entiteta u realnom okruženju. Kada validiramo meru, važno je posmatrati je u kontekstu u kome će biti upotrebljena; jedna mera može da bude validna za jednu namenu, ali ne i za drugu. Na primer, ciklomatski broj može da bude validan za merenje broja nezavisnih putanja, ali ne za merenje razumljivosti. (Pročitajte podsetnik 12.2.)

Strog zahtev za validaciju

Moguće je da mera služi za obe svrhe: kao mera atributa i kao ulaz u sistem za predviđanje. Na primer, redovi koda mogu da mere veličinu programa, a ponekad su korisne kao predskazatelj greške. Ali, mera može da bude jedno ili drugo, a ne oboje. Ne bi trebalo da odbacimo meru kao nevalidnu zato što nije deo sistema za predviđanje. Ako je mera validna za procenu, zovemo je **validnom u užem smislu**, ili **interno validnom**. Mere atributa su interno validne, a takođe su korisne i u predviđanju. Mera je **validna u širem smislu** ako je i interno validna i deo je sistema za predviđanje.

Pretpostavite da želimo da pokažemo da je neka posebna mera validna u širem smislu. Počinjemo postavljanjem hipoteze koja iskazuje specifičan odnos između mere i atributa. Potom bi trebalo da sprovedemo pažljivo kontrolisan eksperiment koji pokazuje da se odnos validira empirijskim podacima. Dokaz mora da bude više od statističke korelacije; moramo da pokažemo uzrok i posledicu. Na primer, možemo da tvrdimo

da je mera modularnosti dobar predskazatelj troškova. Da bismo pokazali da je mera validna, moramo da modelujemo odnos između modularnosti i troškova razvoja; model mora da iskaže sve činioce koji međusobno povezuju i utiču na modularnost i troškove. Nakon toga moramo da pokažemo da promena u modularnosti ima uvek jasan uticaj na troškove. Samo tada možemo da prosudimo da li je merenje modularnosti ekvivalentno merenju troškova razvoja.

PODSETNIK 12.2 REDOVI KODA I CIKLOMATSKI BROJ

Jednostavno je pokazati da broj redova koda predstavlja validnu meru veličine programa. Međutim, on nije validna mera složenosti niti je deo pouzdanog sistema za predviđanje složenosti. Fenton i Pfleeger (1997) objašnjavaju da nedostatak ne leži u broju redova koda kao meri, već u nedovoljno preciznoj definiciji pojma složenosti. Iako se složenost u opštem slučaju opisuje kao atribut koji utiče na pouzdanost, održljivost, trošak itd., neodređenost koja prati njegovu definiciju predstavlja problem u istraživanju složenosti.

Međutim, problemi vezani za složenost ne sprečavaju da se broj redova koda upotrebljava kao korisna mera drugih atributa, a ne veličine. Na primer, pretpostavimo da postoji slučajna asocijacija između velikog broja redova koda i velikog broja grešaka pri jediničnom testiranju. Ta veza može pomoći prilikom izbora strategija testiranja i smanjenja rizika.

S druge strane, više studija je pokazalo značajnu korelaciju između redova koda i ciklomatskog broja. Da li ova korelacija dokazuje da se ciklomatski broj povećava sa veličinom? Ako bi ciklomatski broj predstavljao meru veličine, tada bi veći kod uvek bio i kompleksniji. Jednostavno je formulisati potpuno suprotan primer ovoj hipotezi. Ako pažljivo proučimo podatke koji pokazuju vezu ciklomatskog broja i redova koda, ono što uočavamo jeste da se broj mesta odlučivanja u sklopu komponente obično povećava sa povećanjem dužine koda.

Softverski inženjeri ponekad zaboravljaju da statistička korelacija nije isto što i uzrok i posledica. To što je brak u jakoj korelaciji sa razvodom ne znači da je „u braku" validna mera razvoda! Slično tome, iako može postojati statistička korelacija između modularnosti i troškova razvoja, modularnost može da ne bude i jedini činilac koji određuje te troškove. Meriti ono što je na raspolaganju ili što je lako merljivo predstavlja izazov, ali kao ljudi od nauke moramo da gradimo modele i uočavamo složene odnose.

Courtney i Gustafson (1993) uvode statistički razlog zbog čega se moramo zabrinuti pri korišćenju korelacionih pristupa. Nestrukturisane korelacione studije mogu da identifikuju lažne asocijacije. Na primer, uz nivo značajnosti 0,05, možemo da očekujemo značajnu, ali lažnu slučajnu korelaciju jednom u 20 puta. Ako imamo 5 nezavisno promenljivih i tražimo 10 mogućih korelacija između parova, lažna korelacija se dobija sa verovatnoćom 0,5! U takvim situacijama, bez hipoteze o razlogu postojanja veze, nemamo nikakvo realno uverenje da odnos nije lažan.

12.4 PROCENJIVANJE PROIZVODA

Videli smo da je broj produkata u razvoju softvera velik: zahtevi, konstrukcije, komponente koda, slučajevi korišćenja i scenariji za testiranje, vodiči za korisničke, unakrsne reference i drugo. Svaki produkt možemo ispitati da bismo odredili da li on ima poželjne atribute. To znači da možemo da pitamo da li dokument, datoteka ili sistem imaju izvesne osobine, kao što su potpunost, doslednost, pouzdanost ili održljivost. Možemo da koristimo pojmove kvaliteta, kao što je McCallov model uveden u poglavlju 1, da bismo obezbedili okvir za naša pitanja.

Modeli kvaliteta proizvoda

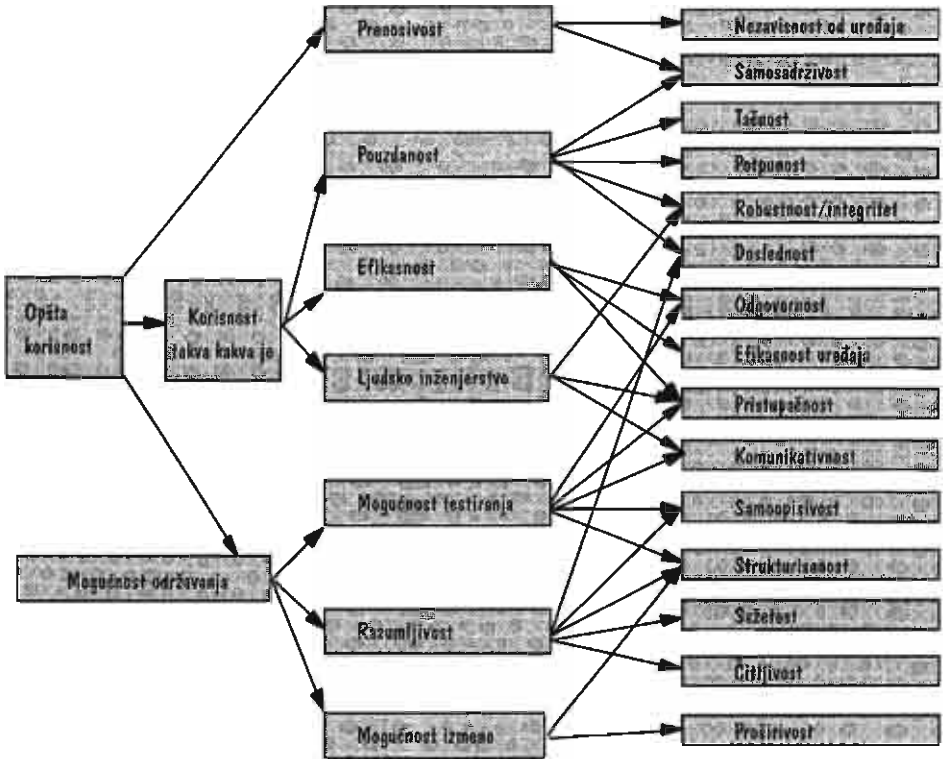
Postoji više modela kvaliteta koji sugerišu na koje se sve načine mogu međusobno povezati različiti atributi kvaliteta. Svaki model nam pomaže da razumemo kako više delova doprinosi celini. Često sužavamo fokus i razmišljamo samo o greškama i otkazima; ti modeli nam pokazuju da je pojam kvaliteta mnogo širi. Kada procenjujemo kvalitet proizvoda procesa razvoja, moramo da sagledamo tu širu sliku.

Boehmov model. Postoji više modela kvaliteta softvera, a Boehm i saradnici su konstruisali jedan od najpoznatijih, koji je prikazan na slici 12.2. On je sličan McCallovom u tome što predstavlja hijerarhiju karakteristika, od kojih svaka doprinosi ukupnom kvalitetu (Boehm i dr. 1978). Boehmov pojam uspešnog softvera uključuje potrebe i očekivanja korisnika isto kao i McCallov, ali Boehm takođe uključuje karakteristike performanse hardvera koje nedostaju u McCallovom modelu. Ispitajmo detaljnije Boehmov model.

Boehmov model počinje korisnošću softvera. Dakle, Boehm i saradnici dokazuju da, prvo i pre svega, softver mora da bude koristan. Ako nije, onda je njegov razvoj bio gubljenje vremena, novca i rada. Korisnost možemo da sagledavamo na više načina, u skladu sa vrstom korisnika koji koriste sistem nakon njegove isporuke.

Prvi tip korisnika je osnovni naručilac, koji je zadovoljan korisnošću ako sistem radi ono što on želi. Međutim, može biti i drugih koji žele da koriste sistem na drugom računaru ili lokaciji. U tom slučaju, sistem mora da bude **prenosiv** bi mogao da se premešta sa jednog računara na drugi, a da još uvek ispravno funkcioniše. Sistem bi takođe trebalo da bude prenosiv u nešto drugačijem smislu. Ponekad ukupna konfiguracija ostaje ista, ali je hardver ili softver dorađen u noviji model ili verziju. U tom slučaju, sistem bi trebalo da bude prenosiv i na nov ili drugačiji model ili verziju, bez uticaja na funkcionalnost. Na primer, ako se jedan prevodilac za programski jezik zameni drugim za isti jezik, funkcije sistema ne bi trebalo da se pogoršaju. Dakle, druga vrsta korisnika sistema su oni koji se bave tim nadograđenim ili dorađenim sistemom.

Najzad, treća vrsta korisnika je programer koji održava sistem, praveći sve promene koje mogu zatrebati s promenom zahteva kupca, ili kada se otkriju greške. Programeri moraju da budu u stanju da otkriju izvor greške, pronađu module koji izvode posebnu funkciju, razumeju kôd i modifikuju ga.



SLIKA 12.2 Boehmov model kvaliteta

Sve tri vrste korisnika se nadaju da je sistem pouzdan i efikasan. Kao što smo napomenuli u poglavlju 9, **pouzdanost** znači da sistem ispravno radi u vrlo dugim periodima vremena bez otkaza. Prema Boehmovom modelu, mnoge softverske karakteristike doprinose pouzdanosti, uključujući tačnost, robustnost i potpunost. Posebno, ako sistem proizvodi ispravan rezultat do korektnog stepena tačnosti, kažemo da sistem poseduje **integritet**, atribut koji je potreban za pouzdanost. Štaviše, ako se isti skup ulaznih podataka podnese sistemu više puta pod istim uslovima, rezultati bi trebalo da su podudarni; ta karakterstika se zove **doslednost** funkcije.

U isto vreme, sistem bi trebalo da proizvede svoje rezultate, ili da izvede svoju funkciju na vreme, kao što je određeno potrebama kupca. Dakle, podaci bi trebalo da budu pristupačni kada je to potrebno, a sistem bi trebalo da ima razuman odziv na zahtev korisnika.

Najzad, korisnici i programeri moraju da otkriju da je sistem jednostavan za učenje i upotrebu. Taj aspekt ljudskog inženjerstva može ponekad da bude i najkritičniji. Sistem može da bude vrlo dobar u izvođenju funkcije, ali ako korisnici ne znaju kako da ga upotrebe, on predstavlja neuspešan sistem.

Otuda, Boehmov model dokazuje da je kvalitetan softver onaj koji zadovoljava potrebe korisnika i programera koji sa njim rade. On odražava shvatanje kvaliteta koji znači da softver:

- radi ono što korisnik od njega traži;
- ispravno i efikasno koristi računarske resurse;

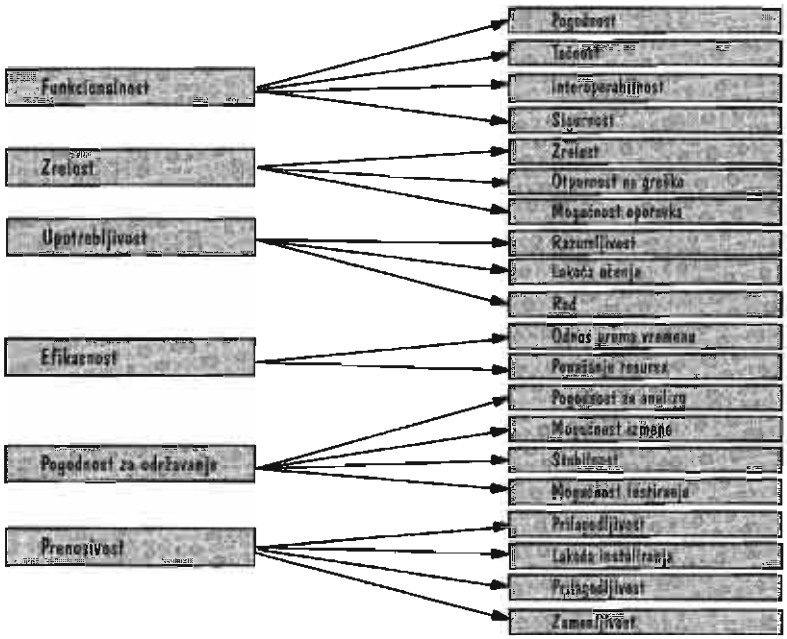
- jednostavan je za učenje i korišćenje;
- dobro je projektovan, dobro kodiran i lako se testira i održava.

ISO 9126. Početkom 1990-ih godina, u softverskom inženjerstvu se pokušalo sa konsolidacijom mnogih aspekata kvaliteta u jedan model koji bi delovao kao svetski standard za merenje kvaliteta softvera. Rezultat je bio ISO9126, hijerarhijski model sa šest glavnih atributa koji doprinose kvalitetu (International Standardization Organization 1991). Na slici 12.3 ilustrovana je hijerarhija, a u tabeli 12.4 definisani su glavni atributi.

U standardu se preporučuje direktno merenje karakteristika na desnoj strani, ali se ne daju detalji o tome kako da se merenja vrše.

Jedna od glavnih razlika između ISO i modela koje su dali McCall i Boehm je ta što je ISO hijerarhija striktna: svaka od karakteristika na desnoj strani odnosi se tačno na jedan atribut na levoj strani. Štaviše, karakteristike na desnoj strani odnose se na korisnikov pogled na softver, a ne na interni pogled njegovog projektanta.

Ti modeli, kao i mnogi drugi, pomažu da se razjasni šta je to što vrednujemo u softveru koji pravimo i koristimo. Ali, nijedan od modela ne uključuje razlog zašto su neke od karakteristika obuhvaćene, a druge nisu i zašto se pojedini atributi pojavljuju u hijerarhiji. Na primer, zašto nijedan od modela ne uključuje bezbednost? I zašto je prenosivost karakteristika na najvišem nivou u modelu ISO 9126, a ne potkarakteristika? Pored toga, nema smernica o tome kako da se komponuju karakteristike nižeg nivoa u one višeg nivoa, da bi se dobila ukupna procena kvaliteta. Ti problemi nam otežavaju da odredimo da li je dati model potpun ili dosledan.



SLIKA 12.3 ISO 9126 model kvaliteta

TABELA 12.4 ISO 9126 karakteristike kvaliteta

Karakteristika kvaliteta	Opis
Funkcionalnost	Skup atributa koji se oslanjaju na skup funkcija i njihovih propisanih osobina. Funkcije su te koje zadovoljavaju postavljene ili sadržane potrebe.
Pouzdanost	Skup atributa koji se oslanjaju na sposobnost softvera da održi svoj nivo performanse pod datim uslovima u datom periodu vremena.
Upotrebljivost	Skup atributa koji se oslanjaju na napor potreban za upotrebu i individualnu procenu te upotrebe od datog ili pretpostavljenog skupa korisnika.
Efikasnost	Skup atributa koji se oslanjaju na odnos između performanse softvera i količine resursa korišćenih pod datim uslovima.
Mogućnost održavanja	Skup atributa koji se oslanjaju na napor potreban da se izvrše propisane izmene (koje mogu da obuhvate ispravke, poboljšanja, ili prilagođenja softvera promenama u okruženju i promenama zahteva i funkcionalnih specifikacija).
Prenosivost	Skup atributa koji se oslanjaju na mogućnost softvera da se prenosi iz jednog okruženja u drugo (uključujući organizaciono, hardversko ili softversko okruženje).

Dromeyev model. Dromey se bavio ovim pitanjima predlažući način da se konstruiše model kvaliteta zasnovan na proizvodu, čije se sve karakteristike najnižeg nivoa mogu meriti (Dromey 1996). Njegova tehnika se zasniva na sledeće dve postavke:

1. Izgleda da više osobina proizvoda utiču na kvalitet softvera.
2. Pored nešto anegdotskih, empirijskih dokaza, ima malo formalne osnove za razumevanje koji atributi niskog nivoa utiču na one višeg nivoa.

Dromey predlaže generičku tehniku za izgradnju modela kvaliteta. On napominje da se kvalitet proizvoda u velikoj meri određuje izborom komponenti koje sačinjavaju proizvod (uključujući dokumente zahteva, korisničke vodiče, dizajn, kao i kôd), opipljivim svojstvima komponenti i njihovom kompozicijom. Štaviše, on klasifikuje kvalitete koristeći četiri osobine:

1. svojstva ispravnosti;
2. unutrašnja svojstva;
3. kontekstna svojstva,
4. opisna svojstva.

Zatim, on predlaže da atributi kvaliteta visokog nivoa budu samo oni čiji je prioritet visok (i ti prioriteti su različiti u različitim projektima). U jednom primeru, on razmatra kombinaciju osam atributa visokog nivoa: šest atributa iz ISO 9126, plus atributi ponovnog korišćenja i zrelosti procesa. Atributi ponovnog korišćenja sastoje se od:

- nezavisnosti od mašine,
- mogućnosti razdvajanja,
- mogućnosti konfigurisanja,

a atributi zrelosti procesa od:

- orijentacije klijenta,
- dobre definisanosti,

- ubedljivosti i
- efektivnosti.

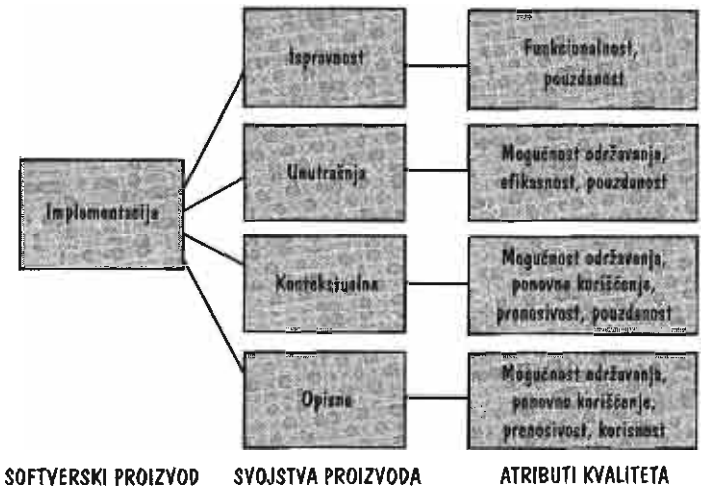
Da bi te karakteristike učinio opipljivijim, Dromey kombinuje atribute sa svojim okvirom da bi dobio veze koje su prikazane na slici 12.4. Zatim, on procenjuje svaku vrstu softverske komponente u svetlu tog okvira: dva primera su pokazana na slici 12.5. Model se zasniva na sledećih pet koraka:

1. Identifikovanje skupa atributa kvaliteta visokog nivoa.
2. Identifikovanje komponenti proizvoda.
3. Identifikovanje i klasifikovanje najznačajnijih, opipljivih svojstava koja donose kvalitet za svaku komponentu.
4. Predlaganje skupa aksioma za povezivanje svojstava proizvoda sa atributima kvaliteta.
5. Procenjivanje modela, identifikacija njegovih slabosti i njegova rafinacija ili ponovno kreiranje.

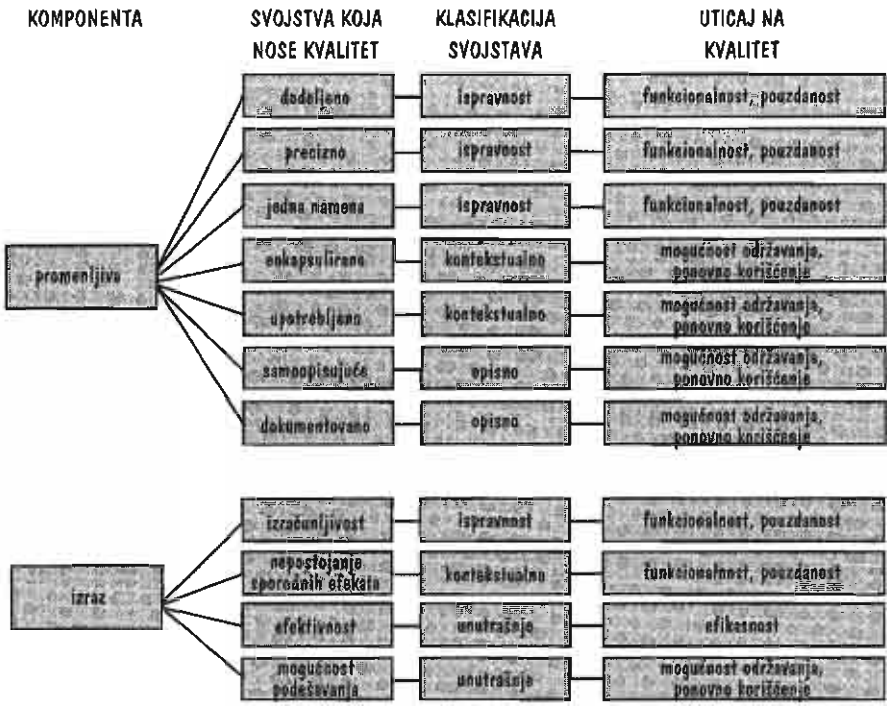
Jasno, ovi koraci mogu da se upotrebe da bi se proizveli modeli za zahteve, dizajn, kôd i druge produkte razvojnog procesa, a svaki model treba da odražava ciljeve proizvoda i projekta.

Ustanovljavanje osnova i ciljeva

Drugi način procene ili ocenjivanja proizvoda je da se on poredi sa osnovom. **Osnova** opisuje, na neki merljiv način, uobičajen ili tipičan rezultat u organizaciji ili kategoriji. Na primer, možemo da kažemo da je srednji ili tipičan projekat u kompaniji onaj od 30 000 redova koda; takođe možemo da napomenemo da se u tipičnom projektu organizacije otkriva jedna greška na hiljadu redova koda u inspekcijama i tri greške na hiljadu redova koda prilikom testiranja. Dakle, 30 000 redova koda je osnovna veličina za tu kompaniju, a tri greške na hiljadu redova koda je osnovna učestalost otkrivanja greške pri testiranju. Drugi projekti se porede sa baznim; kôd je manji ili veći, a otkrivanje greške je bolje ili lošije.



SLIKA 12.4 Povezivanje svojstava proizvoda sa atributima kvaliteta



SLIKA 12.5 Svojstva proizvoda i njihov efekat na kvalitet

Osnove su korisne za upravljanje očekivanjima. Kada je neka određena vrednost bliska baznoj, ona ne ukazuje ni na kakvo izobličeno ponašanje. S druge strane, kada se neka vrednost dramatično razlikuje od bazne, obično poželimo da istražimo zašto. Ponekad postoje dobri razlozi za takvu razliku, ali u drugim slučajevima nas razlika upozorava da nešto treba da se ispravi ili promeni.

Cilj predstavlja varijaciju osnove. Menadžeri obično postavljaju ciljeve kao način da definišu minimalno prihvatljivo ponašanje. Na primer, kompanija može da izjavi da neće isporučivati proizvod dok projektanti ne pokažu da su otklonili najmanje 95% njegovih verovatnih grešaka. Ili, organizacija može da postavi cilj da razvije sisteme čije su komponente 50% višekratno upotrebljive. Ciljevi su opravdani kada odražavaju razumevanje osnove, tako da cilj nije do te mere daleko od postojeće situacije da ne bi mogao da se postigne.

Ministarstvo odbrane SAD koristi ciljeve za procenu softvera. Ministarstvo odbrane je analiziralo performanse Vlade i privrede SAD, grupišući projekte u slabe, srednje i najbolje u klasi. Iz njihovih nalaza, analitičari su preporučili ciljeve za projekte koje je ugovorilo Ministarstvo odbrane, zajedno sa ukazanim nivoima performanse, koji čine ono što su nazvali „menadžerskom zloupotrebom", prikazanom u tabeli 12.5. To znači da ako projekat pokazuje da je neki atribut blizu nivoa zloupotrebe, onda ministarstvo sumnja da nešto ozbiljno nije u redu sa tim projektom.

TABELA 12.5 Kvantitativni ciljevi za upravljanje projektima Ministarstva odbrane SAD (*Neofocus* 1995)

Stavka	Cilj	Nivo zloupotrebe
Efikasnost uklanjanja grešaka	>95%	<70%
Prvobitna gustina grešaka	<4 po tački funkcije	>7 po tački funkcije
Odstupanja ili premašenje troškova rezerve zbog rizika	0%	≥10%
Ukupno odstupanje od zahteva (funkcionalne tačke ili ekvivalentno)	<1% mesečno u proseku	≥50%
Ukupna programska dokumentacija	<3 stranice po funkcionalnoj tački	>6 stranice po funkcionalnoj tački
Promena osoblja	1 do 3% godišnje	>5% godišnje

Višekratna upotrebljivost softvera

Višekratna upotrebljivost zahteva da procenimo proizvode iz postojećih i budućih razvojnih projekata, da bismo odredili da li su oni takve vrste i kvaliteta kakvu bismo želeli za sledeći sistem koji budemo gradili. Dakle, razmotrićemo detaljnije neka pitanja u vezi sa višekratnom upotrebljivošću koja bi mogla da utiču na naše procene.

Mnogi softverski sistemi koje gradimo slični su jedan drugom; na primer, svaka kompanija verovatno ima računovodstveni sistem, personalni sistem i možda sistem za naplatu. Postoje zajednički delovi među sistemima sa sličnom namenom. Na primer, svaki kadrovski sistem uključuje podatke i funkcije koje se odnose na radnike u kompaniji. Umesto da stalno pravimo nove sisteme, možemo da pregledamo komponente iz starijih sistema i odredimo da li one mogu da se prilagode ili čak cele ponovo iskoriste za novi sistem. Čak i kada su aplikacije vrlo različite, moramo da ispitamo da li je odmah potrebno pisati drugu rutinu za sortiranje, komponentu za pretraživanje ili program za ulaz ekrana. Softverski istraživači veruju da višekratna upotrebljivost ima veliki potencijal za povećavanje produktivnosti i smanjivanje troškova, a neki radovi na primeni višekratne upotrebljivosti potvrđuju to njihovo uverenje.

Vrste višekratne upotrebljivosti. Pod **višekratnom upotrebljivošću softvera** podrazumevamo ponovnu upotrebu bilo kog dela softverskog sistema: dokumentacije, koda, konstrukcije, zahteva, slučajeva za testiranje i drugog. Basili (1990) smatra da održavanje treba posmatrati u smislu višekratne upotrebljivosti; uzimamo postojeći sistem i ponovo koristimo njegove delove da bismo izgradili sledeću verziju. Slično tome, on smatra da procesi i iskustvo mogu ponovo da se upotrebe, kao što to može i svaki opipljivi ili neopipljivi produkt procesa razvoja.

Postoje dve vrste višekratnog korišćenja, kako se to određuje iz perspektive korisnika. **Proizvođač višekratno koristi** komponente, ali ih i **potrošač ponovo koristi** u sledećim sistemima (Bollinger i Pfleeger 1990). Kao potrošači, možemo da koristimo ili ceo proizvod, bez modifikacije (to se zove **višekratna upotrebljivost tipa crne kutije**), ili da ga modifikujemo kako bi odgovarao našim posebnim potrebama (**višekratna upotrebljivost tipa providne kutije**). Višekratnu upotrebljivost tipa crne kutije izvodimo često, kada koristimo rutine iz matematičkih biblioteka, ili pravimo grafičke interfejse iz unapred definisanih skupova komponenti. Važno pitanje u višekratnoj upotrebljivosti tipa crne kutije je naša sposobnost da utvrdimo da višekratno upotrebljiv modul obavlja funkciju koja nam je potrebna na prihvatljiv način. Ako je komponenta kôd, želimo da budemo sigurni da je bez greške; ako je to skup testova, želimo da smo sigurni da potpuno izvršava funkciju za koju je namenjen.

Višekratna upotrebljivost tipa providne kutije (to se ponekad zove **višekratna upotrebljivost tipa bele kutije**) je češća, ali još uvek kontroverzna jer rad uložen da bi se komponenta razumela i modifikovala mora da bude manji od onog koji je potreban da bi se napisala nova ekvivalentna komponenta. Da bi se taj problem rešio, mnoge višekratno upotrebljive komponente su izgrađene sa više parametara da bi se jednostavnije prekrajale spolja gledano i da projektanti ne bi bili prinuđeni da se upuštaju u razumevanje komponente iznutra.

Nekoliko dostignuća u softverskom inženjerstvu čini višekratnu upotrebljivost izvodljivom. Jedno od njih je interes i usvajanje objektno orijentisanog razvoja (*object-oriented development*, OOD). S obzirom na to da OOD ohrabruje softverske inženjere da grade konstrukciju oko nepromenljivih komponenti, mnogi delovi konstrukcije i koda mogu da se primene na više sistema. Druga prednost koja ohrabruje višekratnu upotrebljivost je povećana složenost alata koji protežu svoje efekte kroz celokupan životni ciklus razvoja. Alati koji uključuju spremište za komponente razvijene za potrebe jednog sistema, mogu biti podsticajni za višekratnu upotrebljivost tih komponenti u kasnijim sistemima. Najzad, konstrukcija jezika može da ohrabruje ponovno korišćenje komponenti. Na primer, Ada obezbeđuje mehanizme za restrukturisanje softvera koji podržavaju pakovanje, a njene tehnike parametrizacije korisne su jer omogućuju generalizaciju softvera.

Pristupi višekratnoj upotrebljivosti su ili kompozicioni ili generativni. **Kompoziciona višekratna upotrebljivost** posmatra višekratno upotrebljive komponenti kao skup gradivnih blokova, a razvoj se obavlja odozdo nagore, gradeći ostatak sistema oko raspoloživih ponovo upotrebljivih komponenti. Za takvu vrstu višekratne upotrebljivosti, komponente se skladište u biblioteci (koja se često zove **spremište višekratno upotrebljivih komponenti**). Komponente moraju da se klasifikuju ili uvedu u kataloge, a sistem za izvlačenje mora da se koristi da bi se pregledale i birale komponente za upotrebu u novom sistemu. Na primer, naše spremište bi moglo da sadrži rutinu za pretvaranje vremena iz 12-časovnog u 24-časovni format (odnosno, da 9:54 P.M. postane 21.54). Slično tome, generator sistema za upravljanje bazom podataka Genesis obuhvata gradivne blokove koji nam dozvoljavaju da konstruišemo DBMS (sistem za upravljanje bazom podataka) skrojen prema našim potrebama (Prieto-Diaz 1993).

S druge strane, **generativna višekratna upotrebljivost** je specifična za određeni domen aplikacije. To znači da su komponente projektovane posebno da reše potrebe određene aplikacije i da se kasnije ponovo koriste u sličnim aplikacijama. Na primer, Nasa je razvila dosta softvera za praćenje satelita u orbiti. Komponente za analizu efemerida mogu da se projektuju tako da budu ponovo upotrebljive u više Nasinih sistema. Međutim, ti moduli nisu dobri kandidati za opšte spremište višekratno upotrebljivih komponenti, zato što oni verovatno neće biti potrebni u drugim domenima. Generativna višekratna upotrebljivost obećava isplativost visokog potencijala, a praksa se usredsredila na generatore aplikacija u specifičnim domenima. Neki od najpoznatijih takvih generaotra su Lex i Yacc, koji su projektovani da generišu leksičke analizatore i parsere.

Aktivnost koja čini zajedničku osnovu za obe vrste višekratne upotrebljivosti je **analiza domena**, proces je analiziranje domena aplikacije sa ciljem da se identifikuju zajedničke oblasti i načini za njihovo opisivanje (Prieto-Diaz 1987). Kompoziciona višekratna upotrebljivost se oslanja na analizu domena da bi se identifikovale zajedničke funkcije nižeg nivoa u širokoj klasi sistema. Generativni pristup zahteva aktivniju analizu; analitičar domena pokušava da definiše generički domen arhitekture i da specifikuje standarde interfejsa između delova arhitekture.

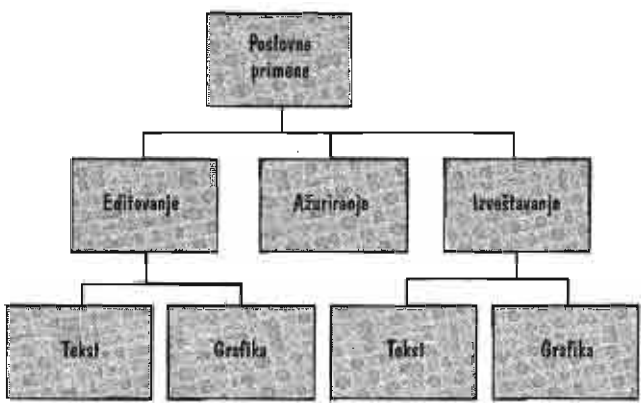
TABELA 12.6 Aspekti višekratne upotrebljivosti (prilagođeno iz Prieto-Diaza 1993)

Sadržaj	Opseg	Način	Tehnika	Namera	Proizvod
Ideje i koncepti Artefakti i komponente Procedure, znanja i iskustvo Obrasci Arhitektura	Vertikalni Horizontalni	Planiran i sistematski Ad hok, oportunistički	Kompoziciona Generativna	Crna kutija, takva kakva je Providna kutija, modifikovana	Izvorni kôd Konstrukcija Zahtevi Objekti Podaci Procesi Dokumentacija Testovi

Te ideje su dovele do pojmova horizontalne i vertikalne višekratne upotrebljivosti. **Vertikalna višekratna upotrebljivost** se bavi istom oblašću aplikacije ili domenom, ali **horizontalna višekratna upotrebljivost** ukršta domene. Višekratna upotrebljivost Nasinih rutina za efemeride je vertikalna, ali ponovna upotreba programa za sortiranje iz sistema baze podataka u grafičkom sistemu je horizontalna.

Tehnologija višekratne upotrebljivosti i izvlačenje komponenti. Jedna od najvećih prepreka za višekratnu upotrebljivost je potreba da se pretražuje u velikom skupu softverskih proizvoda kako bi se pronašao onaj koji je najbolji za neku posebnu potrebu. Taj posao je sličan prevrtanju gomile knjiga, umesto da se ode u biblioteku. Rešenje je u **klasifikaciji komponenti**, gde se zbirke višekratno upotrebljivih komponenti organizuju i uvode u kataloge prema određenoj klasifikacionoj šemi.

Komponente je moguće klasifikovati u hijerarhijskoj šemi, vrlo slično kao što su knjige organizovane u biblioteci. Kategorija na najvišem nivou se deli u potkategorije, koje se opet dele na potkategorije i tako dalje, kao što je prikazano na slici 12.6. Međutim, taj pristup nije fleksibilan i novi predmeti mogu lako da se dodaju samo na najnižem nivou.



SLIKA 12.6 Primer hijerarhijske klasifikacione šeme

Štaviše, potrebno je obimno unakrsno referenciranje, zato što, na primer, moramo da budemo u stanju da nađemo rutinu koja radi sa tekstom pod izveštavanjem, kao i pod editovanjem.

Rešenje tog problema je Prieto-Diazova **facetna šema klasifikacije** (Prieto-Diaz i Freeman 1987). Umesto da se koristi hijerarhija, svaka komponenta se opisuje pomoću uređene liste karakteristika koje se zovu facete. **Faceta** je vrsta deskriptora koji pomaže da se identifikuje komponenta, a zbirka faceta nam dozvoljava da identifikujemo više karakteristika odjednom. Na primer, facete ponovno upotrebljivog koda mogu da budu:

- aplikacijski domen;
- funkcija;
- jezik za programiranje;
- operativni sistem.

Svaka komponenta koda se identifikuje stavkom za svaki od pet faceta. Dakle, određena rutina bi mogla da bude označena kao

```
<switching system, sort, telephone number, Ada, UNIX>
```

Sistem za klasifikaciju se podržava pomoću **sistema za izvlačenje** ili **spremišta**, automatizovane biblioteke koja može da pretražuje i izvlači komponente u skladu sa korisnikovim opisom. Spremište često ima rečnik sinonima koji nam pomaže da razumemo terminologiju upotrebljenu u klasifikaciji. Na primer, rečnik može da nam kaže da pokušamo upotrebu reči „search" kao sinonima za „look up". Pored toga, spremište bi trebalo da rešava problem **konceptualne bliskosti** za izdvajanje sličnih faceta, koji nisu baš identični željenoj komponenti. Na primer, ako ne može da se nađe komponenta „modify", možemo da se usmerimo na „add" i „delete", koje su slične.

Sistem za izvlačenje takođe može da zapisuje informacije o zahtevima korisnika. Veliki broj neispunjenih zahteva određene vrste može da pokrene bibliotekara da predloži pisanje određene vrste komponente. Na primer, ako više korisnika traži komponentu koja menja datum iz američkog formata (11/05/98, ili November 5, 1998) u evropski format (05/11/98, ili 5. novembar 1998), a takva komponenta ne postoji, bibliotekar može da obezbedi da se ona napravi. Slično tome, ako više projektanata traži komponente za crtanje geometrijskih oblika, ali nijedna nije na raspolaganju, bibliotekar može da identifikuje dobre kandidate i da ih doda u biblioteku.

Najzad, sistem za izvlačenje može da sačuva opisnu informaciju o komponenti koja nam pomaže da odlučimo koju komponentu da izaberemo. Informacija o izvoru komponente (odnosno, o tome gde je kupljena, ili u okviru kog projekta je razvijena), njenoj pouzdanosti (broju otkrivenih grešaka, ili broju sati u kome je radila bez otkaza), ili u kom kontekstu je prethodno korišćena, može da nas ubedi da koristimo neku komponentu, a ne neku drugu. U podsetniku 12.3 opisano je kako se meri pogodnost komponente za ponovno korišćenje.

Iskustva sa višekratnom upotrebljivošću. Više kompanija i organizacija imalo je uspeha u ponovnom korišćenju značajne količine svog softvera. U svakom slučaju, zajednička karakteristika bila je predana i bezuslovna podrška rukovodstva (Braun i Prieto-Diaz 1990). Takva podrška je neophodna, zato što rukovodstvo mora da podnese dodatne troškove koje proizvođač ima zbog višekratne upotrebljivosti, dok te troškove ne potre ušteda koju će imati na strani potrošača.

PODSETNIK 12.3 MERENJE PONOVNOG KORIŠĆENJA

Kako da merimo komponentu koja je kandidat i kažemo da li bi ona trebalo da bude smeštena u našem spremištu za ponovno korišćenje? Mnogi menadžeri bi voleli jednostavnu metriku koja, kada se primeni na komponentu, razdvaja ponovo upotrebljive komponente od onih koje to nisu. Međutim, pronalaženje prave metrike nije tako lako kao što zvuči. Mere moraju da se bave ciljem, obično povezanim sa kvalitetom, produktivnošću ili vremenom za nabavku. A one takođe moraju da održavaju tačku gledišta osobe koja je postavila pitanje; projektanti imaju drugaciju perspektivu od rukovodstva. Pfleeger (1966) opisuje kako je prosto pitanje šta da se meri dovelo do generisanja preko 100 mogućih merenja – što je, jasno, bio neprihvatljiv broj.

Ali, čak i kada bismo imali dobar spisak merenja, kako da znamo koja ograničenja da stavimo na svako od njih? Da li je mala komponenta podesnija za višekratnu upotrebu od velike? Da li je bolje ponovo upotrebiti složen ili jednostavan kod? Ta istraživačka pitanja bila su rešavana na mnogo različitih načina. Neke organizacije gledaju u prošlost da bi odredile karakteristike komponenti koje su najčešće višekratno korišćene. Druge su birale merenja zasnovana na „inženjerskom prosuđivanju". Pristup koji najviše obećava koristili su u Tehnološkom centru Comel, gde su organizovalo automatizovano spremište koje je koristilo fasetnu klasifikaciju. Spremište je pratilo informacije o upitima, kao što su deskriptori koji su najčešće korišćeni. Ono je takođe zapisivalo kada je komponenta izabrana, kada je bila ispitivana ali ne i izabrana, i kada je odgovarala upitu, ali nije ispitivana. Onda je administrator spremišta razgovarao s korisnicima da bi shvatio zašto određene komponente nisu bile upotrebljene, a zašto druge jesu i to mnogo puta. Mere, kombinovane s ličnom interakcijom, dozvolile su administratoru da doda nove sinonime fasetnoj klasifikaciji, da promeni komponente da bi ih učinio ponovo upotrebljivim, da izbriše komponente koje projektantima nisu bile od koristi (i tako smanji vreme pretraživanja) i da pronađe nove komponente koje su bile poželjne, ali nisu bile ugrađene u spremište.

Štaviše, višekratna upotrebljivost često izgleda radikalno i preteće projektantima koji su navikli da pišu softver od samog početka. Jasno je da mnoga društvena, kulturološka i pravna pitanja, kao i ona tehnička stoje u vezi sa višekratnom upotrebom softvera.

Jedan od prvih izveštaja o iskustvima sa ponovnom upotrebom je onaj iz kompanije Rayteon (Lanergan i Grasso 1984). Organizacija za informacione procesne sisteme kompanijinog sektora za raketne sisteme zapazila je da je 60% konstrukcija i kodova njenih poslovnih aplikacija bilo redundantno i da su oni zato dobri kandidati za standardizaciju i ponovno korišćenje. Osnovan je program ponovnog korišćenja i ispitano je preko 5 000 COBOL izvornih programa, koji su klasifikovani u tri velike klase modula: za editovanje, ažuriranje i izveštavanje. U Rayteonu su takođe otkrili da je većina njihovih poslovnih aplikacija pala u jednu od tri logičke strukture. Te strukture su standardizovane i napravljena je biblioteka ponovo upotrebljivih komponenti. Nove aplikacije su varijacije standardnih softverskih struktura i grade se sklapanjem komponenti iz biblioteke. Softverski inženjeri su obučeni za korišćenje biblioteke i testirano je njihovo prepoznavanje kada neka struktura može ponovo da se upotrebi da bi se izgradila nova aplikacija. Ponovno korišćenje je sada obavezno. Posle šest godina ponovnog korišćenja koda na taj način, Rayteon je izvestio da je novi sistem sadržao u proseku 60% višekratno upotrebljenog koda, što je povećalo produktivnost za 50%.

Jedan od najuspešnijih programa višekratne upotrebljivosti o kome postoji izveštaj u literaturi, bio je onaj u GTE-u. GTE Data Services je osnovala Asset Management Program da bi razvila kulturu višekratne upotrebljivostiu korporaciji. Program je bio sličan onima koje smo do sada opisivali. Počelo se od analize postojećih sistema i identifikovanja ponovo upotrebljivih objekata. Svaki softverski proizvod koji bi mogao u potpunosti ili delimično da se ponovo upotrebi, postao je kandidat za takav objekat. Zbirka objekata je zatim klasifikovana i uvedena u katalog u automatizovanom bibliotečkom sistemu. U cilju održavanja biblioteke, promovisanja višekratne upotrebljivosti i podrške onima koji rade sa višekratno upotrebljivim kodom, formirano je nekoliko grupa:

- rukovodeća grupa za podršku, da bi obezbedila inicijativu, fondove i politiku višekratne upotrebljivosti;
- grupa za identifikaciju i kvalifikaciju, da bi identifikovala moguća oblasti višekratne upotrebljivosti i identifikovala, kupovala i sertifikovala nove priloge zbirci;
- grupa za održavanje, da bi održavala i ažurirala nove višekratno upotrebljive komponente;
- razvojna grupa, da bi pravila nove višekratno upotrebljive komponente;
- grupa za podršku korisnivima, da bi im pomogla i obučila ih da ispituju i procenjuju komponente koje se mogu ponovo upotrebiti.

Ne želeći da višekratna upotrebljivost postane obavezna, GTE je uvela podsticaje i nagrade. Programerima je plaćeno do 100 USD po svakoj komponenti koja je prihvaćena u biblioteci, a autori programa su dobijali stimulaciju kad god bi neka njihova komponenta bila ponovo upotrebljena u novom projektu. Ustanovljena je nagrada „Ponovni korisnik meseca", a menadžerima su davani bonusi ili povećanja budžeta za ostvarenje ciljeva višekratne upotrebljivosti. U prvoj godini programa, GTE je izvestila o 14% višekratnostni na svojim projektima, što je donelo uštede u vrednosti od 1,5 miliona USD (Swanson i Curry 1987).

U kompaniji Nipon Novel, koja je zapošljavala oko 100 softverskih inženjera, započeo je program višekratne upotrebljivosti u kome su takođe korišćeni novčani podsticaji. Plaćano je 5 centi (SAD) po redu koda projektantu koji bi ponovo upotrebio neku komponentu, a tvorac te komponente bi dobio 1 cent po redu koda koji bi bio ponovo upotrebljen. U 1993. godini, program je kompaniju koštao 10 000 USD, što je daleko manje od cene pisanja ekvivalentnog koda (Frakes i Isoda 1994).

Širom sveta pokrenuti su i drugi programi višekratne upotrebljivosti; većina njih usredsredila se na ponovno korišćenje koda. U podsetniku 12.4 opisano je nekoliko primera višekratne upotrebljivosti u Japanu. Vladine organizacije, kao što su European Space Agency i U.S. Department of Defense imaju velike institucionalizovane programe za ponovno korišćenje sa spremištima komponenti. Komercijalne kompanije, kao što su Hewlett-Packard, Motorola i IBM takođe su dosta uložile u višekratnu upotrebljivost. I Evropska zajednica je bila sponzor više poduhvata u vezi sa vušekratnom upotrebljivošću, uključujući Reboot i Surprise. Međutim, Griss i Wasser (1995) nas podsećaju da efikasno ponovno korišćenje ne zahteva ni široku implementaciju ni usavršenu automatizaciju.

Koristi, troškovi i uspesi višekratne upotrebljivosti. Višekratna upotrebljivost obećava povećanje produktivnosti i kvaliteta u celom procesu razvoja softvera. Produktivnost može da se poveća ne samo smanjivanjem vremena kodovanja, nego i smanjivanjem

vremena testiranja i izrade dokumentacije. Neka od najvećih smanjenja troškova nude se ponovnim korišćenjem zahteva i konstrukcije, zato što ona ima efekat talasa: ponovno korišćenje konstrukcije automatski obuhvata ponovno korišćenje koda, na primer. Štaviše, ponovno korišćenje komponenti može da poveća performansu i pouzdanost, zato što komponente mogu da se optimizuju i provere pre nego što se stave u spremište.

PODSETNIK 12.4 PONOVNO KORIŠĆENJE SOFTVERA KOD JAPANSKIH PROIZVOĐAČA VELIKIH RAČUNARA

Krajem dvadesetog veka, softverski razvoj u Japanu većinom su radili proizvođači velikih računara i njihove filijale, tako da su to mesta gde nalazimo najviše ponovnog korišćenja softvera. Krajem 1980-ih godina, Nipon Electric Company (NEC), Hitachi, Fujitsu i Toshiba su svi osnovali integrisana razvojna okruženja za softver, koja su podržavala višekratnu upotrebljivost.

NEC-ova laboratorija za softversko inženjerstvo u Tokiju započela je svoj program ponovnog korišćenja analiziranjem svojih poslovnih aplikacija. NEC je standardizovao 32 logička šablona i 130 zajedničkih algoritama, zatim je osnovana biblioteka višekratno upotrebljivih komponenti da bi klasifikovali, uvela u katalog, dokumentovala i komponente držala na raspolaganju. Ta laboratorija je deo NEC-ove arhitekture softverskog inženjerstva, projektovane da sprovede višekratnu upotrebljivost u svim razvojnim aktivnostima. Koristeći tri obimna projekta kao kontrolisane eksperimente, NEC je postigao prosečnu produktivnost koja je bila najmanje sedam puta veća od one na projektima koji nisu koristili višekratno upotrebljivi softver. Prosečan kvalitet proizvoda je skoro utrostručen (Nipon Electric Company 1987).

Hitachi se takođe usredsredio na poslovne aplikacije pisane na jezicima COBOL i PL/I. U kompaniji se koristilo integrisano softversko okruženje nazvano Eagle, koje je dozvoljavalo softverskim inženjerima da ponovo koriste standardne programske obrasce i funkcionalne procedure (približno 1 600 stavki podataka i rutina za ulazno/izlazno pretvaranje podataka i validaciju). Obrasci su korišćeni kao šabloni, a projektanti su dodavali svoj sopstveni kod da bi ih menjali (Tsuda i dr. 1992).

Fujitsu je osnovao Information Support Center (ISC) za elektronske komutacione sisteme. ISC je redovna biblioteka koja zapošljava analitičare sistema, softverske inženjere i stručnjake iz oblasti višekratne upotrebljivosti iz komutacionih sistema. Biblioteka ima referentni desk, unakrsno referenciranje projektante i pisce koda, softversku arhivu i komercijalni softver. U razvojnom procesu svih softverskih projekata moralo je da se koristi ISC i to se odnosilo na sve članove ISC-ovog osoblja u sve projekte i pregleđe softvera. Pre nego što je započet ISC program, samo 20% od 300 projekata bili su završeni po planu. Od uvođenje programa, 70% projekata bilo je završeno na vreme (Fujitsu 1987).

Dugoročna korist je poboljšana interoperabilnost sistema. Objavljeni su opisi pozitivnih efekata koje višekratna upotrebljivostmože da ima na razvoj; vrlo malo ih je govorilo o zamkama i nedostacima. Primer dobre analize ponovnog korišćenja je Limova (1994), u kojoj se opisuju napori na ponovnom korišćenju, preduzeti kod Hevlett-Packarda. Njegova pažljiva procena dva velika programa višekratne upotrebljivosti pokazuje značajna povećanja kvaliteta i produktivnosti i veliko smanjenje vremena prodaje. U tabeli 12.7 dat je sumarni pregled njegovih nalaza.

TABELA 12.7 Kvalitet, produktivnost i vreme prodaje kod Hewlett-Packarda (prilagođeno iz Lima 1994) ©1996 IEEE

Karakteristika projekta	Hewlett-Packard Projekat 1	Hewlett-Packard Projekat 2
Veličina	1100 nekomentarisanih izvornih iskaza	700 nekomentarisanih izvornih iskaza
Kvalitet	51% smanjenje greške	24% smanjenje greške
Produktivnost	57% povećanje	40% povećanje
Vreme za prodaju	Podaci neraspoloživi	42% smanjenje

On je takođe uporedio troškove sa višekratnom upotrebljivošću i troškove izrade koda od početka u tri projekta. U tabeli 12.8 prikazani su njegovi rezultati, koji nagoveštavaju da u stvari postoje značajni dodatni troškovi u stvaranju višekratno upotrebljivog koda i u njegovoj upotrebi. Dakle, moramo da uravnotežimo prednosti kvaliteta i produktivnosti sa tim dodatnim ulaganjem u komponente koje se mogu ponovo upotrebiti.

TABELA 12.8 Troškovi proizvodnje i višekratne upotrebljivosti kod Hewlett-Packarda (Lim 1994) © 1996 IEEE

	Sistem za kontrolu vazdušnog saobraćaja (%)	Sistem za upravljanje menijima i formularima (%)	Grafički firmver (%)
Relativni troškovi da se napravi višestruko upotrebljiv kôd	200	120 do 180	111
Relativni troškovi ponovnog korišćenja	10 do 20	10 do 63	19

Joos (1994) nudi pogled na upravljačku stranu pokretanja programa višekratne upotrebljivosti. U objašnjavanju pilot studija višekratne upotrebljivosti u kompaniji Motorola, ona ističe potrebu za dobrom obukom, upravljanjem očekivanjima i dobijanjem unapred definisane obaveze da se komponente višekratno koriste. U podsetniku 12.5 opisani su ključni činioci uspeha u NTT-u.

Pfleeger (1996) takođe predstavlja priču o ponovnoj upotrebljivosti, formulišući mešavinu iskustava sa programima višekratne upotrebljivosti koji nisu bili tako uspešni kao oni koje smo do sada opisali. Ona nagoveštava više ključnih lekcija koje su dobijene iz tih pokušaja:

- Ciljevi višekratne upotrebljivosti treba da budu merljivi, tako da možemo da kažemo da li smo ih ispunili.
- Ciljevi višekratne upotrebljivosti mogu da budu kontradiktorni, pa rukovodstvo mora na vreme i jasno da ih razreši. Na primer, kao što je pokazano na slici 12.7, cilj koji menadžer sektora želi da postigne višekratnom upotrebljivošću može da bude u sukobu sa ciljem menadžera projekta, pa da se nikada ne završi nijedan takav projekat.
- Različite perspektive mogu da stvore različita pitanja. Dok programeri postavljaju specifična tehnička pitanja, kao što je: „Da li je rutina za sortiranje napisana na jeziku C++?", potpredsednik se usredsređuje na korporaciju kao celinu i njen položaj na tržištu pitajući: „Da li smo već uštedeli pare?". Čak i kada su pitanja u suštini ista, možemo da dobijemo različite odgovore. Različite tačke gledišta odražavaju različite prioritete za učesnike u programu višekratne upotrebljivosti.

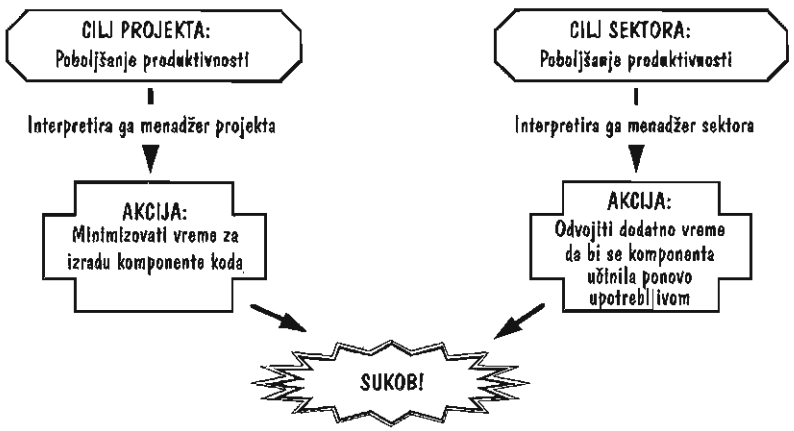
- Svaka organizacija mora da odluči na kom nivou se određena ključna pitanja postavljaju i na njih odgovara. Na primer: ko plaća troškove višekratne upotrebljivosti? Kada se troškovi unesu u projekat u nadi da će se oni nadoknaditi u budućim projektima, oni moraju da odražavaju udružene izbor i nameru. Ko je vlasnik višekratno upotrebljivih komponenti? Ko je njihov vlasnik i ko je odgovoran za njihovo održavanje mora da se odredi na višem nivou. Ko pravi komponente? Drugi nivo mora da bude odgovoran za konstrukciju, punjenje i podršku biblioteka za višekratne upotrebljivosti, kao i za merenje podataka koje one generišu. Ta pitanja su povezana sa načinom na koji je organizovano poslovanje, tako da su brige oko profita i gubitaka u vezi sa strategijama ulaganja i stavljanja u funkciju. Pitanja i odgovori moraju da budu podržani merenjem, pa je važno da se zna ko pita i kome treba odgovor.

PODSETNIK 12.5 KRITIČAN USPEH VIŠEKRATNE UPOTREBLJIVOSTI U NTT-U

Kompanija Nippon Telephone and Telegraph započela je projekat ponovnog korišćenja u svojim softverskim laboratorijama koji je trajao četiri godine i obuhvatio 600 inženjera. Kada su implementirali višekratnu upotrebljivost, oni su mnogo naučili o značaju dobrog upravljanja. Otkrili su da su kritični činioci njihovog uspeha bili (Isoda 1992):

- obavezivanje višeg rukovodstva;
- biranje odgovarajućih ciljnih domena;
- sistematski razvoj višekratno upotrebljivih modula zasnovan na analizi domena;
- ulaganje višegodišnjeg napora u višekratnu upotrebljivost.

- Integrišite procese višekratne upotrebljivosti u proces razvoja, ili učesnici neće znati kada se od njih očekuje da se drže višekratne upotrebljivosti.
- Povežite merenja sa procesom višekratne upotrebljivosti da biste mogli da merite svoj proces i da ga poboljšate. Neka vaši poslovni ciljevi nagoveste šta da se meri.



SLIKA 12.7 Sukobljavanje interpretacije ciljeva

Pfleeger (1996) takođe postavlja pitanja na koja mora da se odgovori da bi se princip višekratne upotrebljivosti uspešno sproveo:

- Da li imate pravi model višekratne upotrebljivosti?
- Šta su kriterijumi za uspeh?
- Kako postojeći modeli troškova mogu da se doteraju da bismo posmatrali zbirke projekata, a ne samo pojedinačne projekte? Odluke o mogućim uštedama u troškovima zahtevaju tehnike procene troškova koje obuhvataju više projekata. Ali, većina komercijalnih modela troškova se usredsređuje na pojedinačne projekte, a ne na zbirke projekata potrebne da bi se opravdalo ulaganje u višekratnu upotrebljivost. Potrebno je modifikovati postojeće ili izraditi nove modele troškova, koji mogu da integrišu aspekte višestrukih projekata da bi se podržale odluke o višekratnoj upotrebljivosti.
- Kako se uobičajeni pojmovi iz računovodstva (kao što su povraćaj ili ulaganje) uklapaju u višekratnu upotrebljivost? Za one koji su visoko u poslovodstvu ulaganje u višekratnu upotrebljivost isto je što bilo koja druga investicija. Njih zanimaju opcije, indikatori i povraćaj od investicije, a ne broj računara, jezici ili tehnike pretraživanja biblioteke. Za softverske inženjere je važno da terminologija višekratne upotrebljivosti može da se prevede u računovodstvenu terminologiju da bi investicija u vušekratnu upotrebljivost mogla da se uporedi sa drugim investitorskim inicijativama u korporaciji.
- Ko je odgovoran za kvalitet komponente? Šema mora da se analizira i organizuje tako da odgovara korporativnoj kulturi. Šta se dešava kada autor napusti kompaniju? Šta se događa kada se komponenta razvija ili se umnožava u više verzija? Koji kriterijumi se uspostavljaju za kvalitet komponente, kako za početno prihvatanje u biblioteci, tako i za održavanje komponente kako se ona menja u toku vremena?
- Ko je odgovoran za kvalitet procesa i održavanje? Višekratna upotrebljivost je više od uspostavljanja biblioteka i njihovog punjenja komponentama. Neko treba da nadgleda nivo ponovnog korišćenja, obezbeđujući da se dobre komponente identifikuju, ispravno obeleže i često koriste. Neko treba da probere neiskorišćene komponente i analizira ih da bi odredio zašto se one ne koriste. Neko mora da proceni proces analize domena da bi odredio da li su ciljne komponente stvarno ponovo upotrebljive. A neko treba da proučava efekat ponovnog korišćenja na nivou cele korporacije, da bi se izračunao povraćaj od investicije i osmišljavala buduća politika ponovnog korišćenja. Te aktivnosti se mogu obavljati na različitim nivoima u korporaciji i moraju da se prepoznaju kao neophodni delovi programa vipekratne upotrebljivosti.

12.5 PROCENJIVANJE PROCESA

Veliki deo prakse softverskog inženjerstva opisanog u ovoj knjizi obuhvata procese čija je svrha poboljšavanje softvera na neki način. Neki procesi, kao što su inspekcija ili čista soba, treba da poboljšaju naš proizvod na direktan i dramatičan način. Drugi procesi, kao što je upravljanje konfiguracijom ili projektom, indirektno utiču na proizvode, dajući nam više kontrole i razumevanja o uticaju naših radnji na rezultujući kôd.

Rad potreban za postavljanje procesa je promenljiv. Neki procesi obuhvataju čitav životni ciklus razvoja softvera, dok se drugi usredsređuju na malu grupu aktivnosti. U

oba slučaja, rad ne sme da bude uzaludan; želimo da naš proces bude efektivan i efikasan. Na primer, videli smo u poglavlju 8 da možemo da izračunamo efikasnost testiranja poređenjem broja pronađenih grešaka sa ukupnim brojem grešaka otkrivenim tokom celokupnog životnog ciklusa. A klaisfikaciono stablo na slici 8.20 pokazuje da velike komponente čije su konstrukcije pregledane verovatno neće imati mnogo grešaka; analiza klasifikacionog stabla pokazuje da je proces pregleda bio delotvoran. U ovom poglavlju, ispitujemo više drugih tehnika za procenjivanje efekata koje naši procesi imaju na proizvode i na ljude koji ih realizuju.

Analiza post mortem

Svaki projekat je sastavljen od nizova procesa, od kojih je svaki projektovan da ispuni određeni cilj. Na primer, inženjering zahteva ima za cilj formulisanje i kompoziciju zahteva koji su dosledni i potpuni; tehnike pregleda imaju za cilj da otkriju greške znatno pre testiranja. Jedan od načina da procenimo projekat jeste da prikupimo velike količine podataka, ne samo u toku razvoja, nego i po završetku projekta; onda možemo da analiziramo podatke da bismo odredili da li ispunjavamo ciljeve i potražiti područja mogućih poboljšanja.

Petroski (1985) nas podseća da mnogo učimo iz naših uspeha, ali čak i više iz neuspeha. Podaci o neuspehu su potrebni da bi nam predstavili uravnoteženu sliku iz koje možemo da gradimo modele, pa je izvođenje post mortem analize od suštinske važnosti za praksu softverskog inženjerstva. U podsetniku 12.6 navedene su brojne prednosti koje se mogu izvući iz grešaka iz prošlosti.

Analiza post mortem je procena svih aspekata projekta posle implementacije, uključujući proizvode, procese i resurse, čija je svrha da identifikuje oblasti poboljšanja za buduće projekte. Analiza se obično obavlja u kratkom vremenu posle završetka projekta; međutim, u Kumarovom pregledu (1990) napominje se da ona može da se obavi u bilo koje vreme, od trenutka isporuke do 12 meseci kasnije (pogledajte tabelu 12.9).

TABELA 12.9 Kada je urađena post mortem procena

Vremenski period	Procenat odgovaranja (od 92 organizacije)
Baš pred isporuku	27,8
Prilikom isporuke	4,2
Jedan mesec posle isporuke	22,2
Dva meseca posle isporuke	6,9
Tri meseca posle isporuke	18,1
Četiri meseca posle isporuke	1,4
Pet meseci posle isporuke	1,4
Šest meseci posle isporuke	13,9
Dvanaest meseci posle isporuke	4,2

PODSETNIK 12.6 KOLIKO ORGANIZACIJA IZVODI POST MORTEM ANALIZU

Kumar (1990) je anketirao 462 organizacije srednje veličine (izabrane između vrhunskih 500 prema kanadskom indeksu Dunna i Bradstreeta) koje su razvijale softver za upravljačke informacione sisteme. Od 92 organizacije koje su odgovorile, više od jedne petine nije pravilo nikakvu post mortem analizu. Od onih koje jesu, analize su rađene na manje od polovine projekata u organizaciji. Kumar je pitao menadžere zašto analize nisu rađene. Odgovori su se odnosili na nedostatak ljudi, nekvalifikovanost ljudi, nepostojanje kriterijuma za procenu i pritisak posla. Međutim, oni koji su odgovorili, navili su više koristi od post mortem analiza:

- utvrđeno je da je instalirani sistem ispunio sistemske zahteve;
- obezbeđena je povratna sprega za osoblje koje je razvijalo sistem;
- opravdano je usvajanje, nastavak ili odustajanje od instaliranog sistema;
- razjašnjeni su i uspostavljeni prioriteti za potrebne promene sistema;
- prenesena je odgovornost za sistem sa projektanata na korisnike;
- napravljen je izveštaj rukovodstvu o efikasnosti sistema;
- procenjene su i prečišćene kontrole sistema;
- obezbeđena je povratna sprega za promenu metoda razvoja;
- verifikovana je ekonomska isplativost sistema;
- zatvoren je projekat razvoja;
- obezbeđena je povratna sprega za promenu metoda upravljanja projektom;
- procenjeno je osoblje projekta.

Kako napominju Collier, DeMarco i Fearey (1996), „otkriti ponašanje koje mora da se promeni nije trivijalan zadatak u složenim sistemima, posebno ne u velikim i dugotrajnim projektima". Oni predlažu post mortem proces koji je pozitivan, koji ne optužuje i koji ohrabruje komunikaciju između učesnika. Njihove sugestije se zasnivaju na više od 22 post mortem analize koje su uključile preko 1300 članova projekata. Proces ima pet delova:

1. Dizajn i objavljivanje pregleda projekta radi prikupljanja podataka bez ugrožavanja poverljivosti;
2. Prikupljanje objektivnih informacija o projektu, kao što su troškovi resursa, granični uslovi, predvidljivost rokvoa i broj grešaka;
3. Sprovođenje istražnih sastanaka sa ciljem prikupljanja informacija koje je pregled propustio;
4. Uvođenje „dana istorije projekta" sa podskupom učesnika u projektu, radi pregleda projektnih događaja i podataka i sticanje ključnih uvida;
5. Objavljivanje rezultata sa usredsređivanjem na lekcije iz prošlosti.

Anketa. Anketa je polazna tačka zato što tako dobijeni odgovori mogu da predstavljaju usmerenje za ostatak post mortem analize. Ona definiše opseg analize i dozvoljava da dobijemo informacije koje se ukrštaju sa interesima članova projektnog tima. Postoje tri vodeća principa za administriranje anketa: ne pitajte više nego što vam je potrebno, ne postavljajte sugestivna pitanja i sačuvajte anonimnost. Prva od tih smernica je posebno značajna. Želimo da minimizujemo vreme koje je ispitaniku potrebno da odgovori na pitanja u anketi, tako da ima izgleda da više članova projekta popuni i vrati upitnik.

Podsetnik 12.7 sadrži primere iz ankete koju su administrirali Collier, DeMarco i Fearey (1996). Odgovori iz ankete odražavaju mišljenja i stavove članova tima.

PODSETNIK 12.7 UZORAK PITANJA IZ ANKETE U FIRMI WILDFIRE (COLLIER, DE MARCO I FEAREY 1996) © 1996 IEEE

U kompaniji Wildfire Communications razvijena je anketa za pomoć u post mortem analizi; veb pokazivač na potpun pregled nalazi se u ključnim referencama za ovo poglavlje. Anketa sadrži osam kategorija pitanja, sa sledećim primerima:

Kategorija 1: Podrška i ciljevi

Primer stanja: Da li su međusektorske linije odgovornosti jasno definisane u projektu?

[ ] uvek [ ] ponekad [ ] retko [ ] nikad

Kategorija 2: Očekivanja i komunikacija

Primer pitanja: Da li je na sastancima u vezi s projektom efikasno iskorišćeno vaše vreme?

[ ] uvek [ ] ponekad [ ] retko [ ] nikad

Kategorija 3: Razrešavanje problema

Primer pitanja: Da li vam je dozvoljeno da učestvujete u raspravama o problemima koji utiču na vaš rad?

[ ] uvek [ ] ponekad [ ] retko [ ] nikad

Kategorija 4: Pristup informacijama

Primer pitanja: Da li su promene u rokovima i odluke u vezi sa tim obuhvatale prave ljude?

[ ] uvek [ ] ponekad [ ] retko [ ] nikad

Kategorija 5: Specifikacije proizvoda

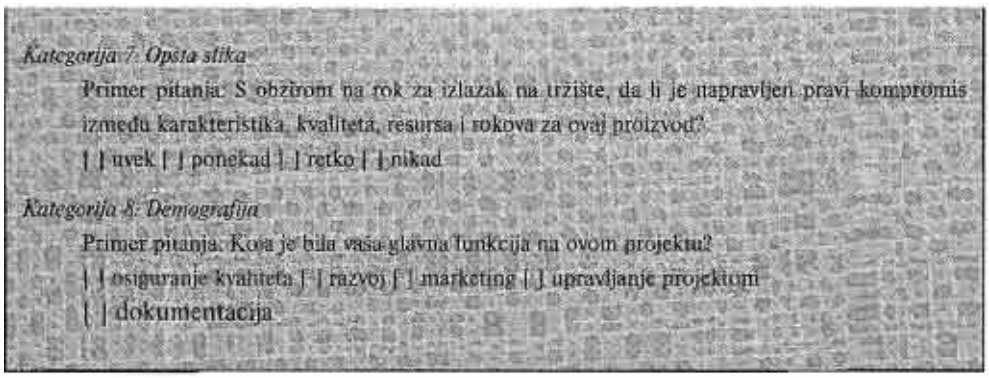
Primer pitanja: Da li su projekat definisali odgovarajući pojedinci?

[ ] uvek [ ] ponekad [ ] retko [ ] nikad

Kategorija 6: Inženjerska praksa

Primer pitanja: Da li je bio efikasan proces izgradnje komponenti na kojima ste radili?

[ ] uvek [ ] ponekad [ ] retko [ ] nikad


Kategorija 7: Opšta slika

Primer pitanja: S obzirom na rok za izlazak na tržište, da li je napravljen pravi kompromis između karakteristika, kvaliteta, resursa i rokova za ovaj proizvod?

[ ] uvek [ ] ponekad [ ] retko [ ] nikad

Kategorija 8: Demografija

Primer pitanja: Koja je bila vaša glavna funkcija na ovom projektu?

[ ] osiguranje kvaliteta [ ] razvoj [ ] marketing [ ] upravljanje projektom [ ] dokumentacija

Suština je da razmišljamo o tabelarnom prikazivanju rezultata pre nego što podelimo upitnik. Ponekad proces tabeliranja i analize rezultata sugeriše da treba preformulisati pitanje da bi se razjasnilo ili proširilo. Štaviše, ta pitanja se postavljaju za svaki projekat, pa moramo da budemo sigurni da su dovoljno opšteg karaktera. Zbir svih odgovora sa velikog broja projekata nam omogućava da uočimo trendove, odnose i oblasti otvorene za poboljšanja.

Objektivne informacije. Zatim, potrebne su nam objektivne informacije da bismo komplementirali mišljenja izražena u anketi. Ovde opet želimo da prikupljamo podatke na jednostavan način, koji olakšava izvođenje ukrštenih poređenja u projektu. Collier, deMarco i Fearey (1966) predlažu tri vrste merenja: troškove, raspored i kvalitet. Na primer, merenja troškova bi mogla da uključe:

- rad izražen u mesecima po čoveku, za glavne uloge ili aktivnosti;
- ukupan broj redova koda, najbolje po funkciji;
- broj redova koda koji su promenjeni ili dodati, po funkciji;
- broj interfejsa: ukupno, dodatih, promenjenih ili izbrisanih.

Merenje rokova bi moglo da uključi izveštaj o prvobitnim rokovima, istorijatu događaja koji su prouzrokovali odstupanje od rokova i analizu tačnosti predviđanja rokova. Najzad, kvalitet može da se meri kao broj grešaka pronađenih za vreme svake razvojne aktivnosti i opisivanjem brzine kojom su greške bile pronađene i otklonjene.

U idealnom slučaju, većina tih informacija je već na raspolaganju, pošto su prikupljene u toku razvoja i održavanja. Ali, neke organizacije vrše merenja bolje od drugih. Proces post mortem može da podstakne timove da učine više na sledećem projektu kada shvate da na važna pitanja može da se odgovori sa vrlo malo dodatnog rada na prikupljanju i održavanju podataka. Štaviše, ponovljena merenja su korisnija od onih jednokratnih. Merenje veličine ili promena rasporeda u toku vremena daje timu bolju sliku o napredovanju nego jedan snimak stanja napravljen u sredini ili na kraju razvojnog procesa. Na taj način, čak i ako post mortem analitičari ne mogu da prikupe sve što bi voleli da vide, njihova pitanja mogu da inspirišu poboljšanja na kasnijim projektima.

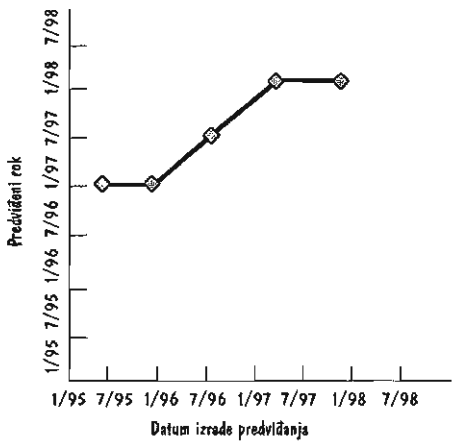
Radni sastanci. Radni sastanci dozvoljavaju članovima tima da izveštavaju o tome šta su uradili i šta nije išlo dobro na projektu. U isto vreme, vođe projekata mogu da ispituju dublje, pokušavajući da identifikuju osnovni uzrok, kako pozitivnih, tako i negativnih efekata. Članovi tima često postavljaju pitanja koja nisu obuhvaćena pitanjima iz

ankete i na taj način često otkrivaju značajne odnose koji se nisu pokazali tokom razvoja. Na primer, članovi tima mogu da istaknu probleme sa primenom neke metodologije za formulisanje zahteva kod nekih kupaca jer ona ne omogućuje lako razumevanje kupčevih zahteva. Ili, ljudi iz tima za testiranje mogu da raspravljaju o problemima sa kojima su se sreli kada je trebalo proceniti performansu na razvojnoj platformi koja nije ista kao radna platforma.

Radni sastanak bi trebalo da bude slobodnije strukturisan i da predsedavajući podstiče učešće i prati raspravu. Za veoma velike razvojne timove, radni sastanak bi možda bilo bolje organizovati kao niz manjih sastanaka na kojima broj učesnika nije veći od približno 30. Glavna korist od istražnog sastanka je sposobnost članova tima da odbace uvrede i usmere se ka poboljšanju aktivnosti.

Dan istorije projekta. Za razliku od radnih sastanaka, dan istorije projekta obuhvata ograničen broj učesnika. Njegova svrha je da se identifikuju osnovni uzroci ključnih problema doživljenih na projektu. Dakle, učesnici su samo oni koji znaju nešto o tome zašto su se pojavile pukotine u rokovima, kvalitetu i resursima. Zato danu istorije projekta mogu da prisustvuju i ljudi koji nisu direktno učesnici u razvojnom timu; dobri kandidati su ljudi iz marketinga, kupci, rukovodioci projekata i hardverski inženjeri.

Učesnici se pripremaju za dan pregledajući sve što znaju o projektu: svoju korespondenciju, dijagrame upravljanja projektom, informacije iz pregleda, podatke merenja i sve drugo što je u vezi sa događajima u projektu. Prva formalna aktivnost u danu istorije projekta je pregled svih dijagrama sa predviđenim rokovima, kao što je prikazano na slici 12.8. Za svaki ključni miljokaz u projektu na dijagramu se vidi datum kada je predviđanje rađeno i datum za koji je predviđen završetak određene faze. Na primer, na slici se vidi da je u julu 1995. godine predviđeno da će se faza završiti u januaru 1997. godine. To predviđanje je bilo isto i u januaru 1996. godine, ali, kako se vreme primicalo januaru 1997. godine, predviđeni rok je pomeren na juli 1997. godine. Zatim, kada ni u julu 1997. godine faza nije završena, predviđeno je da će se to dogoditi u januaru 1998. godine. Najzad, faza se zaista završila u januaru 1998. godine. Oblik dijagrama predviđanja rokova nam govori nešto o optimizmu ili pesimizmu u našim procenama i pomaže nam da razumemo potrebu da se tačnije procenjuje. Idealna situacija predstavlja se horizontalnom linijom.



SLIKA 12.8 Dijagram sa predviđenim rokovima

Dijagrami predviđanja za rokove u projektu mogu da se koriste kao ilustracije koje prikazuju gde su se pojavili problemi. Oni rasplamsavaju raspravu o mogućim uzrocima svakog problema, a tim se usredsređuje na identifikovanje iscrpne spiskove uzroka. Zatim, koristeći objektivne podatke kao podršku svakom argumentu, tim sužava svaki spisak uzroka, dok tačno ne shvati zašto se problem pojavio. Collier, DeMarco i Fearey (1996) izveštavaju da početna spisak uzroka ponekad može da dostigne i do 100 stavki i da može biti potrebno više sati da se analizira šta se stvarno dogodilo. Na kraju dana istorije projekta, tim ima prioritetni spisak od približno 20 osnovnih uzroka.

Objavljivanje rezultata. Poslednja faza je da se tako stečena saznanja prenesu ostatku projektnog tima. Umesto održavanja novog sastanka, učesnici u danu istorije projekta pišu otvoreno pismo menadžerima, svojim kolegama i drugim projektantima. Pismo se sastoji od četiri dela. Uvod sadrži opis projekta u kome se objašnjava generalni tip projekta i sve one informacije koje pokazuju da li je projekat u nečem jedinstven i zašto. Na primer, pismo može da opisuje projekat kao izgradnju naplatnog sistema u telekomunikacijama, što je nešto po čemu je kompanija dobro poznata. Ali u tom posebnom projektu, po prvi put je upotrebljen programski jezik Eiffel, u sprezi sa alatima za podršku objektno orijentisanom dizajnu.

U pismu se dalje sumiraju svi pozitivni nalazi post mortem analize. Nalazi mogu da opisuju ne samo ono što je urađeno dobro, već i ono što se može upotrebiti u budućim projektima. Na primer, u projektu su možda napravljeni višekratno upotrebljv kôd, novi alati, ili preporuke za uspešno korišćenje programskog jezika Eiffel, a sve to može biti korisno za buduće slične projekte.

Zatim pismo sumira tri najgora faktora koji su onemogućili projektnom timu da postigne postavljene ciljeve. Ti faktori su obično tri najviše rangirane stavke u spisku prioritetnih uzroka koji je napravljen u toku dana istorije projekta.

Najzad, u pismu se predlažu aktivnosti za poboljšanje. Collier, DeMarco i Feary (1996) predlažu da tim odabere jedan problem koji je toliko važan da mora da se reši pre nego što počne rad na novom projektu. U pismu bi trebalo da se jasno opiše problem i predloži način njegovog rešavanja. Opis problema i rešenje bi trebalo podržati objektivnim merenjima, tako da projektanti mogu da procene obim problema i prate promene kroz poboljšanja.

Arango, Schoen i Pettengill (1993) nude širi pristup objavljivanju rezultata post mortem analize. U njihovom radu u kompaniji Schlumberger, razmatrano je ponovno korišćenje svega iz projekta, uključujući i pouke iz prošlosti. Schlumbergerovi istraživači su razvili tehnologiju knjige projekta i knjige tehnologije, koje mogu da koriste i projektanti na drugim projektima, čime se dele iskustva, alati, konstrukcije, podaci, ideje i sve drugo što bi moglo da bude korisno nekom drugom u kompaniji. Korišćenjem tehnologije kao što je njihova, možemo da učimo jedni od drugih i da sa svakim novim projektom budemo sve bolji, a ne da pravimo iste greške i zatim se čudimo zašto.

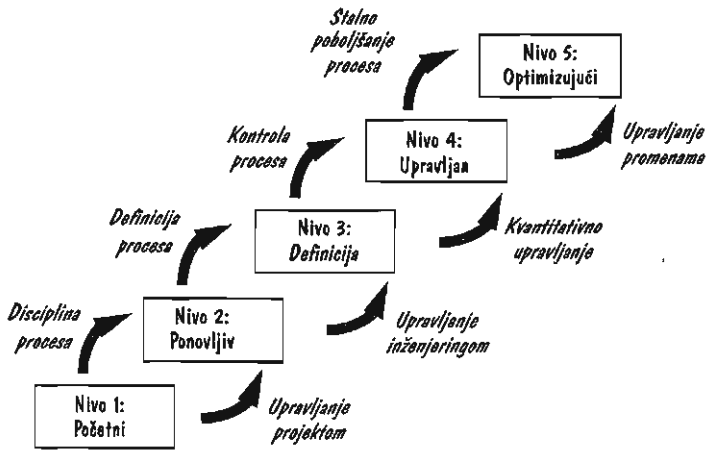
Modeli zrelosti procesa

U 1980-im godinama, podstaknute IBM-ovim radom, više organizacija je počelo da ispituje proces razvoja softvera kao celinu, a ne da se usredsređuju na pojedinačne aktivnosti. Više istraživača je pokušalo da formuliše šta je to što proces čini efikasnim. Iz tog rada je proizašao pojam **zrelosti procesa**, što znači da proces razvoja poseduje povratnu spregu i kontrolne mehanizme tako da se veoma kvalitetni proizvodi pojavljuju na vreme i bez velikih iznenađenja za rukovodstvo.

Model zrelosti sposobnosti. Model zrelosti sposobnosti (*Capability Maturity Model,* CMM) je razvijen od strane američkog Instituta za softversko inženjerstvo (*Software Engineering Institute,* SEI) sa ciljem podrške Ministarstvu odbrane SAD prilikom procene kvaliteta ugovarača. CMM je, inspirisan radovima Deminga (1989), počeo je kao **model zrelosti procesa** u kome je razmatrana organizacija vrednovana na skali od 1 (nizak) do 5 (visok), na osnovu odgovora na 110 pitanja vezanih za primenjivani proces razvoja. Na slici 12.9 prikazan je sumarni pregled uzdizanja od nižih nivoa zrelosti ka višim nivoima zrelosti.

U tabeli 12.10 navedeno je 12 pitanja neophodnih za procenu nivoa 2 (ponovljiv); ako se na bilo koje od tih pitanja odgovori sa „ne", onda se organizacija automatski procenjuje kao organizacija nivoa 1, bez obzira na odgovore na ostalih 98 pitanja.

Ovaj pristup je sadržavao niz problema, pa je razvijen CMM s ciljem da se oni reše i zameni model zrelosti procesa. Mnogi od osnovnih principa prvobitnog pristupa zrelosti procesa su zadržani: CMM koristi upitnik da bi se procenila zrelost razvojnog projekta, upitnik je dopunjen zahtevom da se daju dokazi koji potvrđuju odgovore, a ocenjuje na skali od pet stepeni. CMM opisuje principe i praksu za koje se uzima da daju bolje softverske proizvode, organizovan je u pet nivoa, čime obezbeđuje put ka boljem uočavanju i kontroli procesa i ka poboljšanim proizvodima koji iz toga proizilaze. Model ima dvojaku namenu: koriste ga potencijalni kupci da bi identifikovali jake i slabe tačke svojih dobavljača i koriste ga projektanti da bi procenili sopstvene sposobnosti i sagledali puteve koji vode do poboljšanja.



SLIKA 12.9 Nivoi zrelosti definisani od strane Instituta za softversko inženjerstvo

TABELA 12.10 Pitanja na koja se traže odgovori za nivo 1 modela zrelosti procesa

Broj pitanja	Pitanje
1.1.3	Da li funkcija obezbeđenja kvaliteta softvera poseduje kanal za izveštavanje rukovodstva odvojen od rukovodstva razvojem softverskog projekta?
1.1.6	Da li postoji funkcija upravljanja konfiguracijom softvera za svaki projekat koji uključuje razvoj softvera?
2.1.3	Da li se koristi formalni proces pri analizi svakog razvoja softvera od strane rukovodstva pre formulisanja ugovornih obaveza?
2.1.14	Da li se koristi formalna procedura za procenu veličine softvera?
2.1.15	Da li se koristi formalna procedura za izradu plana razvoja softvera?
2.1.16	Da li se primenjuju formalne procedure za procenu troškova razvoja softvera?
2.2.2	Da li se održavaju profili veličine softvera za svaki element konfiguracije softvera u vremenu?
2.2.4	Da li se prikupljaju statistički podaci o greškama prilikom kodiranja i testiranja softvera?
2.4.1	Da li više rukovodstvo poseduje mehanizam za redovne uvide u stanje projekata razvoja softvera?
2.4.7	Da li menadžeri prvog nivoa razvoja softvera potpisuju svoj plan rada i procene troškova?
2.4.9	Da li se koristi mehanizam za kontrolisanje promena u softverskim zahtevima?
2.4.17	Da li se koristi mehanizam za kontrolisanje izmena koda?

Svakom od pet nivoa sposobnosti pridružen je skup **ključnih oblasti procesa** na koje bi organizacija trebalo da se usredsredi kao deo svojih aktivnosti na poboljšanju. Prvi nivo modela zrelosti, **početni**, opisuje ad hoc ili čak haotičan proces razvoja softvera. To znači da su ulazi u proces slabo definisani; tamo gde se očekuju izlazi, prelazak od ulaza ka izlazima je nedefinisan i nekontrolisan. Slični projekti mogu značajno da variraju u pogledu produktivnosti i atributa kvaliteta, zbog nedostatka odgovarajuće strukture i kontrole. Za taj nivo zrelosti, teško je čak napisati ili predstaviti proces u celini; proces je izrazito reaktivan i loše definisan pa je uočljivost nikakva, a sveobuhvatno merenje teško. U projektu su možda postavljeni ciljevi poboljšanja kvaliteta i produktivnosti, ali menadžeri nisu svesni trenutnog nivoa kvaliteta i produktivnosti.

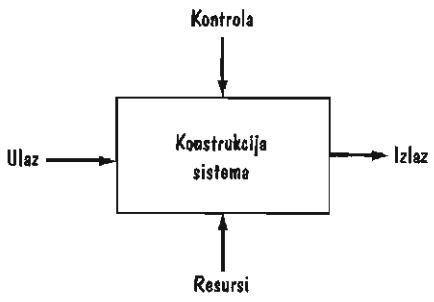
Kao što je prikazano u tabeli 12.11, na ovom nivou ne postoje ključne oblasti procesa. Samo neki procesi su definisani, a uspeh razvoja zavisi od napora pojedinaca, a ne od timskih ostvarenja. Organizacija na nivou 1 bi trebalo da se usredsredi na značajnije strukturisanje procesa i uvođenje više kontrola, delom i da bi se omogućila smislenija merenja.

Sledeći nivo je **ponovljiv**; kod njega se identifikuju ulazi i izlazi, ograničenja (kao što su budžet i raspored rada) i resursi koji se koriste da bi se izradio konačni proizvod. Osnovno upravljanje projektom prati troškove, raspored i funkcionalnost. Postoji određena disciplina među članovima tima, tako da se uspesi sa ranijih projekata mogu ponoviti na sličnim novim projektima. Ovde su ključne oblasti procesa uglavnom aktivnosti upravljanja koje pomažu razumevanje i kontrolu akcija i rezultata procesa.

TABELA 12.11 Ključne oblasti procesa u sklopu CMM (Paulk i dr. 1993a,b)

Nivo CMM	Ključne oblasti procesa
Početni	Nema
Ponovljiv	Upravljanje zahtevima Planiranje softverskog projekta Praćenje i uvidi u softverski projekat Upravljanje podugovaračima za izradu softvera Obezbeđenje kvaliteta softvera Upravljanje konfiguracijom softvera
Definisan	Usredsređenje na organizaciju procesa Definicija organizacije procesa Program obuke Integrisano upravljanje softverom Inženjering softverskog proizvoda Koordinacija između grupa Direktni pregledi
Upravljan	Kvantitativno upravljanje procesom Upravljanje kvalitetom softvera
Optimizujući	Prevencija grešaka Upravljanje promenama u tehnologiji Upravljanje promenama u procesu

Proces je ponovljiv u istom smislu u kome i potprogram: ispravni ulazi proizvode ispravne izlaze, ali nema uvida u to kako se izlazi formiraju. Ako od vas zatraže da definišete i opišete proces, vi i vaš razvojni tim ne možete da nacrtate više od dijagrama sa slike 12.10. Ta slika prikazuje ponovljiv proces kao pojednostavljeni dijagram tehnike strukturne analize i dizajna (SADT), sa ulazom na levoj strani, izlazom na desnoj, ograničenjima na vrhu i resursima na dnu. Na primer, zahtevi mogu da budu ulaz u proces, sa softverskim sistemom kao izlazom. Kontrolna strelica predstavlja takve stavke kao što su rokovi, budžet, standardi i upravljačke direktive, a strelica za resurse uključuje alate i osoblje.



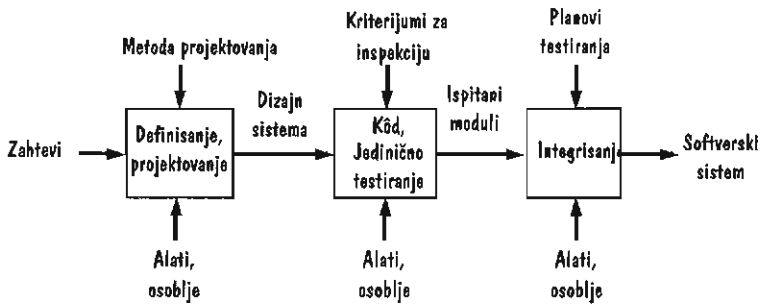
SLIKA 12.10 Ponovljiv proces (nivo 2)

Kako je moguće meriti samo ono što je uočljivo, na slici 12.10 se nagoveštava da merenja vezana za upravljanje procesom imaju najviše smisla kod ponovljivih procesa. Odnosno, pošto se vide samo strelice, svakoj strelici možemo da pridružimo mere. Tako za ponovljive procese mere ulaza mogu da uključe veličinu i stalnost zahteva. Izlazi mogu da se mere kao veličina sistema (funkcionalna ili fizička), resursi kao ukupan rad osoblja, a ograničenja kao troškovi i rokovi, u dolarima i danima, prema redosledu kojim smo ih naveli.

Poboljšanje ponovljivog procesa dovodi do **definisanog** procesa, gde se upravljačke i inženjerske aktivnosti dokumentuju, standardizuju i integrišu; rezultat je standardni proces za svakog u organizaciji. Mada neki procesi mogu da se razlikuju od ostalih, standardni proces je skrojen prema tim posebnim potrebama, a prilagođenje može da odobri rukovodstvo. Na tom nivou zrelosti, fokus ključnih oblasti procesa je organizovanje.

Definisan nivo zrelosti (nivo 3) se razlikuje od nivoa 2 po tome što obezbeđuje vidljivost u pravougaoniku „konstrukcije sistema" na slici 12.10. Na nivou 3, definisane su međuaktivnosti, a njihovi ulazi i izlazi su poznati i shvaćeni. Ta dodatna struktura znači da ulaz u i izlaz iz međuaktivnosti može da se testira, meri i procenjuje, s obzirom na to da su međuaktivnosti dobro definisane i vidljive. Na slici 12.11 prikazan je jednostavan primer definisanog procesa sa tri tipične aktivnosti. Međutim, različiti procesi mogu da se podele na više različitih funkcija ili aktivnosti.

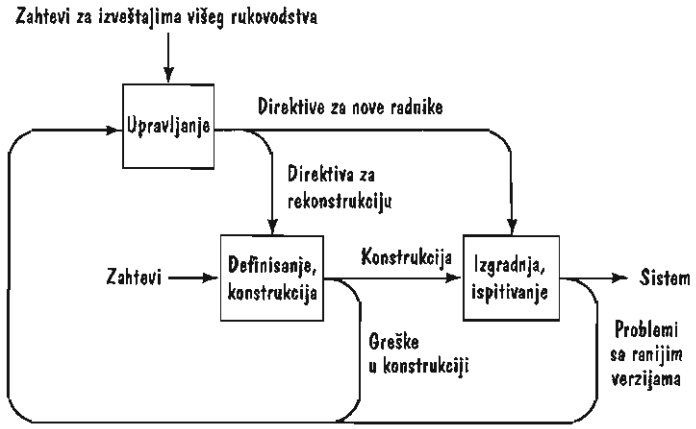
S obzirom na to da su u definisanom procesu aktivnosti opisane i razdvojene jedne od drugih, atribute proizvoda možemo da merimo najranije od nivoa 3. Greške otkrivene u svakoj vrsti proizvoda mogu da se prate i možemo da poredimo gustinu grešaka svakog proizvoda sa planiranim ili očekivanim vrednostima. Određenije, mere proizvoda dobijene u ranoj fazi mogu da budu korisni pokazatelji u odnosu na kasnija merenja. Na primer, može da se izmeri kvalitet zahteva ili konstrukcije i da se upotrebi za predviđanje kvaliteta koda. Takva merenja koriste vidljivost u procesu da bi obezbedila više kontrole nad razvojem: ako je kvalitet zahteva nezadovoljavajući, na njima može dodatno da se poradi pre nego što započnu aktivnosti dizajna. Takvo rano ispravljanje problema pomaže ne samo da se kontroliše kvalitet, nego i da se poboljša produktivnost i smanji rizik.



SLIKA 12.11 Definisani proces (nivo 3)

Upravljani proces usmerava trud ka kvalitetu proizvoda. Uvođenjem detaljnih merenja procesa i kvaliteta proizvoda, organizacija može da se usredsredi na korišćenje kvantitativnih informacija da bi problemi postali vidljivi i procenio se efekat mogućih

rešenja. Dakle, ključne oblasti procesa bave se kvantitativnim upravljanjem softverom, kao i upravljanjem njegovim kvalitetom.



SLIKA 12.12 Upravljani proces (nivo 4)

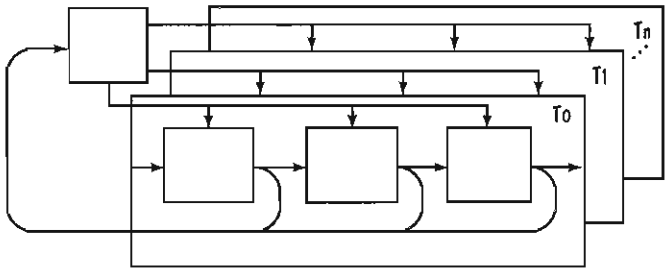
Kao što je prikazano na slici 12.12, možemo da koristimo povratne informacije s početka rada na projektu (na primer, problematične oblasti otkrivene u konstrukciji) da bismo uspostavili prioritete za aktivnosti koje se već odvijaju u projektu (na primer, rekonstrukciju) i aktivnosti koje će tek uslediti (na primer, iscrpnije preglede i testiranje programskog koda i promenjen redosled integracije). S obzirom na to da možemo da poredimo i nijansiramo, efekti promena u jednoj aktivnosti mogu da se prate i u drugima. Na nivou 4, raspored resursa se određuje na osnovu povratnih informacija; osnovne aktivnosti se same po sebi ne menjaju. Na tom nivou možemo da procenimo delotvornost procesnih aktivnosti: Koliko su efikasni pregledi? Upravljanje konfiguracijom? Obezbeđenje kvaliteta? Testiranje inicirano pojavom grešaka? Možemo da iskoristimo prikupljene mere da stabilizujemo proces, tako da produktivnost i kvalitet odgovaraju očekivanjima.

Značajna razlika između nivoa 3 i 4 je u tome što merenja koja se dobijaju na nivou 4 odražavaju karakteristike ukupnog procesa i interakcije između glavnih aktivnosti i u njima. Upravljački pregled se oslanja na bazu podataka sa merenjima koja može da obezbedi informacije o takvim karakteristikama kao što su raspodela grešaka, produktivnost i delotvornost zadataka, dodeljivanje resursa i verovatnoća da će se planirane i stvarne vrednosti podudarati.

Najpoželjniji nivo zrelosti sposobnosti je **optimizujući**, gde je kvantitativna povratna sprega ugrađena u proces, da bi proizvela njegovo stalno poboljšavanje. Posebno, novi alati i tehnike se ispituju i nadgledaju, da bi se videlo kako utiču na proces i proizvode. Ključne oblasti procesa obuhvataju prevenciju grešaka, upravljanje promenom tehnologije i upravljanje promenom procesa.

Da biste razumeli kako se nivo 5 poboljšava u odnosu na nivo 4, razmotrite sliku 12.13. Niz naslaganih pravougaonika pokazuje napredovanje procesa označenih sa *T0, T1,...,Tn.* Prvi pravougaonik pokazuje proces upotrebljen u trenutku *T0*, drugi upotrebljen u trenutku *T1* i tako dalje. U datom vremenskom trenutku, merenja aktivnosti se koriste da bi se poboljšao tekući proces, možda uklanjanjem i dodavanjem procesnih ak-

tivnosti i dinamičkim menjanjem strukture procesa kao reakcija na povratnu informaciju dobijenu merenjem; rezultat je prelazak na sledeći proces na dijagramu. Dakle, promena procesa može da utiče na organizaciju i projekat, kao i na proces. Rezultati iz jednog ili više procesa, koji su u toku ili su okončani, mogu takođe da dovedu do prečišćenog drukčijeg procesa razvoja za buduće projekte. Spiralni model je jedan od primera takvog dinamičkog menjanja procesa, koji odgovara na povratnu spregu iz ranih aktivnosti da bi se smanjio rizik u kasnijim.



SLIKA 12.13 Optimizujući proces (nivo 5)

Na primer, pretpostavimo da počinjemo razvoj standardnim pristupom vodopada. Kada su zahtevi definisani i razvoj počeo, merenja i verbalna povratna informacija mogu da ukažu na visok nivo neodređenosti u zahtevima. Na osnovu tih informacija, možemo da odlučimo da promenimo proces zasnovan na izradi prototipova zahteva i dizajna, tako da neka neodređenost može da se razreši pre nego što značajno uložimo u implementaciju postojećeg dizajna. Na taj način, sposobnost da optimizujemo proces daje nam maksimum fleksibilnosti u procesu razvoja. Merenja deluju kao senzori i monitori, a proces nije samo pod kontrolom, nego se i značajno menja, kao odgovor na signale upozorenja.

Važno je zapamtiti da zrelost sposobnosti ne obuhvata diskretni skup pet mogućih ocena. Umesto toga, zrelost predstavlja relativna mesta u kontinuumu od 1 do 5. Individualni proces se ocenjuje ili procenjuje po više dimenzija, a neki delovi procesa mogu da budu zreliji ili vidljiviji od drugih. Na primer, ponovljiv proces može da nema dobro definisane međuaktivnosti, ali aktivnost projektovanja može zaista jasno da se definiše i da se njom upravlja. Dijagrami vidljivosti procesa koje smo ovde predstavili namenjeni su za opšti opis tipičnih procesa. Od suštinske je važnosti da ispitate proces i odredite šta je vidljivo. Slike i tabele ne bi trebalo da zabrane aktivnosti samo zato što je ukupan nivo zrelosti određen kao ceo broj; ako je jedan deo procesa zreliji od ostatka, aktivnost ili alat može da poboljša vidljivost tog dela i pomogne da se ispune ukupni ciljevi projekta, u isto vreme dovodeći ostatak procesa na viši nivo zrelosti. Dakle, u ponovljivom procesu sa dobro definisanim projektom, mere kvaliteta dizajna mogu da budu odgovarajuće i poželjne, čak i ako nisu u opštem slučaju preporučljive na nivou 2.

CMM ima drugi nivo granularnosti koji nije prikazan u tabeli. Svaka oblast procesa podrazumeva skup **ključnih postupaka** čije prisustvo ukazuje na to da je projektant imple-

metirao i institucionalizovao domen procesa. Svrha ključnih postupaka jeste obezbeđenje dokaza da je procensi domen delotvoran, ponovljiv i dugotrajan (Paulk i dr. 1993b).

Ključni postupci se organizuju prema sledećim zajedničkim svojstvima:

- *Posvećenost realizaciji:* Koje akcije obezbeđuju formulisanje procesa i njegovo dalje korišćenje? Ta kategorija uključuje politiku rukovođenja i rukovodilačke postupke.
- *Sposobnost realizovanja:* Koji preduslovi obezbeđuju da je organizacija spremna da implementira proces? Ovi postupci imaju veze sa resursima, obukom, orijentacijom, alatima i organizacionom strukturom.
- *Preduzimanje aktivnosti:* Koje uloge i procedure treba da se implementiraju u ključnom procesnom domenu? U ovu kategoriju spadaju postupci s planovima, procedurama, izvedenim radom, korektivnim akcijama i praćenjem.
- *Merenja i analiza:* Koje procedure mere proces i analiziraju merenja? Postupci u ovoj kategoriji uključuju merenja procesa i analizu.
- *Verifikovanje implementacije:* Šta obezbeđuje usaglašenost aktivnosti s definisanim procesom? Postupci obuhvataju upravljačke preglede i revizije.

Za organizaciju se kaže da odgovara ključnom procesnom domenu samo kada je taj domen implementiran i institucionalizovan. Implementacija se utvrđuje odgovorima na pitanja o preduzetim aktivnostima; ostali postupci se bave institucionalizovanjem.

SPICE. Iz CMM-a je proisteklo mnoštvo metoda za ocenu procesa, od Trilliuma (koji su proizvele kanadske telekomunikacione kompanije) do BOOTSTRAP-a (proširenja CMM-a koje se razvilo u okviru projekta ESPRIT Evropske zajednice). Taj porast, kao i tehnike procene proizvoda koji su bili komercijalno osetljivi, doveli su do toga da je Ministarstvo odbrane Velike Britanije predložilo međunarodni standard za procenu procesa (Rout 1995). Namena novog standarda, nazvanog **SPICE** (*Software Proces Improvement and Capability dEterminationn*), bila je da usaglasi i proširi postojeće pristupe. Slično CMM-u, SPICE se preporučuje i za poboljšanje procesa i za određivanje sposobnosti. Okvir je izgrađen na arhitekturi procene koja definiše poželjne postupke i procese.

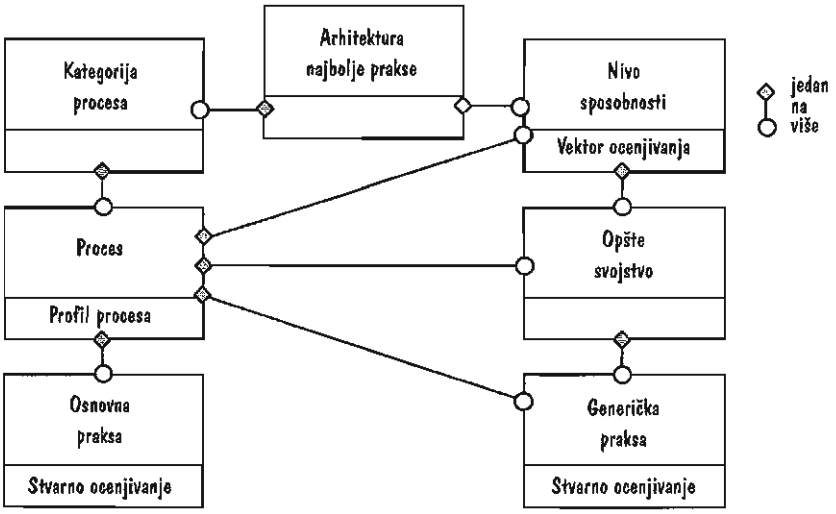
Postoje dve različite vrste postupaka:

1. *Osnovni postupci* koji su suštinske aktivnosti određenog procesa.
2. *Generički postupci* koji institucionalizuju ili implementiraju proces na opšti način.

Na slici 12.14 ilustruje se kako arhitektura SPICE povezuje te dve vrste i uključuje stvarno procenjivanje za svaku od njih. Na levoj strani dijagrama predstavljeni su funkcionalni postupci u razvoju softvera. Taj funkcionalni aspekt se bavi s pet vrsta aktivnosti:

1. *Za kupca:* procesi koji direktno utiču na kupca, podržavaju razvoj i isporuku proizvoda kupcu i obezbeđuju ispravan rad i upotrebu.
2. *Inženjering:* procesi koji definišu, implementiraju ili održavaju sistem i njegovu dokumentaciju.
3. *Projekat:* procesi koji ustanovljavaju resurse, koordiniraju ih ili upravljaju njima, ili obezbeđuju usluge kupcima.

4. *Podrška:* procesi koji ustanovljavaju ili podržavaju performansu drugih procesa.
5. *Organizacija:* procesi koji ustanovljavaju poslovne ciljeve i razvijaju sredstva za postizanje tih ciljeva.



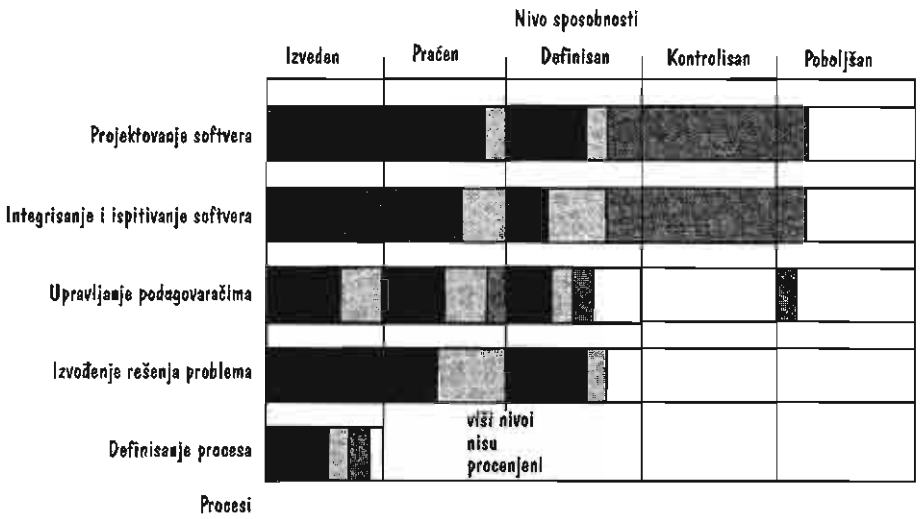
SLIKA 12.14 Arhitektura SPICE za procenu procesa (Rout 1995)

Na desnoj strani slike 12.14 dat je upravljački aspekt; generički postupci, koji su primenljivi na sve procese, raspoređeni su u šest nivoa sposobnosti:

0. *Neizvedeni:* izvođenje nije uspelo i nema ili nije moguće identifikovati proizvod koji funkcioniše.
1. *Neformalno izvedeni:* nisu planirani ni praćeni, zavise od znanja pojedinaca i proizvoda rada koji se mogu identifikovati.
2. *Planirani i praćeni:* verifikovani prema određenim procedurama, proizvodi rada podležu propisanim standardima i zahtevima.
3. *Dobro definisani:* dobro definisan proces koji koristi potvrđene, namenske verzije standarda, dokumentovani procesi.
4. *Kvantitativno kontrolisani:* detaljne mere performanse, sposobnost predviđanja, objektivno upravljanje i proizvodi rada koji se kvantitativno procenjuju.
5. *Stalno poboljšavani:* kvantitativni ciljevi za efikasnost i delotvornost zasnovani na poslovnim ciljevima, kvantitativne povratne informacije od definisanih procesa, plus isprobavanje novih ideja.

Izveštaj o proceni je jedna vrsta profila procesa; procenjuje se svaka oblast procesa i izveštava se da je na jednom od šest nivoa sposobnosti. Na slici 12.15 prikazan je primer kako se izveštava o profilu. Senčenje ukazuje na stepen zadovoljavanja aktivnosti na svakom nivou.

Dakle, dok se CMM bavi organizacijama, SPICE se bavi procesima. Kao i kod CMM-a, procena metodom SPICE se administrira na unapred pažljivo propisan način da bi se subjektivnost u ocenjivanju svela na najmanju meru.



SLIKA 12.15 Profil SPICE procene (Rout 1995)

ISO 9000. Međunarodna organizacija za standarde (ISO) izradila je grupu standarda koji su zajedno poznati kao **ISO 9000**. Standardi određuju akcije koje treba da se preduzmu u svakom sistemu (to jest, ne samo u softverskom sistemu) koji ima ciljeve i ograničenja kvaliteta. Posebno, ISO 9000 se primenjuje kada kupac zahteva od proizvođača da pokaže dati nivo stručnosti u projektovanju i izgradnji proizvoda. Kupac i proizvođač ne moraju da pripadaju različitim kompanijama; taj odnos može da postoji i unutar iste organizacije.

Među standardima ISO 9000, standard 9001 je najprimenljiviji na način kako razvijamo i održavamo softver (International Standards Organization 1987). On objašnjava šta kupac mora da uradi da bi obezbedio da se proizvođač pridržava zahteva vezanih za dizajn, razvoj, proizvodnju, instalaciju i održavanje. U tabeli 12.12 navedene su klauzule ISO 9001. Kako je ISO 9001 sasvim opšti, postoji poseban dokument, ISO 9000-3, u kome su date smernice za tumačenje ISO 9001 u kontekstu softvera (Intenational Standards Organization 1990).

U klauzuli 4.2 ISO 9001 zahteva se da organizacija ima dokumentovani sistem kvaliteta, uključujući priručnik, planove, procedure i instrukcije za kvalitet. ISO 9000-3 tumači tu klauzulu za softver objašnjavajući kako bi sistem kvaliteta trebalo da se integriše u proces razvoja softvera. Na primer, u klauzuli 4.2.3 se govori o planiranju kvaliteta u projektu, a 5.5 se njime bavi unutar datog razvojnog projekta.

Slično tome, u klauzuli 4.4 ISO 9001 se zahteva ustanovljavanje procedura za kontrolu i verifikaciju dizajna, uključujući

- aktivnosti planiranja, projektovanja i razvoja;
- definisanje organizacionih i tehničkih interfejsa;
- identifikovanje ulaza i izlaza;
- pregledanje, proveravanje i validacija dizajna
- kontrolisanje promena u dizajnu.

TABELA 12.12 Klauzule ISO 9001

Broj klauzule	Predmet
4.1	Odgovornost rukovodstva
4.2	Sistem kvaliteta
4.3	Pregled ugovora
4.4	Kontrola dizajna
4.5	Kontrola dokumenata i podataka
4.6	Naručivanje
4.7	Kontrola proizvoda isporučenih kupcu
4.8	Identifikacija proizvoda i mogućnost praćenja
4.9	Kontrola procesa
4.10	Inspekcija i testiranje
4.11	Kontrola opreme za inspekciju, merenja i testiranje
4.12	Inspekcija i provera statusa
4.13	Kontrola neodgovarajućeg proizvoda
4.14	Korektivna i preventivna akcija
4.15	Rukovanje, skladištenje, pakovanje, očuvanje i isporuka
4.16	Kontrola zapisa o kvalitetu
4.17	Unutrašnja kontrola kvaliteta
4.18	Obuka
4.19	Servisiranje
4.20	Statističke tehnike

Zatim, ISO 9000-3 preslikava te aktivnosti u kontekst softvera. Klauzula 5.3 bavi se specifikacijom zahteva kupca, 5.4 razmatra planiranje razvoja, 5.5 planiranje kvaliteta, 5.6 dizajn i implementaciju, 5.7 ispitivanje i validaciju, a 6.1 upravljanje konfiguracijom.

Standardi ISO 9000 koriste se za regulisanje unutrašnjeg kvaliteta i obezbeđenje kvalitetnih isporučilaca. Obično izvođač angažuje podizvođače za delove sistema, na osnovu dobavljačevog sertifikata ISO 9000. Proces sertifikacije ima definisan opseg i izvode ga revizori sistema kvaliteta. U Velikoj Britaniji, sertifikacija se izvodi pod okriljem programa TickIT; postoji iscrpan vodič TickIT za tumačenje i razradu koncepata i primene ISO 9000 (*Department of Trade and Industry* 1992).

Merenje je takođe deo standarda ISO 9000, ali ne tako eksplicitno kao u SPICE-u ili CMM-u. Posebno, u standardu ISO 9000 nema jakog naglaska na statističkoj kontroli procesa prisutnoj u SPICE-u i CMM-u. Međutim, kao i kod svih drugih okvira, ciljevi mogu lako da se pretoče u pitanja i metriku.

12.6 PROCENJIVANJE RESURSA

Mnogi istraživači veruju da je kvalitet resursa daleko važniji činilac kvaliteta proizvoda od bilo kog tehnološkog prodora koji možemo da napravimo. Na primer, DeMarco i Lister (1987) raspravljaju o dokazu da su potrebni kreativnost, neprekidno vreme i dobra komunikacija; oni tvrde da kohezivni timovi grade dobre proizvode.

Slično tome, Boehmovi modeli COCOMO (1981, 1995) barataju parametrima koji omogućuju da se potreban rad i rokovi predvide na osnovu atributa osoblja, kao što je iskustvo. Njegovo prvobitno istraživanje je otkrilo da su razlike između timova sa visokim i niskim performansama najviše uticale na produktivnost rada na projektu. Dakle, istraživači tvrde da bi trebalo više da se usredsredimo na ljude, a manje na tehnologiju.

S druge strane, softver se obično izrađuje u poslovnom okruženju. Resursi, kao što su vreme i novac su date veličine i od nas se traži da koristeći ih rešimo poslovni ili socijalni problem. Moramo da budemo u stanju da ocenimo da li svaki od datih resursa koristimo na odgovarajućem nivou. U ovom odeljku, ispitujemo dva modela za procenu tih vrsta resursa: model zrelosti ljudi za osoblje i model povraćaja ulaganja za vreme i novac.

Model zrelosti ljudi

Značajno je da se CMM ne bavi pitanjima koja se odnose na ljude i njihovu produktivnost. Iako se naziva modelom „sposobnosti", CMM je u stvari projektovan da meri sposobnost procesa, a ne sposobnost ljudi koji čine organizacije. Curtis, Hefley i Miller (1995) su pokušali da isprave taj propust predlažući **model zrelosti sposobnosti ljudi** za poboljšanje znanja i stručnosti radne snage.

Kao i CMM, model zrelosti ljudi ima pet nivoa, gde je nivo 5 najpoželjniji. Svaki nivo je povezan sa ključnim postupcima koji odražavaju promene i poboljšanja u organizacionoj kulturi. Tabela 12.13 predstavlja pregled tih nivoa i postupaka

TABELA 12.13 Model zrelosti sposobnosti ljudi (Curtis, Hefley i Miller 1995)

Nivo	U fokusu je	Ključni postupci
5: Poboljšavajući	Stalno poboljšanje znanja i stručnosti	Stalno investiranje u radnu snagu Obuka Razvoj lične kompetencije
4: Upravljan	Merljiva delotvornost, razvijeni timovi koji daju visoke performanse	Podešavanje organizacione performanse Upravljanje organizacionom kompetentnošću Rad zasnovan na timovima Izgradnja timova Mentorstvo
3: Definisan	Radni zadaci raspoređeni na osnovu kompetencije	Kultura zajedničkog rada Rad zasnovan na kompetenciji Razvoj karijere Razvoj kompetentnosti Planiranje radne snage Analiza znanja i stručnosti
2: Ponovljiv	Rukovodstvo preuzima odgovornost za upravljanje svojim ljudima	Kompenzacija Obuka Upravljanje performansom Popuna osobljem Komunikacija Radno okruženje
1: Početni		

Najniži nivo predstavlja polaznu tačku, sa dosta prostora za poboljšanje. Na **početnom** nivou, organizacija ne uzima aktivnu ulogu u razvoju ljudi koji rade za nju. Sposobnost upravljanja se zasniva na iskustvu iz prošlosti i ličnoj komunikacionoj veštini, a ne na formalnoj obučenosti za upravljanje. Na ovom nivou se izvode neke aktivnosti u vezi s ljudima, ali se ne stavljaju u širi kontekst motivacije i dugoročnih ciljeva.

U nezreloj organizaciji kao što je ona na nivou 1, mnogi menadžeri ne smatraju talenat osoblja važnim resursom. Projektanti slede sopstvene ciljeve i nisu dovoljno motivisani da ih usklade sa poslovnim ciljevima kompanije. Znanje i veštine stagniraju, zato što se radnici premeštaju sa posla na posao i nema sistematskog plana za unapređivanje njihovog znanja.

Nivo 2 predstavlja prvi korak ka usavršavanju radne snage. Menadžeri prihvataju unapređivanje i usavršavanje radnika kao ključnu odgovornost, ali samo ako shvate da je performansa organizacije ograničena znanjima ljudi koji je čine. Dakle, usredsređenje na **ponovljivom** nivou je da se ustanove osnovni principi rada između različitih zanimanja u datoj jedinici ili organizaciji.

Među najjednostavnijim su oni postupci koji podržavaju neometani rad u okruženju. Preduzimaju se koraci da se poboljša komunikacija i koordinacija, a menadžeri zapošljavanje i izbor radnika shvataju veoma ozbiljno. Menadžeri se staraju da razgovaraju o poslovnom učinku s radnicima i nagrađuju ih kada je učinak izuzetan. Cilj obuke je da dopuni postojeće znanje, a kompenzacija se daje da bi se postigla pravičnost, motivisanost i zadržavanje ljudi.

Na nivou 3, organizacija počinje da kroji radne postupke prema domenu svog poslovanja. Ona započinje ovaj **definisani** nivo zrelosti stvaranjem strateškog plana za lociranje i razvoj talenata koji su joj potrebni. Potrebe se utvrđuju u skladu sa znanjima i veštinama potrebnim za poslovanje, za koja se smatra da predstavljaju jezgro kompetentnosti jedne kompanije. Radnici se nagrađuje zbog savladavanja neophodnih znanja i razvijanja svoje stručnosti. Nastupanje tih promena treba da postakne radnike da učestvuju u ostvarivanju poslovnih ciljeva kompanije.

Mentorstvo igra glavnu ulogu na nivou 4, **upravljanom** nivou zrelosti. Ne podstiču se samo pojedinci da uče glavne veštine, nego se i timovi fomiraju oko znanja i veština koje se međusobno dopunjavaju. Aktivnosti izgradnje timova dovode do timskog duha i kohezije, i mnogi organizacioni postupci se usredsređuju na motivisanje i razvoj timova.

Na tom nivou, organizacija postavlja kvantitativne ciljeve za rast centralne kompetencije, a performansa se motiviše preko pojedinaca, timova i organizacija. Ispituju se trendovi da bi se odredilo kako se u praksi povećavaju kritična znanja. Zbog tog kvantitativnog razumevanja, mogućnosti osoblja su predvidljive, što značajno olakšava upravljanje.

Optimizujući nivo je peti i najviši nivo zrelosti. Tu se pojedinci, menadžeri i cela organizacija usredsređuju na poboljšanje timskih i individualnih znanja. Organizacija može da identifikuje prilike za jačanje znanja osoblja i to činii ne čekajući da se javi problem ili nazadovanje koji bi je na to naterali. Analiziraju se podaci da bi se odredila moguća poboljšanja performansi uvođenjem izmena u postojeće postupke ili isprobavanjem novih, inovativnih tehnika. Novi postupci koji nude najbolje rezultate implementiraju se širom organizacije. Uopšte, optimizujuća kultura je u tome da se svaki član osoblja usredsredi na svaki aspekt poboljšanja: pojedinca, tima, projekta, organizacije i kompanije.

Curtis, Hefley i Miller (1995) ističu da model zrelosti ljudi:

- razvija sposobnosti;
- gradi timove i kulturu;
- motiviše i upravlja performansom;
- uobličava radnu snagu.

Ocenjivačke smernice korisne su ne samo za ocenu date organizacije, nego i za planiranje programa poboljšanja.

Povraćaj ulaganja

U nastojanjima da unapredimo razvoj softvera, biramo između preporučenih metoda i alata. Ograničeni resursi obično sužavaju naš izbor; ne možemo sve da radimo, pa tražimo kriterijume za izbor jedne ili više stvari koje najverovatnije mogu da nam pomognu u postizanju našeg cilja. Dakle, moramo dobro da pogledamo kako projektanti i menadžeri softverskog inženjerstva donose odluke o ulaganju u tehnologiju.

Cenjene poslovne publikacije se često bave problemom ulaganja u tehnologiju. Na primer, u nedavno štampanom članku govori se da svaka procena postojećeg ili predloženog ulaganja u tehnologiju treba da daje izveštaje na više načina odjednom, da bi formirala „uravnotežen rezultat": u pogledu kupca (odnosno, njegovog zadovoljstva), operacionalnog pogleda (odnosno, kompetencija), finansija (odnosno, povraćaja ulaganja, deljenja cene) i u pogledu poboljšanja (odnosno, liderstva na tržištu i dodate vrednosti) (Kaplan i Norton 1992). Na primer, u podsetniku 12.8 opisuju se neki od povraćaja ulaganja u banci Chase Manhattan. Favaro i Pfleeger (1997) izveštavaju da ekonomska vrednost može da bude objedinjavajući princip. To znači da možemo da gledamo na svaku od alternativa ulaganja kao na moguću ekonomsku vrednost za kompaniju u celini. U stvari, maksimizovanje ekonomske vrednosti može sasvim dobro da dovede do povećanja kvaliteta, zadovoljstva kupaca i liderstva na tržištu.

Međutim, postoji više različitih načina da se dođe do ekonomske vrednosti. Moramo da odlučimo koji pristup analize ulaganja u vezi sa softverom najviše odgovara za donošenje odluka, zasnovano na ekonomskoj vrednosti. Analiza ulaganja bavi se samo najboljim načinom dodeljivanja kapitala i ljudskih resursa. Dakle, ona je uočljivo različita od procene troškova ili metrike; ona odmerava više alternativa, uključujući mogućnost korišćenja tržišta kapitala da obezbedi očekivan godišnji nivo povraćaja. Drugim rečima, iz korporativne perspektive visokog nivoa, rukovodstvo mora da odluči kako se predložena tehnologija poredi sa jednostavnim puštanjem da novac zarađuje kamatu ležeći u banci!

Ne odražavaju sve analize ulaganja tu realnost. Posmatrajući iz ugla finansijskog analitičara, Favaro i Pfleeger (1997) kritički sagledavaju većinu obično korišćenih pristupa: neto sadašnju vrednost, isplatu, prosečan povraćaj po knjigovodstvenoj vrednosti, unutrašnju brzinu povraćaja i indeks profitabilnosti. Oni pokazuju da neto sadašnja vrednost (NSV) ima najviše smisla za procenu ulaganja u vezi sa softverom.

NSV izražava ekonomsku vrednost izraženu u ukupnom životu projekta, bez obzira na razmeru ili vremenski okvir. Kako planiranje ulaganja obuhvata trošenje novca u budućnosti, možemo da razmišljamo o **sadašnjoj vrednosti** ulaganja kao o današnjoj vrednosti predviđenog budućeg toka gotovine. Proračun NSV koristi **procenat popusta** ili **oportunu cenu**, što odgovara očekivanoj brzini povraćaja podjednakog ulaganja na tržištima kapitala; ta brzina može da se menja u vremenu. Drugim rečima, procent po-

pusta odražava koliko novca kompanija može da zaradi ako uloži svoj novac u banku ili finansijsku ustanovu, umesto u softversku tehnologiju. U kompaniji Hewlett-Packard upotrebili su NSV da bi procenili ulaganje u dva višegodišnja korporativna višekratno upotrebljiva projekta.

PODSETNIK 12.8 POVRAĆAJ ULAGANJA KOD CHASE MANHATTANA

U poglavlju 11. naučili smo o Chase Manhattanovom RMS-u (*Relationship Management System*), koji je združio nekoliko nasleđenih sistema u jedan, da bi obezbedio informacije o kupcima za servisne predstavnike. Novi sistem omogućava predstavnicima da provode manje vremena tragajući za podacima, a više da bi saznali koje su potrebe kupaca.

RMS je razvijen novim pristupom. Projektanti su ohrabrivani da razgovaraju, međusobno i sa kupcima, a ta povišena komunikacija je dovela do mnogo boljeg razumevanja onoga što je bilo potrebno – u nekim slučajevima bilo je potrebno manje nego što su projektanti mislili! Izgrađeno je pet različitih prototipova, a podaci su organizovani tako da se maksimizuje integritet.

Jedna od najvećih isplativosti ulaganja Chase Manhattana u tehnologiju bila je u povećanoj koheziji tima. „Razvojni tim RMS se držao demokratskog pristupa rešavanju problema. O prioritetima se glasalo i članovi tima su morali da se povinuju većini. Taj pristup je često rezultovao kompromisom, ali je takođe razvio ukrštenu funkcionalnu saradnju i poverenje." (Field 1997b).

Projekat je počeo 1994. godine, a do kraja 1996. godine RMS je instaliran u 700 od Chase Manhattanovih 1000 predstavničkih lokacija srednjeg tržišta. Ali čak i sa potpunim razmeštajem, RMS je povećao pozive kupaca za 13% i poboljšao profitabilnost za 27%. Štiteći svoja stara ulaganja i ohrabrujući komunikaciju između zaposlenih, Chase Manhattan je postigao četiri stvari.

1. Izbegao je ogromna ulaganja u novi hardver.
2. Obezbedio je svojim predstavnicima više podataka za kraće vreme.
3. Postigao je zadivljujući povraćaj ulaganja.
4. Stvorio je kohezivne timove koji bolje razumeju poslovanje Chase Manhattana.

Denis O'Leary, izvršni potpredsednik i CIO, zapazio je da je „stvarni izazov da se dobiju poslovne i IS grupe, ujedinjene oko partnerskog odnosa koji će opstati i tehničke infrastrukture koja je jaka". (Field 1997b).

Neto sadašnja vrednost je sadašnja vrednost dobitka umanjena za vrednost početnog ulaganja. Na primer, da bi uložila u novi alat, kompanija mora da potroši novac za vreme obuke i učenja, kao i za sam alat. Proračun NSV oduzima te troškove početnog ulaganja od projektovanog dobitka.

TABELA 12.14 Proračun neto sadašnje vrednosti za dve alternative

Tok novca	COTS	Ponovno korišćenje
Početno ulaganje	-9000	-4000
Godina 1	5000	-2000
Godina 2	6000	2000
Godina 3	7000	4500
Godina 4	-4000	6000
Suma svih tokova novca	5000	6500
NSV za 15%	2200	2162

Pravilo prihvatljivosti za NSV je jednostavno: uložiti u projekat ako je njegova NSV veća od nule. Da biste videli kako radi NSV, razmotrite sledeću situaciju. Kompanija može da napravi novu proizvodnu liniju na dva načina:

1. Zasnovati je na gotovom komercijalno raspoloživom softveru (*commercial off-the-shelf*, COTS). Taj izbor obuhvata velike početne troškove nabavke, sa kasnijim velikim povraćajima (na osnovu izbegnutog rada), ali će proizvod COTS zastareti i moraće da se zameni posle tri godine.
2. Izgraditi proizvod pomoću ponovnog korišćenja dizajna. Ponovno korišćenje zahteva značajne troškove za konstrukciju i dokumentaciju, ali su dugoročni troškovi manji nego što je uobičajeno.

Proračun neto sadašnje vrednosti može da izgleda kao u tabeli 12.14. Alternativa COTS ima neznatno veću NSV i zato predstavlja bolji izbor.

Pristup NSV je osetljiv na vremenski raspored tokova novca; što su povraćaji kasniji, to se više smanjuje opšta vrednost. Dakle, vreme za pojavu na tržištu je od suštinskog značaja za analizu i utiče na ishod. Veličina ili razmera projekta takođe se odražava na NSV. S obzirom na to da je NSV aditivna, možemo da procenimo efekte više projekata jednostavnim zbrajanjem njihovih pojedinačnih NSV-ova. S druge strane, značajni dobici iz jedne tehnologije mogu da maskiraju gubitke od ulaganja u drugu; iz tog razloga korisno je proceniti svaku vrstu ulaganja posebno. U praksi stvarnog okruženja, NSV se ne koristi za naivne procene samo jednog projekta, nego u kontekstu svestranijeg finansijskog i strateškog okvira (Favaro 1996).

12.7 PRIMER IZ INFORMACIONIH SISTEMA

Sistem Piccadilly sasvim jasno dodaje vrednost televizijskim oglašivačima koji su ga angažovali. Vreme oglašavanja može brže da se prodaje, cene i šeme mogu lako i brzo da se menjaju, a specijalne ponude mogu da se skroje tako da reaguju na konkurenciju. Ali, kako da ugradimo tu dodatu vrednost? Ako se prihodi povećavaju, onda možemo da ih poredimo sa novcem koji se ulaže u razvoj sistema. Međutim, možemo da se nađemo u situaciji gde prihodi ostaju isti, ali bi opali da nema takvog sistema. Odnosno, ponekad moramo da ulažemo u tehnologiju da bismo održali svoje mesto na tržištu i preživeli, a ne da poboljšavamo svoj položaj.

Tim pitanjima bi trebalo da se bavimo u sklopu post mortem analize, pored tehničkih pitanja opisanih u ovom poglavlju. Drugim rečima, post mortem analiza mora da razmotri poslovanje kao i tehnologiju, povezujući ih međusobno kada je to potrebno da bi se odgovorilo na pitanje: „Da li je ovaj sistem dobar za poslovanje?" Odgovor može da bude težak, a sigurno ne može lako da se kvantifikuje. Ponekad se usvajaju nove tehnologije ne zato što su najbolje za posao, nego zato što će dobri zaposleni otići ako se ne obuče za rad sa najnovijim tehnikama i alatima. Kao menadžeri, moramo da imamo na umu da se projektanti i serviseri motivišu ne samo zaradama. Oni takođe vole stalni izazov, priznanje od onih koji su im ravni i priliku da ovladaju novim znanjima. Zato povraćaj ulaganja obuhvata ne samo novčanu nagradu i zadovoljstvo korisnika, nego i zadovoljstvo zaposlenih. Ulaganje u zaposlene i timove može takođe da bude dobro za poslovanje.

12.8 PRIMER IZ PRAKSE

Izveštaj Ariane-5 je odličan primer post mortem analize. Istražni tim je sledio proces sličan onom koga preporučuju Collier, DeMarco i Fearey (1996) i usredsredio se na očiglednu potrebu da odredi šta je prouzrokovalo neispravnost zbog koje je raketa eksplodirala. U izveštaju su izbegnute optužbe i žalbe; umesto toga, u njemu su navedeni koraci koji su mogli da se preduzmu u toku razvoja, a koji bi ukazali na početni problem: pregledi zahteva, strategije projektovanja, tehnike ispitivanja, simulacija i ostalo.

Sledeći korak nije dokumentovan u izveštaju: da se, korišćenjem preporuka iz izveštaja, promeni način na koji se sledeća raketa konstruiše, gradi i ispituje. Kao što ćemo videti u poglavlju 13, možemo da uporedimo podatke iz post mortem analize Ariane-5 sa onima od sledećih raketa, da bismo odredili da li je postignuto ikakvo poboljšanje. Poboljšavanje je trajan proces, pa verovatno treba da napravimo istorijat iz niza post mortem analiza; kada rešimo jedan problem, okrećemo se sledećem najkritičnijem problemu, sve dok uspešno ne odgovorimo na glavne izazove.

12.9 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju smo potražili načine da procenimo proizvode, procese i resurse. Počeli smo pregledom više pristupa proceni, uključujući analizu svojstava, ankete, studije slučajeva i formalne eksperimente. Videli smo da je merenje od suštinskog značaja za svaku procenu i da je važno razumeti razliku između procene i predviđanja. Zatim, razmotrili smo kako da vrednujemo mere; odnosno, želimo da budemo sigurni da merimo ono što mislimo da merimo i da su naša predviđanja tačna.

Procena proizvoda se obično zasniva na modelu atributa od interesa. Pregledali smo tri modela kvaliteta da bismo videli kako je svaki od njih rešavao pojedinačna pitanja o tome kako se različiti delići kombinuju da bi se oblikovala celina. Zatim smo razmotrili ponovno korišćenje softvera, ističući pitanja koja se pojavljuju kada moramo da procenimo komponentu kao kandidata za ponovno korišćenje.

Procena procesa može se izvršiti na više načina. Post mortem analiza gleda unazad na završen proces, da bi procenila osnovne uzroke stvari koje su išle loše. Modeli procesa, kao što su model zrelosti sposobnosti, SPICE i ISO 9000, korisni su za postavljanje pitanja o kontroli i povratnoj sprezi koje imamo nad korišćenim procesom.

CMM je inspirisao mnoštvo drugih modela zrelosti, uključujući model zrelosti ljudi, za procenu stepena do koga su pojedincima i timovima dati resursi i slobode koje su im potrebne da bi dali sve od sebe. U naše projekte ulažemo i druge resurse, uključujući novac i vreme. Strategije povraćaja ulaganja nam pomažu da shvatimo da li poslovanje ima koristi od ulaganja u ljude, alate i tehnologiju.

12.10 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Mnogi od modela za procenu o kojima smo govorili u ovom poglavlju usredsređuju se na interakciju tima. Modeli zrelosti procesa nadgledaju koordinaciju i komunikaciju tima, ohrabrujući merljivu povratnu spregu od jedne do druge aktivnosti procesa. Ti modeli pomažu timovima da kontrolišu ono što rade i da prave bolja predviđanja o tome šta će se dogoditi u budućnosti. Slično tome, modeli kao što je model zrelosti ljudi procenjuju da li su pojedinci i timovi nagrađeni i motivisani da daju sve od sebe.

Analiza svojstava, studije slučajeva, ankete i eksperimenti ohrabruju timove da dele informacije, u nadi da će razumeti i verifikovati odnose između proizvoda, procesa i resursa. Timovi moraju da rade zajedno, za vreme formalnih istraga kao i post mortem analiza, ostavljajući po strani pojedinačne sklonosti, da bi odredili osnovne uzroke glavnih problema koji se mogu otkloniti u budućnosti.

Najzad, videli smo kako povraćaj ulaganja obuhvata i ulaganje u ljude, kao i u tehnologiju. Obučeni i motivisani timovi će verovatno biti produktivniji, prenoseći svoje iskustvo i razumevanje sa jednog projekta na drugi.

12.11 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Postoji velika potreba za više empirijskog procenjivanja inženjerskih praksi i proizvoda. Istraživači moraju da prilagode standardne istražne tehnike realnostima softverskog inženjerstva. Ne možemo da radimo isti projekat dva puta, jednom sa tehnikom ili alatom, a drugi put bez njih. Zato moramo da učimo od naših kolega iz društvenih nauka i prilagodimo naše metode istraživanja, dok saznajemo što više možemo kako da budemo efektivniji.

Modeli i okviri nam pomažu da razumemo odnose koje istražujemo. Istraživači nastavljaju da nam predlažu nove načine kako da sagledavamo različite aspekte naših proizvoda, procesa i resursa; onda možemo da procenjujemo i same modele i okvire, da bismo videli kako se oni podudaraju sa onim što već znamo.

12.12 SEMINARSKI RAD

Sad kad ste završili svoj projekat Organizator pozajmica, razmotrite kako biste procenili softver i proces razvoja upotrebljene da biste ga izgradili. Koliko je softver dobar? Kako biste poredili kvalitet softvera Organizatora pozajmica sa onim od nekog drugog projekta koji ste završili?

Slično tome, ispitajte proces koji ste koristili da proizvedete Organizator pozajmica. Da li biste ga smatrali Početnim? Definisanim? Ponovljivim? Upravljanim? Optimizujućim? Koje prakse su pomogle da projekat bude uspešan? Da ne uspe?

12.13 KLJUČNE REFERENCE

Tehnike procene su sasvim složene i o njima se može govoriti više nego što je to učinjeno u ovom poglavlju. Fenton i Pfleeger (1997) imaju tri poglavlja o proceni; jedno o tehnikama, drugo o prikupljanju podataka i treće o analizi podataka. Pfleeger i Kitchenham imaju niz članaka u *ACM Softvware Engioneering Notes*, koji započinje u decembru 1995. godine, detaljno raspravljajući o različitim istražnim tehnikama. Pfleeger (1996b) se bavi potrebom da se tehnologija razvija kada proučavamo njenu efektivnost.

Software Engineering Institute dozvoljava korisnicima da preuzimaju njegove dokumente modela zrelosti sa lokacije ftp://ftp.sei.cmu.edu/pub/documents.

Informacije o modelu SPICE mogu da se nađu na lokaciji http://www.sqi.cit.gu.edu.au/spice.

Više izdanja *IEEE Software* bilo je posvećeno ponovnom korišćenju, uključujući ona iz maja i septembra 1994. godine. Ovo poslednje sadrži listu na čitavoj stranici ključnih članaka i knjiga o ponovnom korišćenju u 1994. godini.

ReNewa, elektronski časopis o ponovnom korišćenju, raspoloživ je preko anonimnog ftp na ftp.vt.edu u pubs/reuse.

Veb lokacija Wildfire sadrži dokumente i uzorke post mortem analize proizvoda uključujući sažet, definisan proces, uzorak pregleda, uzorak tabeliranja rezultata, uzorke dijagrama sličnosti i alat za predviđanje rasporeda. Lokacija je http://www.wildfire.com/research/postmortems.html.

Postoji više konferencija i radionica posvećenih ponovnom korišćenju. One glavne, pod pokroviteljstvom IEEE Technical Council od Software Engineering, su International Conference on Software Reuse i Annual Workshop on Software Reuse. Informacije o prethodnim izdanjima njihovih zbornika i najavama sledećih konferencija su raspoložive na veb lokaciji IEEE Computer Society.

Postoji takođe više konferencija i organizacija u vezi sa procenom. International Symposium on Software Metrics, pod pokroviteljstvom IEEE Computer Society, bavi se pitanjima merenja i empirijskih istraživanja. Informacije o najnovijoj konferenciji takođe se nalaze na veb lokaciji Computer Society. International Software Engineering Research Network objavljuje studije ponovljenih istraživanja i mnoge od njih su raspoložive na njihovoj veb lokaciji. Najzad, časopis *Empirical Software Engineering* objavljuje ne samo studije, nego i podatke i smernice za empirijsko istraživanje.

12.14 VEŽBE

1. Delići facetovane šeme klasifikacije moraju da budu ortogonalni. To znači, karakteristike koje opisuje jedan facet ne mogu da se opišu koristeći kombinaciju jednog ili više drugih faceta. Definišite skup faceta za klasifikovanje knjiga u biblioteci softverskog inženjerstva. Koliko faceta vam je potrebno? Kako znate kada ste definisali dovoljno faceta? Da li je opis svake knjige jedinstven?
2. Objasnite zašto model troškova ponovnog korišćenja softvera mora da uključi troškove za više od jednog projekta.
3. Napravite spisak nekih informacija koje mogu da budu korisne u zapisivanju istorije ponovnog korišćenja komponente. Obavezno uključite objašnjenje osnovnih razloga za svaki element u vašem spisku.

4. Pretpostavite da post mortem analiza otkriva da je određeni projektant odgovoran za glavne probleme u sistemu. Koje vrste aktivnosti poboljšanja bi trebalo da se uključe u preporuku koja se time bavi?
5. Izvršite post mortem analizu na jednom od sopstvenih projekata. Šta biste uradili drukčije kada biste ponovo radili projekat? Kako znate da bi te lekcije poboljšale vaš sledeći projekat?
6. Ispitajte modele kvaliteta opisane u ovom poglavlju: Boehm, ISO 9126 i Dromey. Za svaku karakteristiku koja doprinosi kvalitetu, raspravljajte o mogućim načinima merenja karakteristike i opišite da li je mera subjektivna ili objektivna.
7. Ispitajte modele kvaliteta na slikama 1.5, 12.2 i 12.3. Kako takvi modeli mogu da se upotrebe da bi se sprečili problemi u vezi sa kvalitetom proizvoda? Da li merenje pomaže da se izbegnu takvi problemi?
8. Uporedite i suprotstavite modele kvaliteta McCall, Boehm i ISO 9126. U čemu se oni razlikuju sa tačke gledišta projektanta? A sa tačke gledišta korisnika?
9. Za ISO 9126 se smatra da je opšti model kvaliteta softvera koga može da koristi svako ko se bavi softverom. Ima li smisla posedovati opšti model? Kako on pomaže u poređenju kvaliteta dva različita proizvoda? Da li su neki proizvodi tako neuobičajeni da ISO 9126 ne može da se primeni?
10. Računarska bezbednost se obično smatra potrebnom za visoko kvalitetan softverski proizvod. Kako računarska bezbednost može da se definiše preko termina modela kvaliteta ISO 9126?
11. Pretpostavite da ste implementirali novu tehniku anketiranja za vreme svog procesa formulisanja zahteva. Kako biste mogli da procenite njenu efektivnost? Kako biste kontrolisali promenljive, tako da budete sigurni da je nova tehnika ta koja je odgovorna za razlike u kvalitetu ili produktivnosti?
12. Model zrelosti sposobnosti koriste mnoge kompanije kao podsticaj za implementiranje novih praksi. To znači da organizacije postavljaju ciljeve i nagrađuju ponašanje da bi pomogle njihovo kretanje od nivoa 1 ka nivou 5. Koje vrste merljivih ciljeva mogu da se postave za svaku od ključnih oblasti procesa? Kako te mere mogu da se upotrebe da bi se pratilo napredovanje ka nivou 5?
13. Model zrelosti ljudi pretpostavlja da kohezivni timovi prave bolje proizvode. Opišite kako biste mogli da ispitujete tu hipotezu u formalnoj proceni. Kako biste merili koheziju tima? Koje kriterijume biste koristili da odredite kada je proizvod „bolji"?

13 Poboljšanje predviđanja, proizvoda, procesa i resursa



U ovom poglavlju obradivaćemo:

- poboljšanje predviđanja;
- poboljšanje proizvoda višekratnom upotrebljivošću i inspekcijama;
- poboljšanje procesa korišćenjem čiste sobe i modela zrelosti;
- poboljšanje resursa istraživanjem kompromisa.

Ispitali smo više tehnika i alata softverskog inženjerstva, od kojih je svaka namenjena da nam pomogne da bolje proizvode izgradimo na bolji način. U poglavlju 12, naučili smo kako da procenimo proizvode, procese i resurse da bismo odredili koji aspekti razvoja i održavanja utiču na kvalitet proizvoda koje gradimo i održavamo. Videli smo da nam više vrsta studija, podržanih empirijskim dokazima, omogućava da uspostavimo osnovu sa koje merimo promenu i koje nam pomažu da poredimo i suprotstavljamo efekte različitih tehnika i alata. U ovom poglavlju razmatramo kako da kombinujemo pažljivu procenu sa prihvatanjem tehnologije da bi nam to pomoglo da poboljšamo način na koji koristimo novu tehnologiju. Na primer, nije dovoljno prihvatiti inspekcije i preglede samo zato što nam izgledaju kao dobre ideje. Bolje je ispitati način upotrebe inspekcija i pregleda da bismo razumeli šta je to što ih čini dobrim i kako da ih učinimo još efektivnijim.

Usredsređujemo se na četiri aspekta tehnologije softverskog inženjerstva: predviđanje, proizvode, procese i resurse. Za svaki od njih, govorimo o više strategija poboljšanja, zasnovanih na savremenoj empirijskoj studiji. Primeri koji se ovde predstavljaju izabrani su da prikažu tehnike procene i poboljšanja koje se koriste; oni nisu zamišljeni da posluže kao potvrda određenih tehnika ili strategija. Kao i uvek, predlažemo da koristite i procenjujete tehnike i alate koji vam izgledaju najpogodniji za vaša sopstvena okruženja razvoja i održavanja.

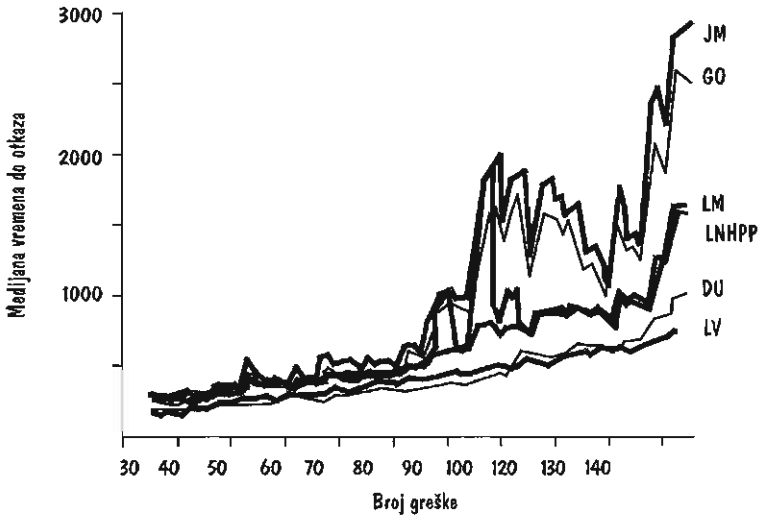
13.1 POBOLJŠAVANJE PREDVIĐANJA

U celoj ovoj knjizi sagledali smo potrebu da predviđamo razne stvari: rad i rokovi na predloženom projektu, broj grešaka u softveru, pouzdanost novog sistema, vreme potrebno za testiranje novog proizvoda i još mnogo toga. U svakom slučaju, želimo da naša predviđanja budu tačna. To znači, želimo da predviđena vrednost bude bliska stvarnoj vrednosti. U ovom poglavlju, tražimo načine da poboljšamo proces predviđanja, tako da on daje tačnije procene. Usredsredićemo se na modele pouzdanosti, ali su razmatrane tehnike podjednako primenljive i na druge vrste predviđanja.

Predvidljiva tačnost

U poglavlju 9 ispitali smo više modela koji se koriste za predviđanje verovatne pouzdanosti sistema, uz primedbu da moramo da poredimo tačnost predviđanja sa stvarnim vrednostima pouzdanosti. U ovom poglavlju, sagledaćemo bliže tačnost modela pouzdanosti i ispitaćemo neke tehnike za poboljšavanje te tačnosti.

Abdel-Ghaly, Chan i Littlewood (1986) poredili su više modela pouzdanosti na istom skupu podataka (skup Musa koji se koristi u poglavlju 9) i otkrili dramatične razlike između njihovih predviđanja. Na slici, „JM" označava model Jelinski-Moranda, predstavljen u poglavlju 9; „GO" je model Gole-Okumoto, „LM" je Littlewoodov model, „LNHPP" je Littlewoodov model nehomogenog Possonovog procesa, „DU" je Duaneov model, a „LV" je model Littlewood-Verrall. Svaki model je upotrebljen da bi se generisalo 100 uzastopnih procena pouzdanosti. Iako svaki model pokazuje povećavajuću pouzdanost na razmatranom skupu podataka, postoje značajne razlike u ponašanju svakog predviđanja u toku vremena. Neki modeli su optimističniji od drugih, a neki poseduju značajan nivo „šuma", sa velikim fluktuacijama u predviđanju kada se ispitivanje nastavlja.



SLIKA 13.1 Rezultati primene modela pouzdanosti na Musa skup podataka

Nije neuobičajeno da modeli proizvode takve rezultate, što nam otežava da znamo kako da koristimo modele u našim posebnim projektima. Treba da znamo koji model najviše odgovara i uvek moramo da nađemo načine na koje ćemo odrediti kada su predviđanja najtačnija.

Predviđanja mogu da budu netačna na dva različita načina:

1. Predviđanja su **polarisana** kada su dosledno različita od stvarne pouzdanosti proizvoda.
2. Predviđanja sadrže **šum** kada uzastopna predviđanja mere (kao što je medijana vremena do otkaza ili srednje vreme do otkaza) fluktuiraju više od stvarne pouzdanosti.

Mi naravno ne znamo stvarnu pouzdanost, pa nam je teško da odredimo količinu šuma ili polarisanosti u predviđanju. Međutim, Abdel-Ghaly, Chan i Littlewood (1986) predlažu više tehnika za analizu tačnosti i pomažu nam da odredimo koji model da koristimo.

Bavljenje polarizacijom: u-kriva

Polarizacijom se bavimo tako što poredimo koliko često su posmatrana vremena otkaza manja od onih predviđenih. To znači, kada dati model predviđa da će se sledeći otkaz dogoditi u određeno vreme, merimo stvarno vreme sledećeg otkaza i poredimo te dve vrednosti. Pretpostavite da časovnik krene u 0 i da se prvi otkaz pojavi u vremenu *t1*. Vreme do sledećeg otkaza je *t2*, a mi nastavljamo da zapisujemo vremena između otkaza, sve dok ne budemo posmatrali *n* softverskih otkaza, sa vremenima između otkaza od *t1* do *tn*. Te vrednosti poredimo sa predviđanjima (iz modela) od *T1* do *Tn*. Zatim brojimo koliko puta je *ti* manje od *Ti*; ako je taj broj značajno manji od *n/2*, onda verovatno imamo polarizaciju u našim predviđanjima. Na primer, na slici 13.1, 66 od 100 posmatranja upotrebljenih sa modelom Jelinski-Moranda manje je od predviđene medijane vremena do sledećeg otkaza. Odnosno, predviđene medijane su suviše velike, pa kažemo da je model Jelinski-Moranda suviše optimističan na tom skupu podataka. Zaista, ako gledamo samo poslednjih 50 posmatranja, nalazimo da je 39 stvarnih vremena manje od predviđenih vremena. To znači da je povećanje pouzdanosti predviđeno pomoću modela Jelinski-Moranda veće od stvarnog povećanja. Na isti način možemo da analiziramo model Littlewood-Verral i vidimo da je on dosledno pesimističniji u svojim predviđanjima.

Mi intuitivno očekujemo nešto suprotno. Odnosno, očekujemo da imamo tačnija predviđanja kada se ispitivanje nastavlja, kada nalazimo više grešaka i kada znamo više o podacima.

Polarizovanost izražavamo na formalniji način uspostavljanjem niza brojeva {*ui*}, gde je svako *ui* procena verovatnoće da je stvarno posmatranje bilo manje od onoga što smo predvideli da bude. Na primer, razmotrite sistem za predviđanje, predstavljen u poglavlju 9, gde smo predvidali srednje vreme do sledećeg otkaza uzimajući srednju vrednost dva prethodno posmatrana vremena otkaza. Tu tehniku možemo da upotrebimo na podacima Musa da bismo generisali vrednosti u trećoj koloni tabele 13.1.

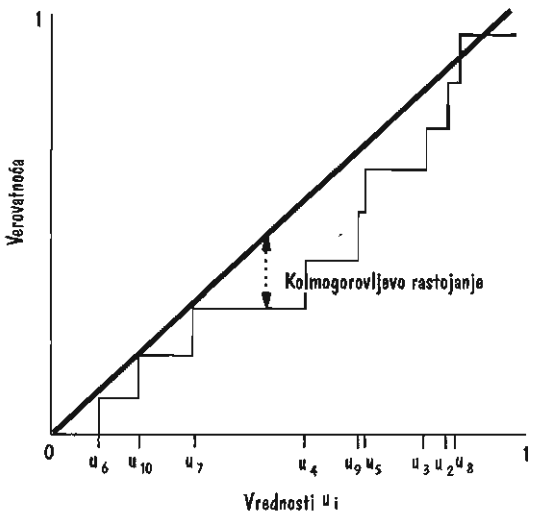
TABELA 13.1 Generisanje vrednosti ui za predviđanja zasnovana na podacima Musa

i	t_i	Predviđeno srednje vreme za i-ti otkaz	u_i
1	3		
2	30	16,5	0,84
3	113	71,5	0,79
4	81	97	0,57
5	115	98	0,69
6	9	62	0,14
7	2	5,5	0,30
8	91	46,5	0,86
9	112	101,5	0,67
10	15	63,5	0,21

Možemo da izračunamo funkciju raspodele za taj niz podataka (za detalje, pročitajte Fentona i Pfleegera [1997]), odakle proračunavamo vrednosti u. Zatim, konstruišemo krivu koja se zove u-**kriva**: stavljamo vrednosti ul duž horizontalne ose i zatim crtamo stepenastu funkciju, gde svaki stepenik ima visinu 1/(n + 1) (pretpostavljajući da ima n vrednosti ui). Na slici 13.2 prikazana je u-kriva za devet vrednosti u tabeli 13.1.

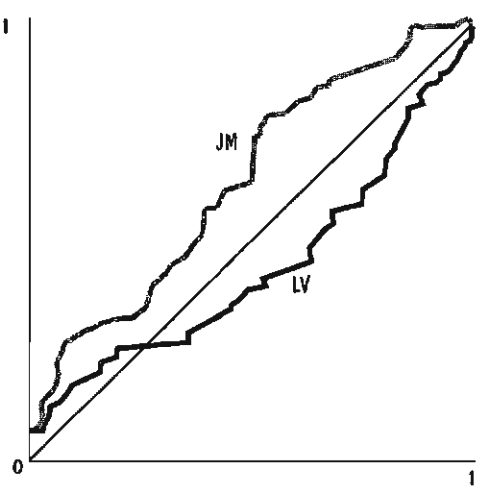
Ako nacrtamo liniju sa nagibom 1 (odnosno, liniju pod uglom od 450 u odnosu na horizontalnu i vertikalnu osu), možemo da je poredimo sa u-krivom. Kada bi naša predviđanja bila savršeno tačna, očekivali bismo da u-kriva bude ista kao ta linija. Dakle, svaka razlika između linije i u-krive predstavlja odstupanje između predviđanja i stvarnog posmatranja. Stepen odstupanja merimo pomoću **Kolmogorovljevog rastojanja** (maksimalnog vertikalnog rastojanja između linije i u-krive), kao što je prikazano na slici 13.2.

Da biste videli kako radi u-kriva, razmotrite dva najekstremnija modela na slici 13.1, modele Jelinski-Moranda i Littlewood-Verral. Možemo da merimo Kolmogorovljevo rastojanje (odnosno najveće vertikalno rastojanje) svakog od njihovih krivih od linije sa jediničnim nagibom, kao što je prikazano na slici 13.3. Razlika za model Jelinski-Moranda iznosi 0,190, što je značajno na nivou 1%; za model Littlewood-Verral, rastojanje je 0,144, značajno na nivou 5%. Dakle, ta dva modela nisu baš tačna na tom skupu podataka.



SLIKA 13.2 u-kriva za vrednosti iz tabele 13.1

РЕПУБЛИКА СРПСКА
Природно математички
факултет Бања Лука
ФИЗИКА



SLIKA 13.3 *u*-kriva Jelinski-Moranda (JM) i Littlewood-Verral (LV) za 100 previdanja po jedan korak unapred

Što je važnije, možemo da vidimo da je kriva Jelinski-Moranda uvek iznad linije, dakle ima mnogo malih vrednosti *u*. Drugim rečima, kada je *u*-kriva iznad linije, model je suviše optimističan. Slično tome, model Littlewood-Verral je suviše pesimističan, zato što je najviše ispod linije. Voleli bismo model čija su predviđanja mnogo bliža liniji od oba ta modela.

Bavljenje šumom: verovatnoća prethodne jednakosti

Eliminisanje polarizacije iz modela nije dovoljno. Ako imamo nepolarizovan, ali veoma zašumljen model, on još uvek nije mnogo koristan. Da biste videli zašto, razmotrite predviđanje napravljeno korišćenjem medijane tri prethodna vremena između otkaza, *ti-1*, *ti-2* i *ti-3*. Predviđamo da je medijana vremena do otkaza *Ti* medijana *ti-1*, *ti-2* i *ti-3*; kako su prva tri vremena između otkaza 3, 30 i 113, postavljamo *T4* da bude 30. Slično tome, *T5* je 113, a *T6* 81. Te procene su verovatno veoma daleko od stvarnih vrednosti i fluktuiraće u velikoj meri čak i kada stvarni brojevi budu imali mnogo mirnije ponašanje. Dakle, u tim predviđanjima imamo mnogo šuma.

Fluktuacije ponekad odražavaju način na koji fluktuira stvarna pouzdanost. Na primer, možemo da dobijemo značajnu fluktuaciju kada pravimo promene u sistemu i unosimo nove greške. **Neopravdan šum** se pojavljuje kada stvarna pouzdanost ne fluktuira, ali procene fluktuiraju. Nema tehnike osetljive na neopravdan šum, ali možemo da upotrebimo opštiju tehniku, *prethodne jednakosti* (*prequential likelihood*) da bismo se bavili šumom i polarizacijom zajedno, što nam pomaže da izaberemo dobar model.

Funkcija verovatnoće *prethodne jednakosti* (Dawid 1984) nam dozvoljava da poredimo više različitih predviđanja na istom izvoru podataka, tako da možemo da izaberemo najtačnije. (Opšti pregled verovatnoće *prethodne jednakosti* može da se pronađe u Fentonu i Pfleegeru [1997], a detaljan opis daje [Dawid 1984].) Koristeći skup podataka Musa i predviđanja generisana izračunavanjem srednje vrednosti dve prethodno posmatrane vrednosti, možemo da izračunamo niz funkcija verovatnoće *prethodne jednakosti* za svako posmatranje. Rezultat je prikazan u tabeli 13.2.

TABELA 13.2 Izračunavanja verovatnoće *prethodne jednakosti*

i	t_i	T_i	Verovatnoće *prethodne jednakosti*
3	113	16,5	6,43E-05
4	81	71,5	2,9E-07
5	115	97	9,13E-10
6	9	98	8,5E-12
7	2	62	1,33E-13
8	91	5,5	1,57E-21
9	112	46,5	3,04E-24
10	15	101,5	2,59E-26
11	138	63,5	4,64E-29
12	50	76,5	3,15E-31
13	77	94	1,48E-33

Te vrednosti možemo da koristimo da bismo poredili predviđanja iz dva modela. Pretpostavite da imamo dva skupa predviđanja, jedno iz modela A, a drugo iz modela B. Računamo funkcije verovatnoće *prethodne jednakosti*, PLA i PLB. Dawid (1984) je dokazao da ako PLA/PLB dosledno rastu kada n raste, onda model A daje tačnija predviđanja od modela B.

Da bismo razumeli kako da koristimo verovatnoće *prethodne jednakosti*, proračunaćemo funkcije za Littlewoodov model nehomogenog Poissonovog procesa i model Jelinski-Moranda, koristeći skup podataka Musa. Rezultati su prikazani u tabeli 13.3, gde je n broj predviđanja na kome se zasniva odnos verovatnoće *prethodne jednakosti*. Na primer, kada je n = 10, odnos obuhvata predviđanja *T35,...,T44*; kada je n = 15, on obuhvata *T35,...,T49*. Odnos se ne povećava mnogo brzo, sve dok ne premašimo 60 posmatranja; onda odnos veoma brzo postaje vrlo velik. Ta analiza nagoveštava da Littlewoodov model nehomogenog Poissonovog procesa daje bolja predviđanja od modela Jelinski-Moranda.

TABELA 13.3 Verovatnoće *prethodne jednakosti* poredeći dva modela

n	Prekvencijalna verovatnoća LNHPP:JM
10	1,28
20	2,21
30	2,54
40	4,5 5
50	2,14
60	4,15
70	66,0
80	1516
90	8647
100	6727

Fenton i Pfleeger (1997) predlažu da se obrati pažnja kada se koristi ta tehnika analize. Oni nam kažu da „činjenica da je određeni model davao bolja predviđanja u prošlosti nije nikakva garancija da će on to nastaviti da radi u budućnosti. Međutim, iskustvo nagoveštava da su takvi preokreti na datom izvoru podataka sasvim retki. Svakako, ako je metod A davao bolja predviđanja od metoda B u nedavnoj prošlosti, izgleda da ima smisla dati prednost metodu A za tekuća predviđanja".

Rekalibrisanje predviđanja

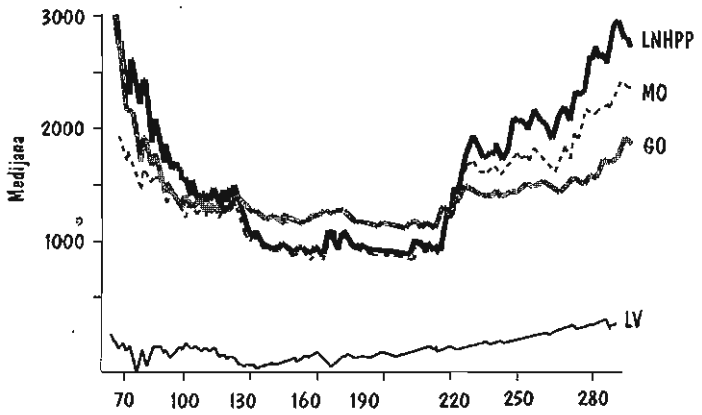
Mi sada imamo više načina da procenimo modele, što nam pomaže da odlučimo koji je najbolji za našu situaciju:

- ispitivanje osnovnih pretpostavki svakog modela;
- traženje polarizacije;
- traženje šuma,
- proračun odnosa verovatnoće *prethodne jednakosti*

TABELA 13.4 Podaci Musa SS3, pokazuju vreme izvršenja do otkaza u sekundama, pročitano sa leva na desno

107400	17220	180	32880	960	26100	44160	333720	17820
40860	18780	960	960	79860	240	120	1800	480
780	37260	2100	72060	258704	480	21900	478620	80760
1200	80700	688860	2220	758880	166620	8280	951354	1320
14700	3420	2520	162480	520320	96720	418200	434760	543780
8820	488280	480	540	2220	1080	137340	91860	22800
22920	473340	354901	369480	380220	848640	120	3416	74160
262500	879300	360	8160	180	237920	120	70800	12960
300	120	558540	188040	56280	420	414464	240780	206640
4740	10140	300	4140	472080	300	87600	48240	41940
576612	71820	83100	900	240300	73740	169800	1	302280
3360	2340	82260	559920	780	10740	180	430860	166740
600	376140	5100	549540	540	900	521252	420	518640
1020	4140	480	180	600	53760	82440	180	273000
59880	840	7140	76320	148680	237840	4560	1920	16860
77040	74760	738180	147000	76680	70800	66180	27540	55020
120	296796	90180	724560	167100	106200	480	117360	6480
60	97860	298580	391380	180	180	240	540	336900
264480	847080	26460	349320	4080	64680	840	540	589980
332380	94140	240060	2700	900	1080	11580	2160	192720
87840	84360	378120	58500	83880	158640	660	3180	1560
3180	5700	226560	9840	69060	68880	65460	402900	75480
380220	704968	505680	54420	319020	95220	5100	6240	49440
420	667320	120	7200	68940	26820	448620	339420	480
1042680	779580	8040	1158240	907140	58500	383940	2039460	522240
66000	43500	2040	600	226320	327600	201300	226980	553440
1020	960	512760	819240	801660	160380	71640	363990	9090
227970	17190	597900	689400	11520	23850	75870	123030	26010
75240	68130	811050	498360	623280	3330	7290	47160	1328400
109800	343890	1615860	14940	680760	26220	376110	181890	64320
468180	1568580	333720	180	810	322110	21960	363600	

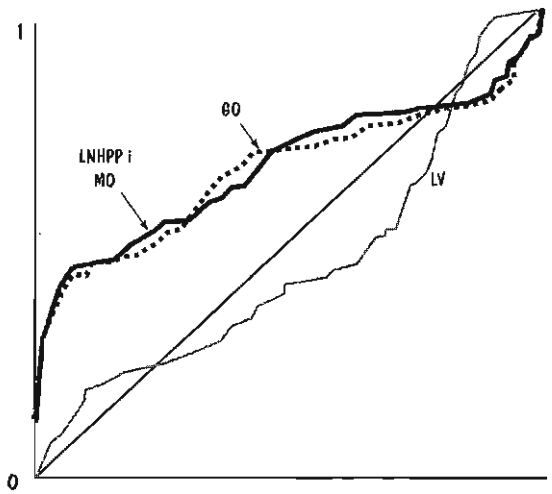
Međutim, nijedna od tih tehnika ne ukazuje na najbolji model. Štaviše, modeli se različito ponašaju na raznim skupovima podataka; možemo da vidimo veoma različite rezultate čak i na istom skupu podataka. Na primer, razmotrite podatke komutacionog sistema u tabeli 13.4. Možemo da ih koristimo sa više modela pouzdanosti i iscrtamo krive predviđanja (106 od 278), kao što je prikazano na slici 13.4. Slika prikazuje da modeli Musa-Okumoto, Goel-Okumoto i Littlewoodov model nehomogenog Poissonovog procesa imaju dosta slično ponašanje, ali je model Littlewood-Verrall veoma različit.



SLIKA 13.4 Predviđanja pouzdanosti različitih modela, koristeći podatke iz tabele 13.4

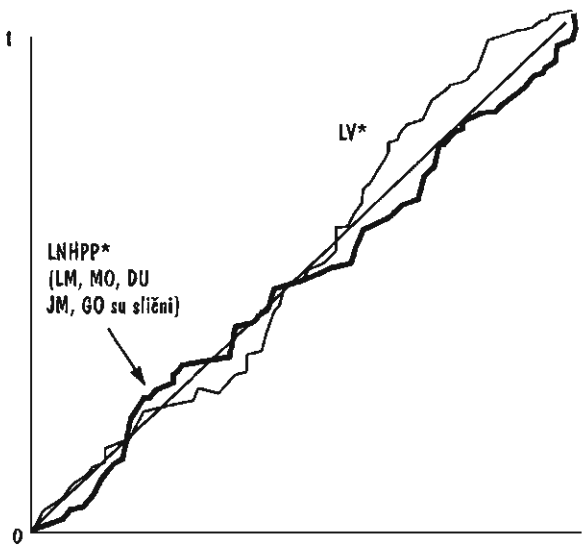
Ako izvedemo analizu verovatnoće *prethodne jednakosti*, naći ćemo da pesimističniji model Littlewood-Verrall u stvari bolje predskazuje od ostalih. Možemo takođe da nacrtamo *u*-krive za te modele, prikazano na slici 13.5, što pokazuje da su svi modeli veoma loši.

Da bismo izašli na kraj sa ukupnom netačnošću tih modela, razmotrićemo njihovo rekalibrisanje, učeći iz netačnosti kada se one pojavljuju. Odnosno, možemo da koristimo prvo nerazumevanje ponašanja modela da bismo poboljšali buduća predviđanja.



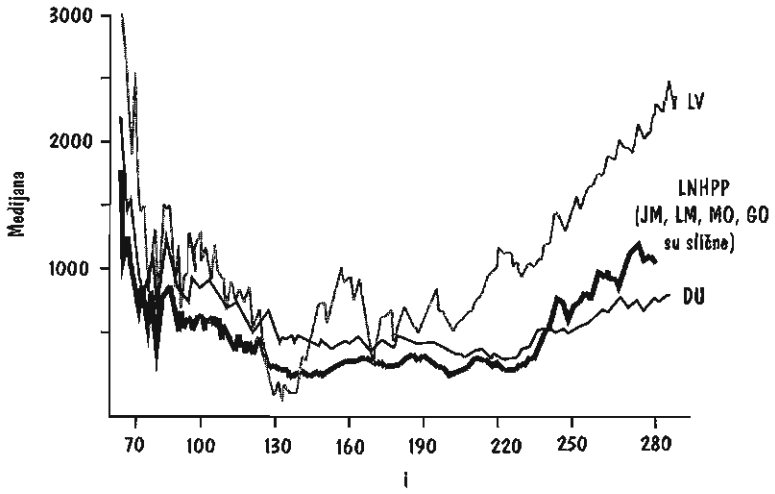
SLIKA 13.5 *u*-krive modela koristeći podatke iz tabele 13.4

SLIKA 13.6 *u*-krive rekalibrisanih modela koristeći podatke iz tabele 13.4



Posebno, iz modela M, možemo da naučimo iz grešaka u prošlosti i formiramo nov model predviđanja, M^*, zasnovan na *u*-krivama. Procedura rekalibracije je izvan domena ove knjige; za više informacija pročitajte Fentona i Pfleegera (1977). Međutim, možemo da razmotrimo rezultat rekalibracije da bismo videli razliku u kvalitetu predviđanja koju ona čini. Na slici 13.6 prikazane su *u*-krive za rekalibracije modela sa slike 13.5.

Možemo takođe da pogledamo sama predviđanja, kao što je prikazano na slici 13.7. Tu postoji mnogo bliža saglasnost između rekalibrisanih, nego što je bila kod prvobitnih modela.



SLIKA 13.7 Predviđanja rekalibrisanih modela na osnovu podataka iz tabele 13.4

Dakle, rekalibracija je donela dve ključne koristi:

1. modeli su u bližoj saglasnosti nego ranije;
2. nove modele sa manje polarizacije od prvobitnih.

Nijedna od ovih tehnika nije posebno namenjena za modelovanje pouzdanosti; možemo da ih koristimo da bismo poboljšali svaki model predviđanja. Dakle, za svaki sistem za predviđanje, možemo da iskoristimo ponašanje u prošlosti da bismo izgradili model i zatim primenili tehnike predstavljene u ovom odeljku da bismo poboljšali predviđanja.

13.2 POBOLJŠAVANJE PROIZVODA

Proučili smo više primera razvoja i održavanja proizvoda: specifikacija zahteva, dokumenata dizajna, koda, test podataka, dokumentacije i korisničkih vodiča, da navedemo samo neke od njih. Jedan od ciljeva softverskog inženjerstva je da se koriste odgovarajuće tehnike za poboljšavanje tih proizvoda, čineći ih lakšim za upotrebu, sa manje grešaka i efektivnijim u izvođenju posla za koji su namenjeni.

U ovom odeljku, ispitujemo dve strategije poboljšavanja proizvoda, inspekcije i višekratna upotrebljivost da bismo pokazali kako je njihovo uvođenje donelo merljiva poboljšanja industrijskih proizvoda. Posmatramo efekte tih tehnika na gustinu grešaka; u podsetniku 13.1 opisuje se kako greške mogu da se nadgledaju u toku razvoja.

Inspekcije

Barnard i Price (1994) izveštavaju da je kompanija AT&T bila zainteresovana za upotrebu inspekcija koda da bi poboljšala kvalitet svog softvera. Međutim, pretraživanje kroz literaturu otkrilo je veliku varijaciju u procentu grešaka uklonjenih pomoću inspekcija: od 30% do 70%. U nastojanju da povećaju broj uklonjenih grešaka, Barnard i Price su koristili skup od devet merenja, stvorenih na osnovu potreba poslovanja i namenjenih planiranju, nadgledanju, kontrolisanju i poboljšavanju inspekcija. Metrike govore AT&T-u ne samo da li se kvalitet koda povećava kao rezultat inspekcija, nego takođe koliko je efektivno osoblje u pripremanju i inspekciji koda. U tabeli 13.5 navedena su neka od merenja, sa primerima vrednosti iz dva uzorka projekta.

TABELA 13.5 Statistički podaci inspekcije koda u AT&T-u

Merenja	Prvi uzorak projekta	Drugi uzorak projekta
Broj inspekcija u uzorku	27	55
Ukupno hiljada redova pregledanog koda	9,3	22,5
Prosek broja linija pregledanog koda (veličina modula)	343	409
Prosek brzine pripreme (linija koda po satu)	124	121,9
Prosek brzine inspekcije (linija koda po satu)	172	154,8
Ukupno otkrivenih grešaka (vidljivih i nevidljivih) po hiljadama linija koda	106	89,7
Procenat ponovnih inspekcija	11	0,5

PODSETNIK 13.1 NADGLEDANJE UNOŠENJA I OTKRIVANJA GREŠAKA

Humphrey (1995) predlaže više tehnika za nadgledanje grešaka i merenje efektivnosti inspekcije. On predlaže da napravimo bazu podataka o greškama koja sadrži ime programa, broj grešaka i njenu vrstu. Pored toga, trebalo bi da pratimo aktivnost razvoja u toku koje je greška unesena u proizvod, aktivnost u toku koje je pronađena i uklonjena i vreme potrebno da se nađe i otkloni. Ta informacija nam pomaže da razumemo odakle dolaze naše greške i koje vrste aktivnosti bi trebalo da se poboljšaju. Poboljšanja se prave na dva fronta: na poboljšanju aktivnosti gde dolazi do unošenja, tako da pre svega možemo da pokušamo da izbegnemo unošenje greške, i na poboljšanju metoda otkrivanja grešaka, tako da one greške koje se unesu budu otkrivene što je pre moguće.

Humphrey nas takođe ohrabruje da proračunamo „doprinos" raznih aktivnosti pregleda, veoma slično kao što je Graham definisao efektivnost ispitivanja. Na primer, u tabeli 13.6 prikazano je kako da se prati gde su greške unete i gde su pronađene. U tabeli se pokazuje da su četiri greške otkrivene za vreme inspekcije dizajna, od kojih su dve unete u toku planiranja, a dve u toku detaljnog projektovanja. Za vreme kodovanja, pronađene su još dve greške, obe nastale u toku detaljnog projektovanja. Zatim su još tri greške otkrivene u toku inspekcije koda: jedna iz detaljnog projekta, a još dve iz procesa kodovanja. Slično tome, pet grešaka je otkriveno u toku prevođenja, četiri za vreme testiranja i dve posle se razvoj završio. Ako analiziramo gde su te greške unete, možemo da proračunamo doprinos dva procesa inspekcije, kao i ukupan doprinos razvoja. Ti brojevi nam pomažu da bolje razumemo gde se greške unose i kako možemo da poboljšamo kvalitet našeg proizvoda.

TABELA 13.6 Proračun prinosa

		Unesene greške					
Aktivnost	Pronađene greške	Inspekcija projekta	Kôd	Inspekcija koda	Prevođenje	Testiranje	Posle razvoja
Planiranje	0	2	2	2	2	2	2
Detaljno projektovanje	0	2	4	5	5	6	6
Inspekcija projekta	4						
Kôd	2			2	7	10	12
Inspekcija koda	3						
Prevođenje	5						
Testiranje	4						
Posle razvoja	2						
UKUPNO	20						
Doprinos detaljne inspekcije		4/4=100%	4/6=67%	4/7=57,1%	4/7=57,1%	4/8=50%	4/8=50%
Doprinos inspekcije koda				3/5=60%	3/10=30%	3/14=25,5%	3/16=18,8%
Ukupan doprinos		4/4=100%	6/6=100%	9/9=100%	9/14=64,3%	9/16=56,3%	9/20=45%

Za prvi uzorak projekta, istraživači su pronašli da je 41% od svih inspekcija vođeno brzinom većom od 150 linija koda na sat, što je preporučio Fagan. U drugom projektu, kod inspekcija sa brzinama ispod 125, pronađeno je u proseku 46% više grešaka po hiljadama linija koda nego kod onih sa većim brzinama. Taj nalaz znači da više grešaka može da se pronađe kada su brzine inspekcija manje, ili da pronalaženje više grešaka prouzrokuje usporavanje inspekcije. Informacija omogućava kompaniji AT&T da skroji proces inspekcije tako da se pronađe više grešaka, a proizvodi na taj način poboljšaju.

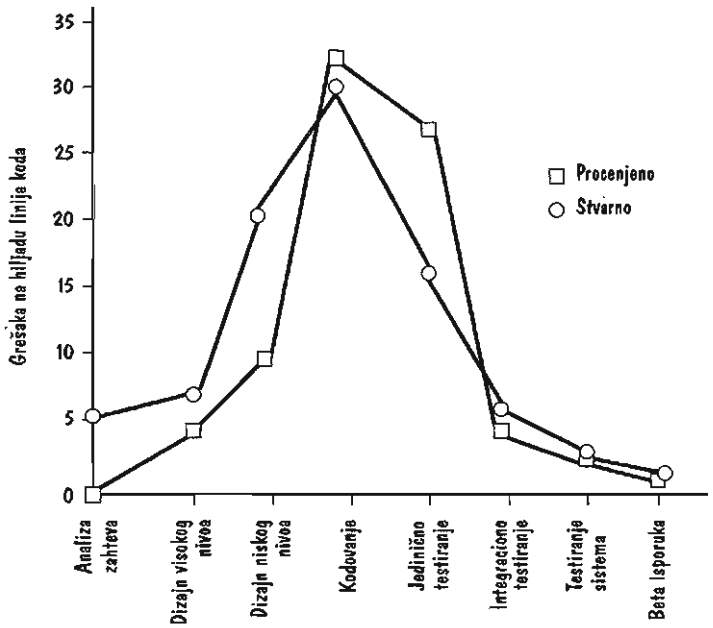
Weller (1994) izveštava o sličnom iskustvu u kompaniji Bull HN Information Systems. Bullovi softverski inženjeri prate stvarne greške pronađene u toku razvoja i porede ih sa procenjenim greškama koje su očekivali da pronađu, kao što je prikazano na slici 13.8.

Kada je gustina grešaka manja od očekivane, razvojni tim pretpostavlja da je to iz jednog od sledeća četiri razloga:

1. Inspekcije ne otkrivaju sve greške koje bi trebalo.
2. Dizajn nema dovoljan sadržaj.
3. Projekat je manji nego što je planirano.
4. Kvalitet je bolji od očekivanog.

Slično tome, ako je gustina grešaka veća od očekivane, dešava se sledeće:

- Proizvod je veći nego što je planirano.
- Inspekcije dobro otkrivaju greške.
- Kvalitet proizvoda je nizak.



SLIKA 13.8 Projektovane prema stvarnim greškama nađenim u toku inspekcije i testiranja

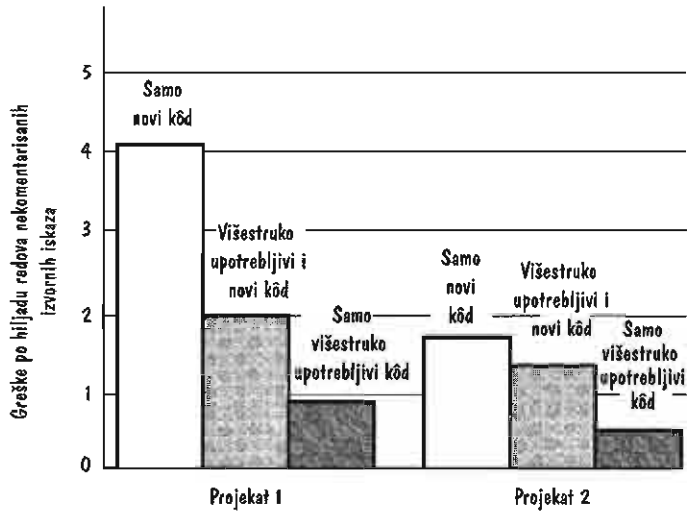
Na primer, ako je očekivani rezultat inspekcije gustina grešaka 0,5 do 1,0 grešaka po stranici, a ona pada na 0,1 do 0,2 greške po stranici, onda je dokument možda nepotpun. Kada je otkrivanje grešaka ispod 0,5 grešaka po stranici, tim istražuje da bi se uverio da su inspektori posvetili dovoljno vremena da se pripreme za inspekciju.

Weller (1994) izveštava da postoji razlika od 7:1 u unošenju grešaka među projektima. Odnosno, neki projekti isporučuju proizvode sa sedam puta više grešaka od drugih. Ali za isti tim koji radi sličan posao, brzina unošenja grešaka ostaje približno ista. Poređenjem očekivanih vrednosti sa stvarnim vrednostima, Bullovi timovi mogu da odrede kako da pronađu greške ranije u toku razvoja i kako da svoj proces inspekcije učine efektivnijim. Rezultat je postepeno poboljšavanje kvaliteta proizvoda.

Višekratna upotrebljivost

Višekratna upotrebljivost je dugo bilo nametano kao metod za poboljšanje kvaliteta proizvoda. Višekratnom upotrebljivošću proizvoda koji su već bili ispitani, isporučeni i upotrebljavani na nekom drugom mestu, izbegavamo pravljenje istih grešaka dva puta. Možemo da iskoristimo rad drugih projektanata i koristimo istorije grešaka i promena komponente dizajna ili koda da bismo potvrdili da će ona dobro raditi i u novoj postavci.

Ono što iznenađuje je da postoji malo empirijskih informacija o efektima višekratne upotrebljivosti na kvalitet. Iz svog rada kod Hewlett-Packarda, Lim (1994) nam pokazuje kako višekratna upotrebljivost poboljšava kvalitet koda. On je izveo dve studije slučaja da bi odredio da li višekratna upotrebljivost stvarno smanjuje gustinu grešaka. Na slici 13.9 ilustrovana je dramatična razlika između gustine grešaka u novom i u višekratno upotrebljivom kodu. Međutim, važno je pogledati gustinu grešaka višekratno upotrebljivog koda kombinovanog sa novim kodom; mnogo grešaka može da se unese u interfejs između ta dva.



SLIKA 13.9 Efekat višekratne upotrebljivosti na greške po hiljadu linija nekomentarisanog izvornog koda

Möller i Paulish (1993) su istraživali odnose koji obuhvataju gustinu i višekratna upotrebljivost kod Siemensa. Oni su pronašli da su ponovo korišćene komponente u kojima je do 25% linija koda prvobitne komponente izmenjeno, imale četiri do osam puta više grešaka od komponenti koje su pisane od samog početka. Dakle, povećanje kvaliteta koje obećava višekratna upotrebljivost komponenti može dalje da se poboljša ako obratimo pažnju na to koliko koda modifikujemo. U nekim slučajevima, ne možemo da izbegnemo promenu prethodno napisanog koda; tada možemo da koristimo dodatne tehnike, kao što su inspekcije ili dodatna ispitivanja, da bismo obezbedili da izmene ne unesu nepotrebne greške.

13.3 POBOLJŠANJE PROCESA

Videli smo kako procesi razvoja softvera mogu da utiču na kvalitet svojih proizvoda. Modeli zrelosti procesa zasnivaju se na ideji da će poboljšanje procesa automatski poboljšati i proizvode, posebno softver. Uže fokusirani procesi, kao što su izrada prototipova i čista soba, takođe imaju za cilj da smanje troškove, povećaju kvalitet i skrate vreme razvoja ili održavanja. U ovom odeljku, govorićemo o istraživanjima koja poboljšavaju same procese, a time indirektno i proizvode.

Zrelost procesa i sposobnosti

U poglavlju 12 napomenuli smo da postoji više predloženih modela za poboljšanje zrelosti ukupnog procesa razvoja, kao što su CMM, ISO 9000 i SPICE. Neki projektanti su mnoge od tih modela prihvatili sa velikim entuzijazmom; neki drugi su odbijali da ih koriste, sem ako im to nije bilo naređeno. U stvari, modeli zrelosti i metode za njihovu procenu su postali *de facto* standardi u mnogim organizacijama. Na primer, za neke ugovore za izradu softvera za vazduhoplovstvo SAD, očekuju se minimalni rezultati modela CMM, a takvi rezultati imaju značajan efekat na odluke o ugovorima za mornaricu SAD (Saiedan i Kuzara 1995). Rugg (1993) ističe da se „sposobnost softvera – merena u podnetom predlogu i na licu mesta – računala kao trećina ukupne težine prilikom razmatranja dodeljivanja ugovora".

Ali, čak i posle uvođenja modela zrelosti procesa, bilo je primedbi na njihovu primenu i upotrebu. Na primer, Bollinger i McGowan (1991) su zapazili mnoge probleme u vezi sa upotrebom prvobitnog upitnika o zrelosti procesa SEI; posebno, oni su istakli da su ograničena pitanja obuhvatila samo mali broj karakteristika dobre softverske prakse, a njihovi odgovori sa da/ne su onemogućili merenje delimične saglasnosti. Oni su takođe zapazili da model zrelosti procesa pretpostavlja proizvodnu paradigmu za softver; kao što smo govorili u poglavlju 1, proizvodnja i replikacija mogu da budu neodgovarajuće analogije za razvoj softvera.

Bollinger i McGowan (1991) takođe tvrde da pristup preko zrelosti procesa ne ulazi dovoljno duboko u to kako se implementiraju prakse razvoja softvera. Na primer, sertifikacija na nivou 2 (ponovljiv) zahteva odgovor „da" na pitanje „Da li menadžeri prve linije razvoja softvera potpisuju svoje procene rokova i troškova?" To pitanje znači da projektom na nivou 2 moraju da upravljaju oni koji su voljni da preuzmu odgovornost za svoje procene. Međutim, nema pitanja o tome da li menadžeri ocenjuju tačnost svojih procena ili poboljšavaju proces procene kada nauče više iz modela procesa i povratne sprege. Opasnost u izvlačenju samo uske informacije o ključnim procesima ili u praksi

leži u tome što ona može da pruži varljivu sliku projekta i njegovog upravljanja. U takvom slučaju, upotrebljeni su neodgovarajući modeli i netačna merenja, a rukovodstvo, svejedno može da bude „nagrađeno" procenom o nivou 2.

Čak i sa svojim nedostacima, da li ti okviri zrelosti stvarno funkcionišu? Da li uspon po lestvicama zrelosti automatski ima za posledicu da projektant proizvodi bolji softver? Na primer, Pfleeger i McGowan (1990) nagoveštavaju da poboljšana zrelost može da dovede do bolje vidljivosti unutrašnjih aktivnosti procesa (pročitajte podsetnik 13.2). Institut SAD za softversko inženjerstvo (SEI), kao osnivač modela žrelosti sposobnosti, preduzeo je istraživanje o efektima poboljšanja procesa. Pod okriljem SEI, Herbsleb i saradnici (1994) su prikupili podatke iz 13 organizacija koje su predstavljale različite nivoe zrelosti. Ispitujući promene performanse u toku vremena kako su se implementirale aktivnosti poboljšanja softverskog procesa, istraživački tim je identifikovao dobitke u produktivnosti, ranom otkrivanju grešaka, vremenu izlaska na tržište i kvalitetu. Rezultati su prikazani u tabeli 13.7.

PODSETNIK 13.2 ZRELOST PROCESA I POVEĆANA VIDLJIVOST

Pfleeger i McGowan (1990) opisali su kako zrelost procesa može da utiče na uvid organizacije ili projekta u proces i, na taj način, na razumevanje pitanja u vezi sa procesom. Mada nije strogo izveden iz modela zrelosti sposobnosti Instituta za softversko inženjerstvo, njihov pojam povećane vidljivosti vam dozvoljava da odlučite šta ima smisla, zasnovano na onome što vidite u slici procesa.

Da biste videli kako to radi, razmotrite sledeći primer. Pretpostavite da vašu organizaciju brine promenljivost zahteva. Cilj razumevanja razloga i efekata promene zahteva nagovestava da bi vaša grupa za razvoj trebalo da meri i broj zahteva i broj promena u tim zahtevima. Na najnižem nivou vidljivosti (bliskom nivou 1 CMM), zahtevi su loše definisani. Tu merimo broj zahteva i broj promena za svaki zahtev, što bolje možemo. Na prvom sledećem nivou (sličnom nivou 2 CMM), zahtevi su dobro definisani, ali aktivnosti procesa nisu. Na tom stepenu, možemo da merimo broj zahteva po vrsti, kao i promene zahteva po vrsti. Sada ne samo da znamo koliko se zahtevi menjaju, nego možemo da kažemo i da li se promene javljaju prvenstveno u zahtevima za interfejs, performansu, bazu podataka ili su raspodeljeni kroz specifikaciju sistemu. Akcije koje se preduzimaju na osnovu tih merenja imaju solidniju osnovu i verovatno će biti efektivnije.

Slično tome, na još višem nivou (veoma sličnom nivou 3 CMM), aktivnosti procesa su jasno diferencirane; na primer, menadžer projekta može da kaže kada je dizajn završen i počinje kodovanje. Tu zahtevi mogu da se mere kao i ranije, uključujući broj zahteva i broj promena u zahtevima po vrsti. Ali, pored toga, zahvaljujući definisanim aktivnostima, možemo da pratimo svaku promenu zahteva do pripadajućeg dizajna, koda ili testiranja komponente, što nam omogućava da analiziramo uticaj promene na ostatak sistema. Povećana zrelost procesa daje nam veću vidljivost, što rezultuje sa mnogo više informacija i boljim razumevanjem sistema nego što smo imali na nižim nivoima.

TABELA 13.7 Ukupni rezultati iz SEI-jeve studije dobitaka (Herbsleb i dr. 1994)

Kategorija	Opseg	Medijana
Ukupni godišnji troškovi aktivnosti poboljšanja softverskog procesa	49000 do 1202000 USD	245000 USD
Godine angažovanja na poboljšanju softverskog procesa	1 do 9	3,5
Troškovi poboljšanja softverskog procesa po inženjeru	490 do 2004	1375
Godišnji dobitak u produktivnosti	9-67%	35%
Dobitak u ranom otkrivanju (greške otkrivene pre testa)	6-25%	22%
Godišnje smanjenje vremena za izlazak na tržište	15-23%	19%
Godišnje smanjenje izveštaja o greškama posle puštanja u rad	10-94%	39%
Poslovna vrednost ulaganja u poboljšanje softverskog procesa (vrednost vraćena po svakom uloženom dolaru)	4,0 do 8,8	5,0

Ta studija daje vrlo pozitivnu sliku o poboljšanju softverskog procesa. Međutim, moramo da budemo oprezni u vezi sa nagoveštajima da rezultati ukazuju na opštu situaciju. Organizacije koje su učestvovale u studiji su se dobrovoljno javile; dakle, ta grupa nije bila slučajni uzorak veće populacije. Projekti nisu bili karakterizovani na način koji nam dozvoljava da poredimo jedan sa drugim, rad uložen u poboljšanje nije bio isti na svim projektima i nije izvršeno merenje da bi se odredilo koliko su projekti bili reprezentativni. Iako možemo da vidimo kako su neki napori na poboljšanju procesa bili korisni za taj uzorak, ne možemo da zaključimo da je poboljšanje softverskog procesa korisno u opštem slučaju.

Nije jasno kako bi se rezultati sagledavali u širem kontekstu poslovne vrednosti. „Vraćena vrednost" u Herbslebovoj studiji izgleda da je merena u terminima rane detekcije grešaka, smanjenja vremena za izlazak na tržište i smanjenja operacionalnih otkaza. Ali te karakteristike ne rešavaju pitanja zadovoljstva kupca ili odgovarajuće funkcionalnosti. To znači da one posmatraju tehnički, a ne poslovni kvalitet, pa daju samo delimičnu sliku poboljšanja. Iz tih rezultata ne možemo da odredimo da li je usvajanje okvira zrelosti dobro za poslovanje. Kao što se pokazuje u podsetniku 13.3, mogu da postoje slučajevi kada velika zrelost može stvarno da ograniči poslovnu fleksibilnost.

Ako su modeli i merenja netačni ili pogrešno vođeni, rezultat može da bude pogrešno dodeljivanje resursa, poslovni gubitak i još mnogo toga.

Card (1992) izveštava da su dobijeni nekonsistentni rezultati iz CMM procena iste organizacije koju su izvršili različiti timovi; dakle, moramo da se zapitamo koliko su pouzdane CMM ocene. Tu se pouzdanost odnosi na obim u kome ista procedura merenja daje iste rezultate prilikom ponovljenih pokušaja.

El Emam i Madhavji (1995) su dalje istraživali pouzdanost, pitajući:

- Koliko su pouzdane takve procene?
- Kakve su implikacije pouzdanosti za interpretiranje rezultata procene?

PODSETNIK 13.3 DA LI ZRELOST SPOSOBNOSTI UNAZAĐUJE NASU?

Više istraživača i praktičara se zapitalo da li nam model zrelosti sposobnosti pomaže da bolje radimo ono što već radimo, ali nam ne pruža fleksibilnost da probamo nove stvari, ili da se menjamo i tehnički razvijamo. Da biste razumeli njihovu zabrinutost, razmotrite softver u Nasinom spejs šatlu. Njega je izgradila i održavala organizacija koja je bila ocenjena nivoom 5 na skali CMM.

Softver je bio izuzetno pouzdan, imao je vrlo malo operacionalnih otkaza. Međutim, softver je radio prvenstveno pomoću tabela. Pre svakog lansiranja, Nasa je morala da razvije nove tabele podataka da bi opisala lansiranje i kontrolisala softver. Proces ažuriranja tih tabela uzima dosta vremena i rada, pa je zato skupa aktivnost koja troši vreme i može da dovede do kašnjenja datuma lansiranja. Procesi razvoja softvera i održavanja, koji su dobili ocenu nivoa 5, podržavaju reviziju tabele. Moguće je da glavna promena u procesu razvoja, delom da bi se izvršila revizija pristupa zasnovanog na tabeli i sistem učini fleksibilnijim, može da rezultuje da proces dobije manju CMM ocenu; ako je tako, moguće je da očekivanje sledeće procene zrelosti obeshrabruje projektante da pokušaju nov, inovativan pristup. Drugim rečima, nije sasvim jasno da li bi reinženjerstvo ili rekonstrukcija softvera za spejs šatl bili sprečeni ili potpomognuti optimizovanim postojećim procesom u Nasi.

Oni su izgradili model organizacione zrelosti zasnovan na CMM-u i više drugih popularnih modela i izvršili studiju slučaja da bi rešili ta pitanja. Pronašli su jasan dokaz

nepouzdanosti kada su merili duž četiri dimenzije: standardizacije, upravljanja projektom, alata i organizacije. Štaviše, kada su istraživali odnos između organizacione zrelosti i drugih atributa procesa i proizvoda, pronašli su mali, značajan odnos između zrelosti i kvaliteta usluge, ali „nikakav odnos nije pronađen sa kvalitetom proizvoda" i „malu negativnu korelaciju između dimenzija standardizacije i upravljanja projektom i kvaliteta projekata".

Pitanja koja su iskrsla o okvirima za poboljšanje procesa nisu posebna za CMM. Seddon (1996) šalje istu poruku u svom izveštaju o efektima ISO 9000 na više poslovanja u Velikoj Britaniji. On kaže da će „ISO 9000, zbog svoje implicitne teorije kvaliteta, dovesti do opštih problema u implementaciji; problema koji će oštetiti ekonomsku performansu i koji mogu da spreče menadžere za bilo kakvo saznanje o mogućoj ulozi kvaliteta u poboljšanju produktivnosti i položaju u konkurenciji". Zaista, britanska organizacija Advertising Standards Authority je odredila da British Standards Institute mora da se uzdrži od tvrdnji da će usvajanje standarda ISO 9000 poboljšati kvalitet i produktivnost. U novinama koje opisuju slučaj, Seddon napominje: „U svakom slučaju koji smo proučavali, videli smo da je ISO 9000 nanosio štetu produktivnosti i položaju u konkurenciji. Direktori svake od tih organizacija su verovali da je ISO 9000 bio od koristi: svi oni, bili su zavedeni" (ESPI Exchange 1996).

Dakle, postoje važna pitanja o merenjima kojima treba da se bavimo prilikom razmatranja upotrebe tih procesnih i organizacionih okvira. Moramo da razumemo koliko su pouzdana i validna merenja i modeli, da znamo koji entiteti i atributi se mere i da ispitamo odnose između rezultata zrelosti i ponašanja da se za zrelost „pretpostavlja" da proizvodi ili poboljšava.

PODSETNIK 13.4 POREĐENJE VIŠE TEHNIKA ZA PROCENU ODRŽAVANJA

De Almeida, Lounis i Melo (1997) su upotrebili algoritme mašinskog učenja da bi predvideli skupe softverske komponente, sklone greškama. Koristeći podatke prikupljene u Nasinom istraživačkom centru, u Goddardovom centru za kosmičke letove i merenja proizvoda izvučena iz Ada koda, oni su klasifikovali komponente za koje je bilo, ili nije bilo skupo da se isprave. Otkrili su da su induktivni logički modeli programiranja bili tačniji od indukcionih stabala odozgo nadole, indukcionih pravila za vrednosti atributa odozgo nadole i pokrivajućih algoritama.

Održavanje

U poglavlju 11, napomenuli smo da troškovi održavanja rastu i često prevazilaze troškove razvoja. Prema tome, važno je istražiti načine kako da se poboljša proces održavanja, smanjujući te troškove, uz održanje ili čak poboljšanje kvaliteta. Kako se ističe u podsetniku 13.4, postoji više modela za izbor; nije jasno koji modeli odgovaraju datoj situaciji. Henry i saradnici (1994) su se bavili tom problematikom u Government Elwectronic Systems Divisionu kao glavnom ugovaraču, pokušavajući da odgovore na sledeća pitanja:

1. Kako možemo kvantitativno da procenimo proces održavanja?
2. Kako možemo da iskoristimo tu procenu da bismo poboljšali proces održavanja?
3. Kako možemo kvantitativno da procenimo efektivnost poboljšanja bilo kojih procesa?

Koristeći metod koji se zasniva na uobičajenim statističkim testovima (kao što su višestruka regresija, korelacija nizova i hi-kvadrat testovi nezavisnosti u dvostrukim tabelama slučajnosti), oni su dosta saznali o procesu održavanja; posebno, posmatrali su kako promene zahteva utiču na atribute proizvoda.

Na primer, istraživački tim je želeo jednostavnu šemu klasifikacije za softverske komponente, koja bi mu dozvolila da predviđa koje od njih su sklone greškama. Oni su napravili tabele slučajnosti zasnovane na vrednostima medijane ispravljenih grešaka i broja nadgradnji i promena specifikacija nadgradnje koje su uticale na komponentu. Otkrili su značajnu korelaciju između ispravljenih grešaka i uticaja nadgradnje i iskoristili su odnos da bi rangirali komponente. Biranjem komponenti za detaljnije ispitivanje (kao što su više ispitivanja ili dodatno pregledanje) prema njihovom odnosu sa srednjim brojem stavki nadgradnje koje su uticale na komponentu, oni su tačno identifikovali 93% komponenti sa učestanošću grešaka iznad medijane.

Henry i saradnici (1994) su takođe ispitivali odnos više atributa projekta prema neuspesima inženjerskog testiranja. Kao rezultat te analize, grupa za inženjersko testiranje je promenila svoju ulogu i strategiju testiranja. Inženjersko testiranje se sada usredsredilo na testiranje regresije poboljšanja prema poboljšanju, a grupa za inženjersko testiranje sada nadgleda odvijanje testa preko podugovarača, zahtevajući detaljne izveštaje o testu.

Istraživačka grupa je u toku proučavanja procesa održavanja mnogo saznala o održavanju, a njihovi predlozi su doveli do merljivog procesa i poboljšanja proizvoda. Ali, oni su takođe mnogo naučili i o samom procesu procene. Oni napominju da, kada se procenjuje poboljšanje procesa, treba imati na umu tri stvari.

1. Pažljivo koristite statističke tehnike, zato što jedna tehnika ne može da proceni pravi efekat poboljšanja procesa. Na primer, kada su razmatrali samo medijanu i korelaciju nizova uticaja nadgradnje na pouzdanost proizvoda, nisu videli nikakvo poboljšanje. Ali, kada su upotrebili srednju vrednost i standardnu devijaciju, poboljšanje se jasno pokazalo.
2. U nekim slučajevima, poboljšanja procesa moraju da budu veoma dramatična ako kvantitativni efekti treba da se pokažu u statističkim podacima. Na primer, tabele slučajnosti koje su korišćene da se klasifikuju komponente po sklonosti greškama, menjale su se vrlo malo kada se proces poboljšavao, sve dok skoro sve greške nisu bile uklonjene.
3. Poboljšanje procesa utiče na linearnu regresiju rezultata na različite načine. Posebno, kada se povećavala tačnost jednačine, njihovi efekti na pojedinačne promenljive su se razlikovali.

Čista soba

U toku dve decenije, Nasina Laboratorija za softversko inženjerstvo (SEL) je procenjivala i poboljšavala procese. Basili i Green (1994) opisuju kako SEL uvodi novu tehnologiju, procenjuje njene efekte i koristi prednosti onih alata i tehnika koji nude značajno poboljšanje. SEL uzima u obzir rizike u vezi sa upotrebom nove tehnologije; gde to odgovara, tehnika ili alat se primenjuju izvan normalnog okruženja projekta, tamo gde njihova primena neće ugroziti ciljeve projekta.

Te „oflajn" studije se obično izvode kao formalni, kontrolisani eksperimenti ili studije slučaja. SEL obično počinje sa malim eksperimentom, gde njegova veličina dozvoljava da se promenljive lako kontrolišu. Onda, kada eksperimentalni rezultati izgledaju obećavajuće, studija slučaja industrijske snage verifikuje da će rezultati u maloj razmeri raditi i u stvarnom okruženju. Jednom kada je SEL zadovoljna saznanjem da će nova tehnika biti dobra za Nasine projektante i servisere, ona priprema materijal o naučenim lekcijama, tako da drugi mogu da razumeju i koriste tehnologiju.

Basili i Green (1994) su istraživali ključne procese u čistoj sobi da bi videli kako bi oni bili od koristi za Nasu. Oni su svoje studije organizovali u pet delova:

1. kontrolisani eksperiment u kome se porede čitanja sa ispitivanjem;
2. kontrolisani eksperiment u kome se porede čista soba sa čistom sobom plus ispitivanjem;
3. studija slučaja u kojoj se ispituje čista soba na razvojnom timu od tri člana i ispitnom timu od dva člana;
4. studija slučaja u kojoj se ispituje čista soba na razvojnom timu od četiri člana i ispitnom timu od dva člana;
5. studija slučaja u kojoj se ispituje čista soba na razvojnom timu od četrnaest članova i ispitnom timu od dva člana.

TABELA 13.8 Rezultati eksperimenta 1 čitanja prema ispitivanju

	Čitanje	Funkcionalno ispitivanje	Strukturalno ispitivanje
Srednji broj otkrivenih grešaka	5,1	4,5	3,3
Broj grešaka otkrivenih po satu upotrebe tehnike	3,3	1,8	1,8

Eksperimenti. U prvom eksperimentu, Basili i Green (1994) su koristili zasejavanje grešaka da bi poredili čitanje stepenaste apstrakcije, ispitivanja granične vrednosti deljenja ekvivalencije i strukturnog ispitivanja pokrivanja iskaza. Njihovi rezultati su prikazani u tabeli 13.8.

Oni su takođe razmatrali poverenje u rezultat. Posle eksperimenta, čitaoci su mislili da su pronašli oko polovine grešaka i bili su približno korektni. Ali, ispitivači su mislili da su pronašli gotovo sve greške, što nikada nije bilo tačno. Basili i Green (1994) smatraju da izvođenje velikog broja test slučajeva daje ispitivačima pogrešno osećanje poverenja.

Čitaoci su takođe pronašli više klasa grešaka, uključujući greške interfejsa, nagovještavajući da bi se rezultati uvećali za veće projekte.

U drugom eksperimentu, Basili i Green (1994) su potvrdili tradicionalno SEL-ovo oslanjanje na testiranje. Oni su osetili da je suviše rizično potpuno ukloniti testiranje izvan kontrole projektanata, pa su zato tako projektovali eksperiment da porede tradicionalnu čistu sobu sa čistom sobom gde je testiranje bilo dozvoljeno. Evo nekih nalaza do kojih su došli:

- Projektanti u čistoj sobi su bili efikasniji u oflajn čitanju.
- Timovi iz čiste sobe plus testiranje su se više usredsredili na funkcionalno testiranje nego na čitanje.
- Timovi iz čiste sobe su provodili manje vremena onlajn i bilo je verovatnije da će ispuniti svoje rokove.
- Proizvodi iz čiste sobe su bili manje složeni, imali su manje globalnih podataka i komentara.
- Proizvodi iz čiste sobe su potpunije ispunili zahteve sistema i imali su veći procenat uspešnih nezavisnih test slučajeva.
- Projektanti iz čiste sobe nisu vrlo strogo primenjivali formalne metode.
- Skoro svi učesnici iz čiste sobe su bili voljni da opet koriste čistu sobu na drugom razvojnom projektu.

S obzirom na to je velika razlika između dva tima bila dozvola da se uradi dodatno ispitivanje, Basili i Green (1994) su predložili da članovi kontrolne grupe (čista soba plus testiranje) ne dobiju vreme da nauče i upotrebe druge tehnike zato što su znali da bi mogli da se pouzdaju u testiranje.

Studije slučajeva. Sve tri studije slučajeva bile su obuhvaćene razvojem softvera za dinamiku leta. Cilj prve studije bio je da se poveća kvalitet i pouzdanost bez povećavanja troškova, kao i da se uporedi čista soba sa standardnim okruženjem u odeljenju za dinamiku leta. S obzirom na to da je SEL već imao osnovicu za razvoj dinamike leta u NasinomGoddardu, istraživači su mogli da porede rezultate čiste sobe i proučavaju

razlike. Proces čiste sobe bio je skrojen, na osnovu rezultata dva eksperimenta, tako da su obuhvaćeni:

- razdvajanje razvojnog tima i tima za testiranje;
- pouzdavanje na pregled sličnog umesto na jedinični test;
- upotreba neformalnih mašina stanja i funkcija da bi se definisala konstrukcija sistema;
- statističko testiranje zasnovano na operacionalnim scenarijima.

Basili i Green (1994) su otkrili da se 6% rada na projektu pomerilo sa kodovanja na dizajn kada je upotrebljena čista soba. Takođe, dok su tradicionalni projektanti potrošili 85% svog vremena na pisanje koda i 15% na čitanje, tim u čistoj sobi je proveo približno oko polovine svog vremena na svakoj aktivnosti. Produktivnost je porasla za 50%, a količina ponovnog rada se smanjila. Međutim, tim je imao teškoće kada je koristio formalne metode, pa su oni zato kombinovali statističko testiranje sa funkcionalnim.

Učeći iz prve studije, Basili i Green (1994) su poboljšali obuku u formalnim metodama i obezbedili su više savetovanja kako da se koriste statističke tehnike. Posebno, naglasili su strukture kutija umesto mašina stanja. U toj studiji slučaja, oni su koristili dizajn sestrinskog projekta, poredeći pristup čiste sobe sa tradicionalnijim pristupom. Rezultati su sumirani u tabeli 13.9.

Učestalost promena i grešaka jasno je bila bolja za tim u čistoj sobi, ali je bilo i nekoliko nedostataka. Učesnicima u čistoj sobi nije se sviđalo da koriste apstrakcije dizajna i strukture kutija, nije im bilo ugodno što ne mogu da prevode svoj kôd i teško su koordinirali projektante i testere.

Treća studija slučaja naučila je svoje lekcije od prve dve. Na raspolaganju je bilo više treniranja u čistoj sobi, a učesnicima je podeljena knjižica o procesu čiste sobe. Rezultati još uvek nisu saopšteni u literaturi.

Zaključci. SEL-ovo iskustvo sa čistom sobom nas je naučilo više stvari. Prvo, Basili i Green (1994) su nam pokazali kako da koristimo kombinaciju eksperimenata i studija slučajeva da bismo poredili novu tehniku sa nekom postojećom. Oni su skrojili i tehniku i istražni proces za obuhvaćenu organizaciju i rezultate prethodnih studija. Odnosno, oni su polako menjali pristup čiste sobe kada su saznali kako su učesnici reagovali na različite aktivnosti čiste sobe. I oni su upotrebili mnogo više od jedne vrste studija slučajeva, tako da bi mogli da kontrolišu varijaciju koliko god je moguće.

TABELA 13.9 Rezultati SEL-ovih studija slučajeva

	Vrednost osnovice	Razvoj u čistoj sobi	Tradicionalan razvoj
Redova koda dnevno	26	26	20
Promena po hiljadu redova koda	20,1	5,4	13,7
Grešaka po hiljadu redova koda	7,0	3,3	6,0

Njihov istraživački rad se nastavlja, što je uobičajeno za zrelu organizaciju. Kako se usvajaju novi alati i tehnike, „tipično" okruženje se menja i poboljšava. Basili i Green (1994) nam nude vredan kvantitativan dokaz efekata čiste sobe u Nasinom Goddardu.

Slične studije možemo da iskoristimo da bismo procenili čistu sobu u sopstvenom okruženju, ali ćemo verovatno dobiti različite rezultate koji odražavaju mogućnosti, potrebe i najpoželjnije izbore u našim sopstvenim organizacijama. Nije važna lekcija da čista soba uvek radi. Mnogo je važnije da čista soba može da radi i da moramo nastaviti istraživanje kako bismo odredili kako da je skrojimo da bi najbolje radila u svakoj posebnoj situaciji.

13.4 POBOLJŠAVANJE RESURSA

Da bi se proizvodio dobar softver, potrebni su mnogi resursi. Moramo da se snabdemo odgovarajućom opremom, alatima i tehnikama i da imamo dovoljno vremena za obavljanje posla. Neki resursi su fiksirani, što ne ostavlja prostor za poboljšanje. Na primer, ako sistem mora da se razvije na određenoj platformi ili na datom programskom jeziku, naši projekti su ponekad ograničeni. Ali drugi resursi su veoma promenljivi, a razumevanje te promenljivosti nam pomaže da ih poboljšamo. Na primer, softverski inženjeri sa istom obukom imaju veoma različite sposobnosti. Svi znamo projektante koji su veoma dobri u pisanju koda, ali loši u testiranju, ili koji su dobri dizajneri, ali slabi analitičari zahteva. Varijacije postoje čak i unutar kategorije; zaista, neki programeri mogu brzo da napišu loš kôd, ili sporo dobar kôd, ili nešto između!

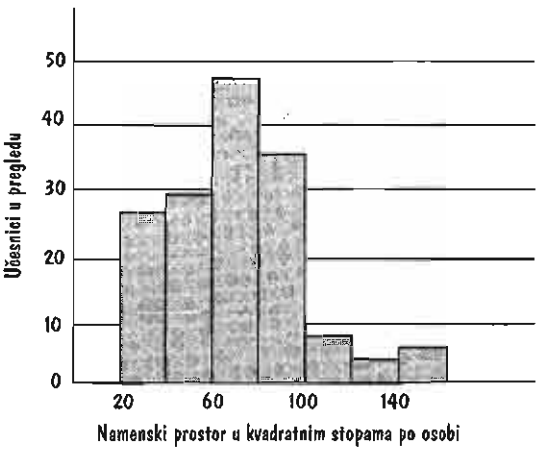
Radno okruženje

Nažalost, u literaturi postoji manje podataka o ulozi ljudi u softverskom inženjerstvu nego o tehnikama i alatima. Većina kvantitativnih analiza usredsređuje se na produktivnost programera sa osnovice, ili procenjivanje kompromisa između troškova i rokova. DeMarco i Lister su među onim malobrojnim istraživačima koji su ispitivali način na koji okruženje utiče na kvalitet rada koji obavljamo. Oni su skovali termin „peopleware" da bi označili raznovrsnost među projektantima i istakli da možemo da preduzmemo korake da poboljšamo kvalitet softvera tako što ćemo ljudima dati okruženje koje im je potrebno da bi dobro obavili posao (DeMarco i Lister 1987).

U poglavlju 3 napomenuli smo da se u studiji McCuea (1978) preporučuje najmanje 100 kvadratnih stopa namenskog radnog prostora po radniku, 30 kvadratnih stopa radne površine po osobi i zaštita od buke. U DeMarcovim i Listerovim anketama programera iz 1984. i 1985. godine (DeMarco i Lister 1985) otkriveno je da je samo 16% učesnika imalo preporučeni minimum prostora, a 58% njih je izjavilo da njihov kancelarijski prostor nije bio dovoljno tih. Rezultati o radnom prostoru prikazani su na slici 13.10.

Da bi pokazali koliko su prostor i buka bitni, DeMarco i Lister su analizirali kvalitet takmičenja u kodovanju, poredeći profile grešaka u bučnim i tihim kancelarijama. Radnici koji su pre takmičenja izvestili da su njihove kancelarije bile tihe imali su 33% veću verovatnoću da isporuče kôd bez grešaka. A što je nivo buke postajao veći, taj trend je bivao sve izraženiji. DeMarco i Lister su preporučili jednostavne, ekonomične mere za poboljšanje okruženja projektanata, kao što je glasovna pošta i prosleđivanje poziva (da bi se eliminisalo zvonjenje telefona), i vrata na kancelarijama (da bi se smanjili nepotrebni prekidi).

SLIKA 13.10 Podna površina za projektante, iz DeMarcovih i Listerovih pregleda



Obično se misli da mali timovi rade bolje od velikih, zato što se broj komunikacionih putanja dramatično povećava kada raste veličina tima. Weller (1993) je potvrdio tu činjenicu kada je ispitivao podatke iz inspekcija. Kao što smo videli u poglavlju 8, tim za inspekciju od tri osobe može da radi isto tako dobro kao i četvoročlani tim. U podsetniku 13.5 objašnjava se kako korisnici mogu da budu vredan resurs; ne smemo da ih zaboravimo kada razmatramo veličinu tima i komunikacione putanje.

DeMarco (1997) ističe važnost „zgusnutog" tima, gde članovi tima rade mirno, koordiniraju svoj rad i poštuju sposobnosti drugih. Iz tog razloga, on predlaže da timovi koji su u prošlosti dobro radili zajedno, budu sačuvani i za buduće projekte. Pred toga, on nas podstiče da koristimo posredovanje prilikom rešavanja sukoba, tako da tim sebe vidi kao jedinstven na jednoj strani sa problemom na drugoj.

Odnos troškova i rokova

Vreme je ključni resurs. Ako na raspolaganju ima dovoljno vremena, razvojni tim može da vrlo pažljivo dizajnira visokokvalitetan proizvod, sveobuhvatno ga testira i provodeći dovoljno vremena sa kupcima i korisnicima obezbedi da svi postignu zajednički nivo razumevanja problema i njegovog rešenja.

Nažalost, nemamo uvek dovoljno vremena. Pritisci tržišta od nas zahtevaju da prodamo proizvode pre naših konkurenata, ili da ponudimo usluge kada ih naši kupci zatraže. Koordinirani pritisci nas prisiljavaju da svoje proizvode stavimo na raspolaganje u trenutku isporuke drugih proizvoda, podstaknuti planovima integracije i testiranja. Dakle, razumevanje odnosa između troškova, rokova i kvaliteta, pomaže nam da planiramo razvoj i održavanje bez žrtvovanja funkcionalnosti ili kvaliteta.

Mnogi modeli procene rada i rokova uključuju ovaj vid analize kompromisa. Na primer, COCOMO opisuje interakciju između rada i rokova i predlaže nominalne mere rada i rokova zasnovane na parametrima projekta (Boehm 1981). Drugi modeli, kao što je Putnamov SLIM, ilustruju efekte skraćenja vremena isporuke, što obično rezultuje povećanom potrebom za osobljem. Međutim, Brooks (1975) nas upozorava da dodavanje osoblja na projekat koji kasni, čini samo da on kasni još više. Slično tome, Lister nam kaže da ljudi pod pritiskom ne misle ništa brže, pa mora postojati minimalna količina potrebnog vremena da bi se zadatak uspešno izvršio (DeMarco 1997).

PODSETNIK 13.5 POSMATRANJE KORISNIKA KAO RESURSA

U poglavlju 11 videli smo kako je Bell Atlantic izgradio svoj SaleService Negotiation System (SSNS) da bi zamenio tri nasleđena sistema. Korišćenjem ekranskih upita, SSNS vodi predstavnike prodaje kroz uređeni ciklus. Aplikacija je stvorila profitabilniji odnos sa Bell Atlanticovim kupcima, čineći Bell Atlantic konkurentnijim na tržištu (Field 1997a).

Jedan od razloga uspeha SSNS-a je što su njegovi projektanti upotrebili korisnike kao resurs. Formiranjem saradničkog odnosa između projektanata informacionog sistema i Bell Atlantic-ovih poslovnih menadžera, SSNS tim je radio na pažljivom definisanju problema. Korisnici su ukazali na probleme i prisilili projektante informacionog sistema da rešavaju ključna pitanja. Članovi projektnog tima informacionog sistema su ohrabrivali akcije korisnika i pažljivo slušali njihove savete.

Pitanja performanse su rešavana tako što su korisnici radili ramc uz ramc sa softverskim inženjerima. Kada je sistem bio završen, neki od korisnika su obučeni da budu promoteri sistema koji su objašnjavali tehnologiju i obučavali druge. To međusobno poverenje je pomoglo u prenosu tehnologije, a korisnici su bili željni da usavrše veštine potrebne za korišćenje SSNS-a.

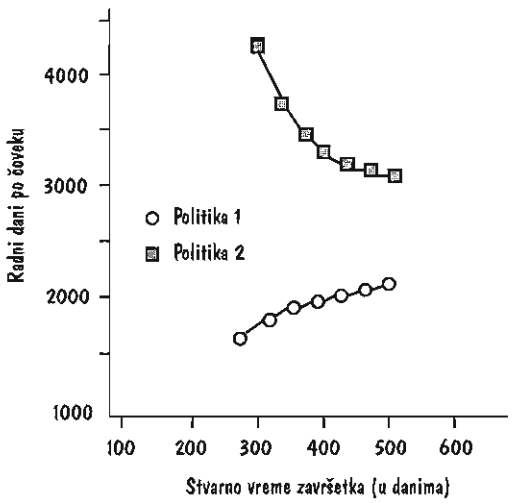
Abdel-Hamid (1990) koristi modele dinamike sistema u istraživanju uticaja skraćenja rokova realizacije. On zapaža da je definisanje rokova rada na projektu kontinualan proces; menadžer projekta vrši reviziju kako sve više saznaje o projektu. Dakle, konačni troškovi i vreme završetka zavise od realističnosti početne procene, a takođe i od toga kako su resursi podešavani da bi se usaglasili sa početnom procenom.

Abdel-Hamid (1990) je primenio svoje modele na podatke iz Nasinog centra za svemisrke letove Goddard, posmatrajući efekte politika upravljanja na troškove projekta i vreme završetka. Na primer, na slici 13.11 prikazan je efekat dve različite hipotetičke politike na broj radnih dana po čoveku potreban za završetak projekta. Prva politika, predstavljena kružićima, pretpostavlja da će upravljanje uvek podesiti radnu snagu na nivo potreban da bi se projekat odvijao u skladu sa planom rada. Druga politika, predstavljena kvadratićima, eliminiše dejstvo maksimalno tolerantnog datuma završetka. Odnosno, teškoće u planu realizacije se rešavaju produžavanjem rokova, a ne podešavanjem nivoa radne snage. Razmatrana ponašanja su veoma različita, a Abdel-Hamidovi modeli pomažu menadžerima da odluče koje politike da primene.

13.5 OPŠTE SMERNICE ZA POBOLJŠANJE

Da bi ostale uspešne, organizacije bi trebalo da budu fleksibilne i da se razvijaju. Takvi bi trebalo da budu i njihovi tehnološki problemi, bilo da se one usredsređuju na višekratnu upotrebljivost, merenja, inspekcije ili bilo koji drugi aspekt softverskog inženjerstva ili upravljanja. Kao i sve drugo što je važno za kompaniju, program zasnovan na tehnologiji zahteva strategijsko planiranje. Plan bi trebalo da se bavi ne samo time kako će se upotrebiti tehnologija, nego i kako će ona unaprediti proizvode, procese i resurse organizacije.

SLIKA 13.11 Kompromis između radnih dana po čoveku i rokova za dve politike upravljanja



Imajući u vidu da se stvari menjaju u toku vremena, trebalo bi izvršavati periodičnu reviziju strategijskog plana. Menadžeri i projektanti bi trebalo da daju odgovore na sledeća pitanja:

- *Da li su ciljevi ostali isti?* Ako su se poslovni ciljevi promenili, onda možda i ciljevi tehnološkog programa treba da se promene. Na primer, početni ciljevi mogu da obuhvate uspostavljanje bazne linije organizacije ili kompanije. Jednom kada se to postigne, ciljevi koji se odnose na povećanu produktivnost mogu da ih zamene. Slično tome, jednom kada se produktivnost poveća, kompanija ili sektor može da poželi da se usredsredi na poboljšavanje kvaliteta.
- *Da li su prioriteti ciljeva ostali isti?* Kada se jedna vrsta poboljšanja uspešno implementira, mogu da se izaberu druge vrste za sledeću inicijativu. Na primer, početni visoki prioriteti mogu da se dodele ciljevima koji obuhvataju upotrebu alata za testiranje. Međutim, jednom kada alat za testiranje postane deo zajedničke kulture razvoja, zahtevi i aktivnosti dizajna mogu da budu stavljeni u centar pažnje sa aspekta razumevanja i poboljšanja.
- *Da li su pitanja ostala ista?* Pitanja koja su bitna za prve stadijume programa (odnosno, koliko bi trebalo da uložimo u tehniku?) mogu da se zamene pitanjima zrelosti (odnosno, koliko novca i vremena štedimo upotrebom tehnike?). Slično tome, prenos tehnologije koja se implementira i počinjanje pilot projektima može na kraju da otvori nova pitanja (odnosno, koji troškovi iz pilot programa će verovatno ponovo nastati kada se tehnologija uvede u celu kompaniju?).
- *Da li su merenja ostala ista?* Kako proces razvoja ili održavanja sazreva, raste bogatstvo merenja potrebnih da se oni shvate i kontrolišu. U isto vreme, prikupljanje nekih mera postaje nepotrebno, pa se one mogu zameniti novim. Na primer, početno prikupljanje mera veličine može da se zameni nekom vrstom mere funkcionalnosti. Slično

tome, kako višekratna upotrebljivost postaje široko usvojena praksa u korporaciji, mere pouzdanosti, raspoloživosti i mogućnosti održavanja mogu da se prikupljaju da bi se utvrdio uticaj višekratne upotrebljivosti na kvalitet zajedničkog proizvoda. Mere koje se odnose na zadovoljstvo kupca mogu takođe da rastu, od mera broja zahtevanih i iskorišćenih usluga, do mera kvaliteta intefejsa i kvaliteta podrške kupcu.

- *Da li je zrelost ostala ista?* Zrelost procesa razvoja ili održavanja može da se poboljša, a sa njom vidljivost potrebna da se razumeju i mere novi činioci. Na primer, početni pokušaji prikupljanja podataka mogu biti ručni, a podaci mogu da se evidentiraju u sklopu jednostavnih proračunskih tabela. Ali, kako razvoj raste, a sa njim i veličina baze podataka metrika, može da se razvije automatizovan sistem za podršku evidentiranju rezultata merenja. Sa automatizacijom dolazi finiji nivo granularnosti, kako se merenja ponavljaju u toku vremena i prati napredak. Kako zrelost raste, tako strategijski plan može da se obraća pitanjima na detaljnijem nivou.
- *Da li je auditorijum ostao isti?* Mnogi tehnološki programi počinju kao mali, često kao pilot programi u odeljenju ili sektoru, i proširuju se na korporaciju, sporo i pažljivo. Čineći to, auditorijum koji treba da shvati njihove efekte se menja, od programera, menadžera i šefova odeljenja, do direktora sektora i korporativnih izvršnih direktora. Dakle, auditorijum se menja dejstvom tehnologije. Za strategijski plan je važno da se takođe menja da bi odražavao pitanja i interese stvarnog auditorijuma.

13.6 PRIMER IZ INFORMACIONIH SISTEMA

U celoj ovoj knjizi, učili smo o sistemu Piccadilly za prodaju vremena oglašavanja. Pretpostavite da sistem dobro radi i da većina promena u održavanju odražava potrebe Piccadillyja kada se kampanje oglašivači menjaju radi ostvarenja poslovnih ciljeva. Kakve strategije poboljšanja bi trebalo da slede serviseri Piccadillyja kako bi mogli brzo da implementiraju promene, bez unošenja grešaka u softver?

Jedna od ključnih strategija je da se izvede usavršavajuće održavanje. Serviseri mogu da testiraju dizajn softvera da bi videli može li da se učini fleksibilnijim i pogodnijim za menjanje. Koristeći istoriju promena u prošlosti, oni mogu da identifikuju komponente na koje promena najverovatnije može da utiče i pogledaju koliko vremena su ranije promene zahtevale.

Druga strategija je da se testiraju softverski sistemi slični Piccadillyju. U ovom poglavlju kao i u prethodnim poglavljima videli smo da su kompanije, kao što su Bell Atlantic i Chase Manhattan, zamenile više nasleđenih sistema jednim većim i sveobuhvatnijim. Analitičari Piccadillyja mogu da preduzmu širi, sistemski pristup. Kako sistem oglašavanja podržava poslovanje? Koji drugi softverski sistemi međusobno interagiraju sa njim? Kako sistemi mogu da se kombinuju ili pojačaju da bi brže ili efektivnije odgovorili na potrebe poslovanja? Drugim rečima, analitičari Piccadillyja bi trebalo da ispitaju granice sistema i odrede da li bi ih tebalo proširiti da bi obuhvatile i druge probleme.

13.7 PRIMER IZ PRAKSE

Strategije poboljšanja su važne i u Evropskoj svemirskoj agenciji. U izveštaju Lionsa i dr. (1996), predlaže se više poboljšanja, uključujući i sledeća:

- Tim bi trebalo da izvrši iscrpan pregled zahteva da bi identifikovao oblasti gde su se zahtevi za Arianu 5 značajno razlikovali od Ariane 4. Posebno, specifikacija bi trebalo da sadrži podatke o putanji Ariane 5 kao funkcionalnom zahtevu.
- Tim bi trebalo da izvrši ispitivanja na zemlji unošenjem simuliranih signala ubrzanja u predviđenim parametrima leta, koristeći obrtni sto za simulaciju ugaonog kretanja lansera.
- Preciznost sistema vođenja bi trebalo da bude pokazana analizom i računarskom simulacijom.
- Pregledi bi trebalo da postanu deo procesa projektovanja i kvalifikacije, koji se izvodi na svim nivoima i uključuje spoljašnje eksperte i glavne partnere u projektu.

Ti koraci bi trebalo da se preduzmu kao deo regresione vežbe, kad god se kôd promeni. Oni bi trebalo da pomognu da načini otkaza koji nisu detektovani u prošlosti, ne budu propušteni u budućnosti.

13.8 ŠTA OVO POGLAVLJE ZNAČI ZA VAS

U ovom poglavlju, ispitali smo više tehnika za poboljšavanje predviđanja, proizvoda, procesa i resursa. Videli smo kako predviđanja mogu da se poboljšaju korišćenjem *u*-krivih, verovatnoće *prethodne jednakosti* i rekalibracije da bi se smanjili šum i polarizacija. Proizvodi mogu da se poboljšaju kao deo programa višekratne upotrebljivosti , ili uspostavljanjem procesa inspekcija. Procesi mogu da se poboljšaju procenjivanjem njihovih efekata i određivanjem odnosa koji dovode do povećanog kvaliteta ili produktivnosti. Na primer, na osnovu istorijskih podataka mogu da se razviju modeli za predviđanje trenutka otkaza komponente; ta tehnika smanjuje vreme održavanja sistema i na kraju dovodi do softvera visokog kvaliteta. Slično tome, okviri zrelosti procesa mogu da pomognu organizacijama u implementiranju aktivnosti koje će verovatno poboljšati kvalitet softvera, ali pažljivo vođene studije za to još uvek nisu dale dovoljno dobar dokaz. Najzad, kako saznajemo više o različitosti ljudi i odnosima između uloženog rada i plana rada, poboljšanja pri dodeljivanju resursa više obećavaju.

Ovi rezultati direktno utiču na projektante ili servisere. Da biste poboljšali svoju okolinu i proizvode, morate biti voljni da učestvujete u studijama slučaja i eksperimentima i pružite povratnu informaciju onima koji pokušavaju da utvrde uzroke poboljšanja. Morate blisko sarađivati sa svojim kupcima i korisnicima da biste izgradili poverenje, tako da oni veruju sistemu koji za njih gradite. Morate raditi kao deo tima, pronalazeći zajedničko utemeljenje kada tražite rešenja za probleme.

13.9 ŠTA OVO POGLAVLJE ZNAČI ZA RAZVOJNI TIM

Rezultati opisani u ovom poglavlju nagoveštavaju duboke promene u radu projektnog tima. Dobra predviđanja zavise od zajedničkog shvatanja ključnih pitanja koja utiču na rad i raspored rada. Dobri proizvodi i efektivni procesi zavise od načina na koji tim

funkcioniše zajedno sa ciljem okončanja posla. Dobri resursi su od suštinske važnosti za uspešnu realizaciju posla.

Više ovde pomenutih činilaca može da utiče na vaš tim na nepredvidljive načine. Ako su okviri zrelosti procesa zaista efektivni u poboljšanju kvaliteta softvera, onda bi mnoge od njihovih preporučenih praksi mogle da se primene u vašem timu i organizaciji. Teško je odrediti dozvoljeni stepen fleksibilnosti pre nego što se dogodi gubitak nivoa zrelosti. Slično tome, ako „kompaktni" tim nastavi da radi zajedno i na drugim projektima, mogli biste napraviti bolje proizvode, ali sa manje prilika da radite sa novim ljudima na potpuno novim problemima.

S druge strane, studije pomenute u ovom poglavlju, ističu potrebu da timovi jedni druge proveravaju. Inspekcije, čista soba, višekratna upotrebljivost i drugi procesi u vezi sa kvalitetom obuhvataju pažljivo detaljno ispitivanje rada pojedinca ili organizacije od strane nekog drugog. Ti pristupi podstiču depersonalizovan razvoj, gde su u žiži kvalitet proizvoda i efektivnost procesa, a ne pojedinačna dostignuća.

13.10 ŠTA OVO POGLAVLJE ZNAČI ZA ISTRAŽIVAČE

Istraživanje pitanja poboljšanja narasta kada projektanti traže empirijski dokaz da su predložene tehnologije stvarno delotvorne. U ovom poglavlju ilustrujemo potrebu za više anketa, studija slučaja i eksperimenata; primer Basilija i Greena pokazuje kako je moguće sprovesti više studija koje se oslanjaju jedna na drugu.

Potrebno je i istraživanje okvira za poboljšanje, zasnovanih na dokazanim tehnikama i skrojenih za posebne primene i domene. Opšta ideja zrelosti je rezultovala nizom odgovarajućih okvira, kao što su model zrelosti višekratne upotrebljivosti , model zrelosti CASE alata itd. Ti okviri moraju da se integrišu da bi praktičari imali bolji osećaj koju od tehnologija da usvoje i zašto.

Najzad, istraživači softverskog inženjerstva bi trebalo da kroče obema nogama u oblast istraživanja ljudskog faktora. Učenjem iz već obavljenih studija u društvenim naukama, možemo bolje da razumemo pitanja resursa kao što su veličina tima, stilovi saradnje i šta je to što stvara dobar radni ambijent. Onda možemo da procenimo koji rezultati su primenljivi na softverske inženjere. Većina istraživača priznaje da je različitost ljudi ključni činilac u određivanju da li će ciljevi kvaliteta i plana rada biti ispunjeni; bolje razumevanje te različitosti pomaže pri projektovanju tehnika i alata koji tu raznolikost stavljaju u funkciju koristi.

13.11 SEMINARSKI RAD

Razmotrite kako ste vi i vaš tim gradili projekat Organizator pozajmica. Kakva poboljšanja u domenu resursa bi bila potrebna da biste bolje uradili posao? Obuka? Drukčije veštine? Veća tišina na radnom mestu? Napravite spisak u kome svom nastavniku predlažete bolje načine organizacije projekta pri ponovnoj dodeli u sklopu kursa.

13.12 KLJUČNE REFERENCE

Sertifikacija ISO 9000 se zahteva u mnogim zemljama. Seddonova procena i komentari o standardu ISO 9000 mogu da se pronađu na http://www.vanguardconsult.co.uk. Veb lokacija http://www.avnet.co.uk/SQM/QiC/news/ASA.html sadrži opsežnu diskusiju o Seddonovim nalazima i svedočanstvo o standardu ISO 9000.

DeMarco i Lister nastavljaju da drže ažurirane seminare o pitanjima koja su prvi put postavljena u njihovoj knjizi „Ljudski resursi"(*Peopleware*), koju su nedavno dopunili i ponovo izdali. Informacije o seminarima i materijalima u vezi sa njima mogu da se pronađu na veb lokaciji Atlantic Systems Guild, http://www.systemsguild.com.

Procena i poboljšanje su predmet mnogih konferencija i simpozijuma. Konferenciju Empirical Assessment of Software Engineering organizuje Keele University i ona privlači istraživače zainteresovane za izvođenje studija slučaja i eksperimenata. Više informacija može da se dobije slanjem poruka na adresu Barbara.Kitchenham@cs.keele.ac.uk. Američki Institut za softversko inženjerstvo i Evropska fondacija za unapređenje softverskog procesa organizuju radionice na temu modela zrelosti. Časopis *Software Process - Improvement and Practice* uključuje više članaka o efektima modela zrelosti. El Emam i njegove kolege u National Research Council of Canada nastavljaju da procenjuju efekte različitih modela zrelosti, kao i probleme sporazuma i validnosti mernih instrumenata.

13.13 VEŽBE

1. Pretpostavite da pratite gustinu grešaka u nizu sličnih proizvoda, tako da možete da nadgledate efektivnost novog procesa inspekcije koji ste uveli. U toku vremena, otkrivate da se gustina grešaka smanjuje. Objasnite kako biste mogli da odredite da li je opadanje gustine grešaka rezultat inspekcija, povećanog razumevanja proizvoda ili nepažljivih aktivnosti prilikom inspekcije i razvoja.
2. Abdel-Hamidov model dinamike sistema uzima u obzir promene u razumevanju projekta, kako projekat napreduje. Koji su razlozi za usvajanje pretpostavki o tim promenama i protiv njega? Kako možemo da testiramo naše pretpostavke, tako da imamo više poverenja u rezultate modela?
3. Objasnite kako dinamika sistema može da se upotrebi za ispitivanje kompromisa između odgovarajuće sigurnosti računara i prihvatljive performanse sistema.
4. Kakve su opasnosti od pretpostavljanja da se iskustvo SEL-a sa čistom sobom može primeniti na vašu organizaciju?
5. Pretpostavite da vaša organizacija razmatra brzu izradu prototipa na svom sledećem projektu. Gordon i Bieman (1995) su stavili u katalog lekcije koje su naučili o brzoj izradi prototipa iz izveštaja u literaturi, a ti rezultati nisu uvek jasni. Kako biste projektovali proces uvođenja brze izrade prototipa, procenili njegovu efektivnost i poboljšali njegovu upotrebu?
6. Predsednik vaše kompanije je naučila o ISO 9000 i insistira da kompanija dobije sertifikat. Ona želi kvantitativni dokaz da je ISO 9000 poboljšao procese i proizvode u kompaniji. Kako biste merili efekat sertifikacije ISO 9000?
7. Da li bi softverski inženjeri trebalo da dobiju licence ili sertifikate, na način kao što to rade ostali profesionalni inženjeri? Da li performansa softverskog inženjera može da se proceni na objektivan, kvantitativan način? U kakvom su odnosu pitanja davanja licence sa pitanjima koja su uvedena u ovom poglavlju?

14 Budućnost softverskog inženjerstva



U ovom poglavlju obradivaćemo:
- Wassermanove principe i šta smo do sada činili;
- prenos tehnologije;
- kako istraživači obezbeđuju dokaz usvajanja tehnologije;
- donošenje odluka u softverskom inženjerstvu;
- sledeće korake u istraživanju i praksi.

U ovoj knjizi detaljno smo ispitali mnogo aktivnosti u vezi sa inženjerstvom kvalitetnog softvera. Videli smo kako se koriste dobro definisani procesi da bi se izgradio softver koji zadovoljava potrebe kupaca, poboljšavajući njihovo poslovanje i živote. Ali, da bismo razumeli kuda smo se uputili, treba da pogledamo unazad na ono što smo postigli u relativno kratkom vremenskom periodu.

14.1 KAKO SMO RADILI

Od onog trenutka 1968. godine, kada je izraz „softversko inženjerstvo" prvi put upotrebljen na konferenciji NATO-a, pa do danas, kada mnoge kompanije shvataju da njihovi proizvodi i usluge zavise od izgradnje ili upotrebe softvera, izgradili smo korpus podataka, anegdota, teorija i praksi koji je promenio živote skoro svih ljudi. Zbog izuzetne zavisnosti sveta od softvera, kao studenti, praktičari i istraživači, obavezni smo da delujemo odgovorno i efikasno. Deo naše odgovornosti je da postignemo bolje razumevanje onoga što radimo i zašto to radimo, a onda da to znanje iskoristimo za poboljšanje naše prakse i proizvoda.

Preduzeli smo značajne korake ka tom ogromnom cilju poboljšanja. Počevši kao prenosioci bitova i bajtova, sada koristimo složene programske jezike za upravljanje našim

digitalnim sistemima. Gnjavili smo svoje šablone i apstrakcije da bismo uobličili ponovo upotrebljive proizvode i stvorili nove pristupe projektovanju. Primenili smo formalne metode na teške, neformalno iskazane probleme, da bismo pomogli da njihove složenosti i sukobi postanu vidljiviji. Izgradili smo ogroman skup alata za ubrzanje rutinskih poslova, kodiranje i kategorizaciju odnosa, praćenje napretka i simuliranje različitih mogućnosti.

Razvili smo mudrost skrivanja detalja kada oni zamagljuju suštinu problema (kao u objektno orijentisanom razvoju), ili njihovog ekspliciranja kada su neophodni za potpunije razumevanje (kao kod testiranja po modelu providne kutije). Ali, pred nama su još uvek izazovi. Težimo da obezbedimo veliku tačnost na širokom prostoru: možemo da kažemo kada će svemirsko vozilo stići na Mars, ili kada će hemijska reakcija dostići kritičnu tačku. Ali, ne posedujemo tačnost u malom: ne možemo precizno da kažemo kada će softverski proizvod ponovo otkazati, ili sa kojom tačnošću korisnik izvršava funkcije sistema.

Wassermanovi koraci ka zrelosti

Ovu knjigu smo započeli predstavljanjem osam Wassermanovih koraka ka zrelijoj disciplini softverskog inženjerstva. Pošto smo u dosadašnjim poglavljima dublje ispitali savremene prakse softverskog inženjerstva, vraćamo se Wassermanovim stavovima kao planu za budući rad:

- *Apstrakcija.* Videli smo kako nam apstrakcija pomaže da se usredsredimo na suštinu problema i kako se ona razlikuje od transformacije. Moramo da nastavimo sa upotrebom apstrakcija da bismo pronašli šablone ne samo u dizajnu i kodu, nego i u zahtevima, radnim navikama, željama korisnika, test slučajevima i strategijama, kao i u opštem načinu na koji pristupamo problemu i pokušavamo da ga rešimo. Apstrakcija može da predstavlja osnov za nove načine učenja, podučavanja i rešavanja problema.
- *Metode i notacije analize i dizajna.* I dalje koristimo širok skup različitih metoda i notacija za predstavljanje problema i rešenja. Svako od nas ima svoje posebne najpoželjnije izbore, na koje utiču načini na koje shvatamo i učimo. Neki više vole slikovne predstave, dok drugi prednost daju tekstu. Multimedijalni softver nas takođe podstiče da stvari predstavljamo koristeći boju, zvuk, položaj ili druge karakteristike. Udobnost i poželjniji izbor su u suprotnosti sa ciljem da se dođe do zajedničke notacije, razumljive za sve. Rešenje nije da se sve napusti zbog jednog metoda i notacije. Naš cilj će pre biti da razvijemo transfomacije iz različitih predstava i metoda u jednu zajedničku, koja će biti korisna za raspravu i arhiviranje. Baš kao što Evropska zajednica čuva sopstvene jezike, ali koristi engleski za zajedničku komunikaciju i razumevanje, mi tako možemo da pronađemo zajednički način za iskazivanje zahteva i projekata, dok istovremeno čuvamo vrednost alternativnih izražavanja.
- *Izrada prototipova korisničkog interfejsa.* Kako unosimo softver u kritične oblasti naših života, uloga korisnika postaje sve važnija. Moramo da naučimo kako korisnik razmišlja o problemima i izvodi rešenja, tako da naš softver podržava i podstiče njegovo odgovarajuće ponašanje. Videli smo primere softvera koji ometa korisnike u obavljanju njihovih poslova, ili onemogućava poslovnom sistemu da kreira nove proizvode i usluge. Usredsređujući se na potrebe korisnika i poslovnih sistema, možemo da izradimo upotrebljivije proizvode sa značajno boljim odzivom.

- *Arhitektura softvera.* Shaw i Garlan su nam pokazali kako razne arhitekture odražavaju različita rešenja istog problema. Svako arhitektonsko rešenje ima argumente za i protiv, a mi moramo dopustiti da željene karakteristike rešenja opredeljuju izbor. Videli smo takođe kako je identifikacija stila i arhitektonskih šablona još uvek u povoju. Moramo da proširimo studiju arhitektura, da bismo bolje razumeli šablone, komponente i značenje stila.
- *Softverski proces.* Nema nikakve sumnje da softverski proces utiče na kvalitet softvera ako proces postaje transparentniji i lakši za upravljanje. Način na koji transparentnost i lakoća upravljanja utiču na kvalitet, predstavlja predmet za dalje proučavanje. U poglavlju 1 smo videli da je razvoj softvera jednako umetnost kao i nauka, pre oblik stvaranja i kompozicije nego proizvodnja. Moramo da naučimo kako da koristimo softverski proces da bismo poboljšali proizvod bez ograničavanja kreativnosti i fleksibilnosti. Moramo takođe da naučimo koji procesi najbolje funkcionišu u kojim situacijama i da shvatimo koje su karakteristike proizvoda i ljudi koji ih grade najvažnije prilikom izbora procesa.
- *Ponovno korišćenje.* U prošlosti, zajednica ponovnog korišćenja se usredsređivala na ponovno korišćenje koda iz ranijih aplikacija i izgradnju novog koda koji može ponovo da se koristi u različitim proizvodima, a ne samo u jednom. U budućnosti, moramo da proširimo horizonte, tražeći prilike za ponovno korišćenje u celokupnom procesu razvoja i održavanja. Više istraživača vidi održavanje kao primer ponovnog korišćenja, a takvo gledište može da nam pomogne da ponovo koristimo svoje iskustvo, jednako kao i svoje proizvode. Moramo da kombinujemo prošireno razumevanje apstrakcije i arhitekture sa potrebom da ponovo koristimo sve što možemo, tako da identifikujemo mnogo širi skup prilika za ponovno korišćenje. U isto vreme, treba da razvijemo tehnike procene koje pomažu u razumevanju kvaliteta komponente ili dokumenta; želimo da budemo u stanju da ponovo upotrebimo neki produkt sa izvesnim stepenom poverenja u to da će on raditi prema očekivanjima. Pitanja ponovnog korišćenja se odnose ne samo na produkte procesa razvoja i održavanja u okviru sopstvene firme, nego i na komercijalne gotove proizvode koje planiramo da integrišemo u naš sledeći softverski sistem.
- *Merenje.* Gotovo sve aktivnosti opisane u ovoj knjizi na neki način obuhvataju merenje. Treba da znamo da li naši proizvodi zadovoljavaju kriterijume kvaliteta i nadamo se da je naša praksa efektivna i efikasna. U budućnosti, moramo da merimo ključne karakteristike proizvoda, procesa i resursa, na načine koji su nenametljivi, korisni i pravovremeni. Trebalo bi da merimo produktivnost koristeći karakteristike koje su različite od veličine koda, potvrđujući da smo produktivni znatno pre početka pisanja koda. Trebalo bi da merimo kvalitet u skladu sa širim okvirom i da, pored grešaka i otkaza, uključimo: zadovoljstvo kupca, potrebu poslovanja i druge činioce. Trebalo bi da merimo i zadovoljstvo softverskog inženjera, obezbeđujući da naši poslovi nastavljaju da budu zadovoljavajući kako pronalazimo sve bolje načine izrade novih i boljih proizvoda.
- *Alati i integrisana okruženja.* Projektanti su mnogo godina tražili alate i okruženja koji bi ih učinili produktivnijim i efektivnijim. Sada, pošto su uložili milione dolara u alate koji nisu ispunili obećanja, projektanti grade i koriste alate sa mnogo realnijim očekivanjima. Alati i okruženja mogu da nam pomognu u automatizaciji poslova koji se ponavljaju, mogu da proračune učine nenametljivim i da prate odnose. U

budućnosti, moramo da tražimo alate koji nam pomažu da pratimo veze između proizvoda, tako da možemo da izvršimo analizu uticaja predloženih izmena. Moramo da izgradimo alate za merenje koji merenja obavljaju u pozadini i obezbeđuju brzu povratnu spregu projektantima i menadžerima kojima je potrebna informacija o proizvodima i napretku. Potrebni su nam alati da pomognu da simuliramo i shvatimo delove problema, moguće interfejse i arhitekture, kao i implikacije izbora jedne strategije rešenja u odnosu na drugu. Moramo da koristimo alate koji podržavaju ponovno korišćenje, tako da možemo lako da izdvojimo ono što nam je potrebno iz ranijih projekata i ugradimo ga u proizvode na kojima trenutno radimo.

Šta sada?

Ovaj pregled Wassermanovih principa otvara najmanje dva važna pitanja kojima bi, kao istraživači i praktičari, trebalo da se bavimo. Prvo, trebalo bi da razmotrimo koliko dobro smo ideje o novom softverskom inženjerstvu preveli u praksu. Odnosno, kada istraživači imaju dobru ideju koja obećava bolje softverske procese, resurse ili proizvode, da li dobro radimo kada tu ideju pretvaramo u tehnologiju koja je praktično primenljiva i efektivna? Poznavajući naš zapis vezan za praćenje prenosa tehnologije, kako možemo da je u budućnosti poboljšamo?

Drugo, moramo da razmotrimo kako naše istraživanje i praksa podržavaju donošenje odluka o procesima, resursima i proizvodima. Svako poglavlje u ovoj knjizi sadrži primere potrebnih odluka u toku razvoja softvera i održavanja. Na primer,

- Koji softverski proces bi trebalo da se upotrebi?
- Koliko će trajati razvoj i koliko ljudi će biti potrebno?
- Koliko toga bi trebalo ponovo da upotrebimo?
- Kakvu vrstu testiranja bi trebalo da izvršimo?
- Kako upravljamo rizicima?

Kako da poboljšamo donošenje odluka, tako da imamo više poverenja u izbore tehnologije i resursa? U ostatku ovog poglavlja analizira se ono što znamo i ono do čega bismo tek u budućnosti mogli da dođemo.

14.2 PRENOS TEHNOLOGIJE

Pretpostavite da upravo krećete sa projektom razvoja novog ili održavanja postojećeg softvera. Da li koristite poznatu tehnologiju koja je proverena u dužem vremenu, ili ugrađujete novu koja mnogo obećava? Da li tražite empirijski dokaz da nova tehnologija radi, ili sledite svoj dobar osećaj, savet kolega ili sugestije prodavca? Vaš izbor verovatno zavisi od toga da li sebe vidite kao proizvođača ili potrošača tehnologije, a takođe i da li težite da se usredsredite na problem koji treba da se reši, ili na njegovo rešenje. Na primer, projektanti firme IBM su zanemarivali „vruće" aktuelne tehnologije i umesto toga izgradili spejs šatl sa oprobanom tehnologijom IBM 360, da bi minimizovali rizik otkaza. Radeći na taj način, IBM je delovao kao potrošač tehnologije, usredsređujući se na stabilnost rešenja, a ne na intrigirajuću novu tehnologiju. Ali kompanije kao što su Silicon Graphics i Tandem Computers, kao proizvođači tehnologije, napravile su sopstvene tehnologije

specijalne namene, a nisu zasnovale svoja rešenja na postojećim tehnologijama. Kada govorimo o **prenosu tehnologije**, mislimo i da proizvođači tehnologije stvaraju i koriste nove tehnologije a da ih potrošači tehnologije usvajaju i koriste u novim proizvodima i uslugama.

Bez obzira na to šta i kako izabrali, možemo da pronađemo uspešne ranije primere sopstvenih strategija izbora tehnologije sa ciljem da podržimo svoja razmišljanja. Nažalost, kao što smo videli u prethodnim poglavljima, možemo takođe da pronađemo i primere ranijih neuspeha. Ta zagonetka uspeh/neuspeh ima najmanje dva lica: tehničko i komercijalno. Možete da napravite čuda rešavajući problem, samo da biste na kraju otkrili da vaš projekat nema komercijalni uspeh. Ili, da biste iskoristili priliku, možete da jurnete sa svojim proizvodom na tržište, da biste tako postigli kratkoročni komercijalni uspeh, ali da propadnete na duge staze, zbog nedostatka podrške i unapređenja. Često ste usredsređeni na dva cilja: želite da izaberete pravu tehnologiju da biste rešili problem i da to učinite na način koji se dopada kupcima. U ovom odeljku, ispitujemo zašto i kako donosimo odluke vezane za izbor tehnologije. Razmatramo takođe i kako empirijsko softversko inženjerstvo, opisano u poglavlju 13, izvodi dokaz koji podržava te odluke. Da li dokaz pomaže ili otežava usvajanje novih tehnologija unutar organizacije ili dela tržišta?

U ovoj knjizi opisali smo više vrsta tehnologija koje su nam na raspolaganju. Kao što smo napomenuli u poglavlju 1, „tehnologija" koju biramo da nam pomogne u sledećem projektu može da predstavlja bilo koji metod, tehniku, alat, proceduru ili platformu razvoja ili održavanja softvera. Da bismo znali da li je tehnologija uspešna, posmatramo da li ona rešava problem, da li je komercijalno validna i koliko vremena je potrebno da postane prihvaćena u praksi. Takođe želimo da znamo kako da povećamo manje bitne razloge koji naš izbor tehnologije definišu kao ispravan.

Kako sada donosimo odluke o prenosu tehnologije

Sredinom osamdesetih godina Redwine i Riddle (1985) su opisali rezultate istraživanja brzine kojom je softverska tehnologija sazrevala i bila prihvatana u praksi. Oni su prikupili studije slučajeva koji su se odnosili na mnoge koncepte i tehnologije koje su početno bile razvijane u 1960-im i 1970-im godinama, uključujući formalnu verifikaciju, modele troškova i objektno orijentisane jezike, kao što je Smalltalk. Oni su otkrili da je „potrebno reda 15 do 20 godina da tehnologija sazre do tačke kada može da se popularizuje i na veliko širi u tehničkoj zajednici". (Redwine i Riddle 1985). U najgorem slučaju, bilo je potrebno 23 godine da se pređe put od formulacije koncepta do tačke u kojoj je mogla da se razmatra popularizacija; u najboljem slučaju, bilo je potrebno 11 godina, a srednje vreme je iznosilo 17 godina. Jednom kada se tehnologija razvila, bilo je potrebno u proseku 7,5 godina da ona postane široko raspoloživa.

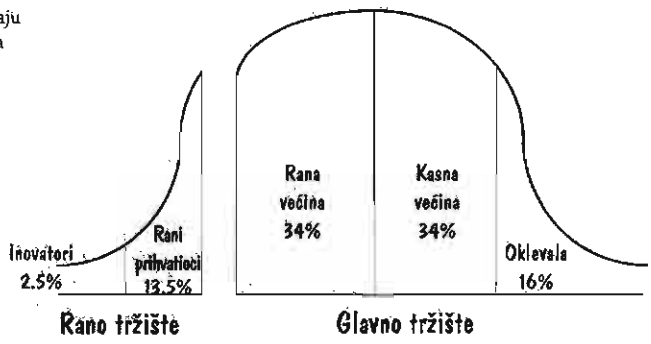
Jasno, 17 godina je suviše dug period čekanja u poslovanju, gde pritisci vezani za vreme izlaska na tržište zahtevaju da se nove tehnologije brzo same dokažu. Na primer, u 1977. godini, polovinu prihoda Hewlett-Packarda su ostvarili proizvodi koji su uvedeni u prethodne dve godine. Čak i ako svi nisu sadržavali novu tehnologiju, odnos između prihoda i brzine promene nam ukazuje da moramo nove tehnologije plasirati na tržište daleko brže nego što smo to činili u prošlosti. Tržišta ne mogu da čekaju deceniju ili dve na tehnološku inovaciju. Iz tog razloga, mnoge razvojne organizacije grabe nove tehno-

logije koje obećavaju daleko pre nego što se dođe do jasnog dokaza kojim bi korist bila potvrđena. Na primer, Capability Maturity Model američkog Instituta za softversko inženjerstvo (SEI) su prigrlile mnoge kompanije znatno pre nego što su SEI i ostali počeli empirijska istraživanja prirode i veličine njegovih efekata na poboljšanje procesa. Slično tome, mnogi razvojni timovi pišu kôd na Javi čak i za vreme razvoja Java standarda.

U drugim slučajevima, usvajanje tehnologije je obavezno kada izgleda da je tehnologija tako očigledno korisna da se ne smatra da je potrebna pažljiva empirijska procena. Nametanje jezika Ada kao standardnog jezika za programiranje od strane američkog Ministarstva odbrane je jedan od primera upotrebe standarda za podsticanje usvajanja tehnologije. Drugi je insistiranje britanskog Ministarstva odbrane na upotrebi formalnih metoda u razvoju bezbedno kritičnih sistema, čak i kada je bilo malo dokaza o prirodi i obimu koristi od primene formalnih metoda. Zaista, nema saglasnosti u zajednici softverskog inženjerstva o tome na šta mislimo kada kažemo da je nešto formalni metod, ali se pritisak za upotrebu takvih metoda svejedno nastavlja.

Upotreba dokaza u donošenju odluka o tehnologiji

Da bismo razumeli kako se stvarno donose odluke o tehnologiji, možemo da se okrenemo poslovnim školama i njihovim programima marketinga, usredsređujući se na ono što nam je poznato kao „istraživanje difuzije". Proučavajući prenos tehnologije u mnogim organizacijama (ne samo prenose koji se odnose na softver), Rogers (1995) napominje da postoje različiti obrasci u načinu i brzini kojom se tehnologija usvaja. Ljudi koji prvi usvajaju tehnologiju su **inovatori**; oni verovatno čine samo 2,5% od ukupne verovatne publike, kao što je prikazano na slici 14.1. Rogers objašnjava da su inovatori „skloni riziku": njih pokreće želja da jurišaju i učine nešto smelo. Inovator obično lansira novu ideju uvozeći je iz prostora izvan normalnih granica sistema. U tom smislu, inovator je čuvar kapije svoje organizacije. Inovatori su takođe pokretači i oni se oslanjaju na lične kontakte da bi ubedili svoje kolege da preuzmu rizik i isprobaju novu tehnologiju.



SLIKA 14.1 Vrste onih koji usvajaju i jaz između ranog i glavnog tržišta (Rogers 1995 i Moore 1991).

Rani prihvatioci takođe žure da prigrle novu tehnologiju. Međutim, ono što ih intrigira nije tehnologija sama po sebi. Njihov interes pokreće opažanje mogućih koristi koje nova tehnologija može da donese poslovanju. Rani prihvatioci obično puštaju da neko drugi pre njih „isproba vodu". Ali, kada pročitaju o uspehu tehnologije u cenjenoj publikaciji, oni grade na tuđem uspehu i uvode tehnologiju u sopstvene organizacije. Na primer, Software Technology Support Center u Hill Air Force Base (Ogden, Utah) procenjuje tehnologije i redovno pravi izveštaje o svojim nalazima; rani prihvatioci mogu da sačekaju da se u izveštajima STSC-a pojavi nešto što obećava i da zatim prihvate alate i tehnike koji zvuče privlačno. Donošenjem razumnih tehnoloških odluka, rani prihvatioci smanjuju neizvesnost o pogodnosti ili efektivnosti nove tehnologije; oni je usvajaju, dok istovremeno lično informišu kolege o njenim prethodnim ili sadašnjim uspesima.

Rana većina je oprezna u donošenju odluka, razmišljajući neko vreme pre nego što poželi dobrodošlicu novoj tehnologiji. Pokrenuti praktičnošću, članovi rane većine se obezbeđuju da inovacija nije prolazni hir. Drugim rečima, oni pre slede nego što predvode, ali su voljni da isprobaju nove stvari za koje su drugi pokazali da su uspešne. Usvajaoci rane većine mogu da se ubede da probaju tehnologiju, ne samo kada je ona uspela na drugom mestu, nego i kada je prate materijali (kao što su vodiči za obuku, funkcije pomoći i jednostavni interfejsi) koji usvajanje čine relativno glatkim i bezbolnim.

Kasna većina usvajalaca je skeptičnija. Usvajanje nove tehnologije je obično rezultat pritisaka, ekonomskih ili od strane konkurencije. Ono što je u vezi sa novom idejom neizvesno uglavnom mora da se razreši pre nego što se kasni usvajalac bude složio da je isproba. Kao rezultat, kasna većina će čekati da se nova tehnologija stabilizuje i da na raspolaganju postoji dovoljna količina podrške. S obzirom na to da prihvatioci kasne većine ne vole neizvesnost, oni nalaze da je posebno privlačno osloniti se na savet prodavca. Na taj način, prodavac može da koristi iskustva drugih kupaca da bi ubedio kasnu većinu da će tehnologija funkcionisati.

Najzad, oklevala su često protivnici usvajanja nečeg novog, bilo iz ekonomskog ili ličnog opravdanja. Oni uskaču u tehnološki voz samo kada su sigurni da nova ideja neće propašti, ili kad su prisiljeni na promenu naređenjem rukovodstva ili kupca. Pravila koja nameće organizacija, komitet za standarde ili kupac mogu da podstaknu oklevala da koriste novu tehnologiju kada su drugi modeli otkazali.

Dakle, uspešan prenos tehnologije zahteva ne samo novu ideju, nego i publiku koja će da je prihvati sa posebnim stilom usvajanja. Moore (1991) je posmatrao prenos tehnologije sa marketinškog aspekta. On ističe jaz, prikazan na slici 14.1, između „ranog tržišta" koje zahteva malo dokaza i „glavnog tržišta" koje traži mnogo više. Članovi ranog tržišta su inovatori i rani prihvatioci koji su usredsređeni na tehnologiju i njene mogućnosti. Oni su voljni da preuzmu rizik da bi videli kako tehnologija radi. Moore nagoveštava da je rano tržište zainteresovanije za radikalnu promenu, dok je konzervativnije glavno tržište više za postepena poboljšanja postojećeg načina obavljanja poslova. Odnosno, rano tržište pravi velike skokove i promene u načinu rada, dok se glavnom tržištu sviđa tekući proces, ali želi da ga dotera da bi bio produktivniji.

Dokaz koji podržava odluke o tehnologiji

Istraživači često izvode istražne studije da bi podržali odluke u vezi sa tehnologijom. Pregled koji su napravili Zelkowitz, Wallace i Binkley (1998) baca svetlo na često veliku razliku između nameravane upotrebe tih studija i načina na koji na njih gledaju praktičari. Ispitujući 90 istraživača o vrednosti različitih eksperimentalnih metoda potrebnih da bi se vrednovala nova tehnologija, Zelkowitz i dr. su otkrili da praktičari najviše vrednuju metode koje su relevantne za njihovo okruženje. Odnosno, tehnike kao što su studije slučajeva, terenske studije i ponovljeni kontrolisani eksperimenti, smatrane su značajnim za proces donošenja odluka u izboru nove tehnologije.

S druge strane, istraživači radije izvode studije koje obuhvataju reproduktivne metode vrednovanja koje se mogu koristiti u izolaciji u laboratoriji, kao što su teoretski dokaz, statička analiza i simulacija. Oni su manje cenili metode koje su zahtevale direktnu interakciju sa praktičarima. Drugim rečima, većina istraživača je izbegavala baš one metode koje su najcenjenije među praktičarima: studije slučajeva, terenske studije i eksperimente. Dakle, istraživači pokušavaju da naprave sistem dokaza za procenu tehnologije, ali praktičari traže veoma različitu vrstu dokazivanja tehničke efektivnosti. Zato sistem dokaza koji su zasnovali istraživači verovatno neće biti ozbiljno shvaćen od strane praktičara koji razmišljaju o tome da upotrebe tehnologiju! Studija nam pokazuje da uspešan prenos tehnologije zahteva razumevanje ciljne publike i prikupljanje dokaza koji su za nju značajni i kredibilni.

Šta to znači i za empirijska istraživanja softverskog inženjerstva i za prenos softverske tehnologije u budućnosti? To svakako znači protraćen istraživački napor ako stvari ostanu kakve su danas. A što je još gore, to znači da će mnogi od nas donositi tehnološke odluke koje nisu zasnovane na bilo kakvom dokazu, na vrlo malo dokaza ili čak na neodgovarajućem dokazu.

Bliži pogled na dokaz

Prodavanje tehnoloških ideja zahteva efektivnu marketinšku strategiju zasnovanu na razumevanju publike. Kao što smo videli, publika može da ima jedan od pet različitih stavova prema tehnologiji, a istražívači bi trebalo da usvoje vrstu dokaza koju traži svaki od njih. Ali, šta znamo o objedinjavanju elemenata dokaza u ubedljiv argument? Na svu sreću, možemo da pogledamo kako se u pravnoj zajednici grade ubedljive zbirke dokaza. Schum (1994) detaljno opisuje analizu dokaza sa najmanje dve tačke gledišta: njegovog izvora i njegove uverljivosti. Na primer, možemo da svrstamo deo dokaza u jednu od sledećih pet kategorija:

- *Opipljiv dokaz:* Ta vrsta dokaza može direktno da se ispituje da bi se odredilo šta otkriva. Primeri: objekti, dokumenti, demonstracije, modeli, mape, grafici.
- *Dokaz svedočenja:* Tu vrstu dokaza obezbeđuje druga osoba, koja nam kaže kako je dokaz dobijen. Schum razlikuje tri vrste dokaza putem svedočenja: direktno posmatranje (druga osoba je učestvovala u događaju i posmatrala ili opažala direktno), iz druge ruke (ona ili on je dobio informaciju iz drugog izvora) i mišljenje (tu nema primarnog izvora informacije). Primeri: izveštaji o pregledu zahteva, izveštaji o prolaženju kroz projekat, članci u časopisima.
- *Dvosmislen dokaz svedočenja:* Onaj ko obezbeđuje dokaz ne seća ga se tačno, ili daje informaciju u formi verovatnoće. Primeri: operacionalni profili, procene pouzdanosti.

- *Dokaz koji nedostaje:* Očekivani dokaz ne može da se pronađe. Primeri: izveštaji od servisera sistema koji više nisu raspoloživi; projektanti sistema koji više nisu tu da bi izveli temeljnu analizu uzroka.
- *Prihvaćene činjenice:* Informacija koja je prihvaćena bez potrebe daljeg dokazivanja. Primeri: fizičke konstante, matematičke formule.

Dokaz može da bude pozitivan ili negativan. Odnosno, on može da obezbedi dokaz koristi od nove tehnologije, ili može da pokaže da nema nikakve koristi, ili čak da je rezultat negativan.

Verodostojnost dokaza se procenjuje posmatranjem i njegove autentičnosti i tačnosti. Na primer, verodostojnost dokaza putem svedočenja je u direktnom odnosu sa verodostojnošću onoga ko obezbeđuje dokaz. Sa svoje strane, verodostojnost svedoka zavisi od njegove istinitosti (da li verujemo u ono što nam svedok govori?), objektivnosti (da li na njegovo ili njeno opažanje utiču očekivanja ili drugi efekti) i osetljivosti posmatranja (koliko je dobra mera ili merni uređaj?).

Jednom kada smo odmerili svaki deo dokaza, možemo da kombinujemo različite delove da bismo videli da li oni, zajedno, podržavaju zaključak o tehnologiji. Ponekad je taj proces iterativan; dođemo do preliminarnog zaključka sa izvesnim stepenom poverenja, a zatim vršimo reviziju našeg zaključka i nivoa poverenja kada se stvori novi dokaz. Ako su delovi dokaza protivrečni, naš zaključak mora da uzme u obzir taj nesavršeni dokaz. Na primer, neke studije nagoveštavaju da objektna orijentacija podstiče ponovno korišćenje, dok ostale to ne čine. Možemo da dođemo do zaključka o vrednosti ponovnog korišćenja samo ispitivanjem prirode sukoba dokaza. U slučaju objektne orijentacije, sukob može da nam pomogne da razlikujemo trenutke kada će OO pomoći ponovno korišćenje, od trenutaka kada će joj naškoditi. Odnosno, pre nego da ukine zaključak, sukob može da pomogne da se on prečisti. Slično tome, veći dokazi nam pokazuju koje promenljive su najvažnije i koje aspekte treba da razmatramo kada projektujemo nove studije.

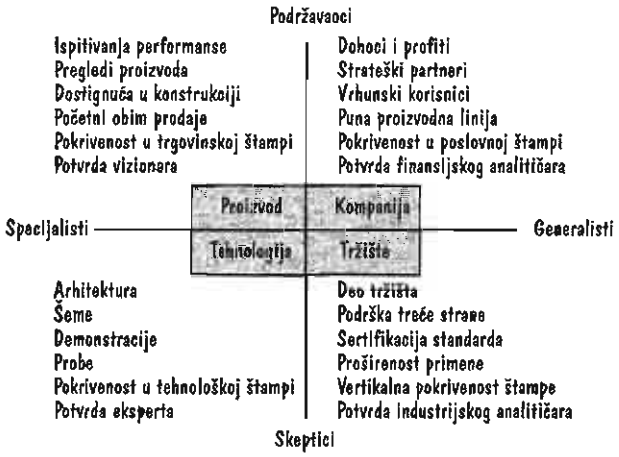
Dakle, može da postoji jaz u dokazu koji odražava Mooreov marketinški jaz: istraživači moraju da nauče da proizvode dokaz koji je koristan za praktičare. Dokaz mora da bude ne samo zanimljiv, nego i verodostojan i za istraživače i za praktičare. Ali, na koja pitanja bi dokaz trebalo da odgovori? Rogers, Schum i Moore daju signal o tome koja vrsta dokaza je pogodna za koga. Na primer, Rogers (1995) napominje da je relativna brzina kojom se usvaja nova tehnologija zasnovana na više važnih činilaca:

- *Priroda komunikacionih kanala koji se koriste da bi se povećala svest i znanje o tehnologiji.* Neke vrste usvajalaca nalaze da su neki komunikacioni kanali verodostojniji od drugih. Inovatori su skloni da se pouzdaju u tehničke časopise i izveštaje istraživača i tehničkih eksperata. Rani prihvatioci, sa svoje strane, pitaju inovatore da im potvrde verodostojnost tehnologije. Članovi rane i kasne većine pitaju druge u svom društvenom sistemu, često izgovorenom reči ili posetama koje za cilj imaju procenu performanse tehnologije.
- *Priroda društvenog sistema u kome radi potencijalni korisnik.* Inovatori se okreću tehnolozima, ali glavno tržište obično posmatra poslovnu zajednicu, da bi videlo šta rade konkurenti.
- *Količina napora potrebnog da bi se tehnologija raširila po organizaciji.* Rano tržište zahteva najmanji napor. Međutim, glavno tržište traži ne samo informaciju o samoj

tehnologiji, već i o pakovanju, podršci korisnicima i svedočenja trenutnih usvajalaca i korisnika. Moore nam kaže da je sama tehnologija „generički proizvod", ali prateći materijali pomažu da se uobliči „ceo proizvod".

- *Atributi tehnologije.* Rogers nagoveštava da bitni atributi obuhvataju:
  - ◊ *Relativnu prednost:* U kom stepenu je nova tehnologija bolja od one koja je već na raspolaganju? Inovatori na to pitanje odgovaraju u tehničkim terminima: da li je brža, arhitektonski nadmoćna, ili pogodnija za održavanje? Rani prihvatioci traže opipljiv dokaz o „celom proizvodu". Usvajaoci rane većine traže informacije o uticaju na kompaniju, dok prihvatioci kasne većine gledaju kako će tehnologija povećati njihov deo tržišta. Dakle, relativna prednost odražava prednost u četiri različite oblasti – tehnologiji, proizvodu, kompaniji i na tržištu – kada svaka vrsta potencijalnog usvajaoca proširuje svoj svet u kome vidi donošenje odluka.
  - ◊ *Kompatibilnost:* U kom stepenu je nova tehnologija konsistentna sa postojećim vrednostima, prethodnim iskustvima i potrebama potencijalnih usvajalaca? S obzirom na to da usvajaoce ranog tržišta zanima radikalna promena, oni nisu toliko zainteresovani za odgovor na to pitanje, kao oni sa glavnog tržišta. Odnosno, rano tržište nema ništa protiv ponovnog početka sa samog dna, ali korisnici sa glavnog tržišta nerado odustaju od nečega sa čime su već upoznati.
  - ◊ *Složenost:* U kom stepenu se nova tehnologija lako razume i koristi? Usvajaoci ranog tržišta će verovatnije biti voljni da uče potpuno nove načine na koje se nešto radi, čak i složenije, da bi zaronili u noviju, uzbudljiviju tehnologiju. Ali, ljudi sa glavnog tržišta vide lakoću upotrebe kao potrebnu karakteristiku nove tehnologije.
  - ◊ *Mogućnost probe:* Da li sa novom tehnologijom može da se eksperimentiše na ograničenoj osnovi? Za rano tržište, mogućnost da se tehnologija isproba je nešto obavezno. Ali mnogi ljudi sa glavnog tržišta će isprobati tehnologiju tek kada su neki drugi napravili solidan dosije o njenim prednostima.
  - ◊ *Mogućnost posmatranja:* Da li su rezultati korišćenja tehnologije vidljivi drugima? Rano tržište voli direktno posmatranje i opipljiv dokaz. Ali, korisnicima sa glavnog tržišta se više sviđa dokaz samo iz organizacija sličnih njihovoj sopstvenoj. Na primer, projektanti bankarskog softvera smatraju dokaz verodostojnim samo ako dolazi iz drugih bankarskih organizacija; studije slučajeva iz firmi koje se bave telekomunikacijama ili odbranom ih ostavljaju ravnodušnim. Štaviše, teškoće se pojavljuju kada direktno posmatrana tehnologija daje dokaz o komercijalnoj prednosti; u takvim slučajevima, u literaturi će se izveštavati samo o dokazima iz druge ruke, ako se to bude uopšte radilo.

Na slici 14.2 zbirno su date vrste dokaza koje privlače različitu publiku. Donji levi kvadrant odražava zanimanja inovatora, gde je pažnja posvećena prvenstveno samoj tehnologiji. Rani prihvatioci okreću svoju pažnju širem pojmu proizvoda, dok su pripadnici glavnog tržišta više zainteresovani za kompaniju i tržište. Ova slika jasno pokazuje da različita publika traži veoma različite vrste dokaza. Nalazi poslovnih škola, kao što je ovaj, pomažu nam da postavljamo prava pitanja o tehnologiji (to jest, pitanja čiji su odgovori zanimljivi za publiku) i projektujemo prave vrste studija da bismo pribavili dokaz koji odgovara na takva pitanja.



SLIKA 14.2 Dokaz koji zahteva različita publika (adaptirano iz Moorea 1991)

Novi modeli prenosa tehnologije

Glavni cilj usvajanja tehnologije je da se na neki način poboljša najmanje jedan novi proizvod, proces ili resurs. Dokaz bi trebalo da nam pomogne da odredimo da li nova tehnologija prouzrokuje poboljšanje. Želimo da istražimo odnos uzroka i efekta između nove tehnologije i jedne ili više promenljivih koje su važne. Čak i kada je korist ista kao kod postojećih ili konkurentnih tehnologija, možemo da izaberemo novu tehnologiju prosto zato što ona smanjuje neizvesnost u pogledu uzroka i efekta. Drugim rečima, želimo da rezultati procesa razvoja i održavanja budu predvidljiviji.

Sada možemo da sastavimo ove gradivne blokove uspešnosti prenosa tehnologije, da bismo bolje razumeli kako prenos tehnologije funkcioniše u pojedinačnim organizacijama. Takvi modeli obično uključuju preliminarnu procenu nove tehnologije, koju izvode oni koji je koriste po prvi put, da bi se videlo da li ona zaista rešava problem za koji je bila namenjena. Taj korak treba da bude osetljiv na kulturu organizacije tako da predloženo tehnološko rešenje može da se sagleda u kontekstu njene primene u organizaciji ili projektu. Trebalo bi takođe da identifikujemo pokretače i kočničare, tako da znamo ko ili šta će verovatno pomoći ili sprečavati usvajanje i efektivnost tehnologije. Na primer, uloga usvajaoca ima veliki udeo u prihvatanju date tehnologije, zasnovano na tome ko je promoviše, da li su slične tehnologije isprobane u prošlosti i koliko se nova tehnologija razlikuje od onoga što se trenutno upražnjava.

Taj sve veći skup dokaza treba podvrgnuti savršenijoj proceni, pri čemu se ispituje ne samo tehnologija nego i prateći dokazi. Odnosno, da bismo procenili da li skup dokaza formira ubedljiv argument za korišćenje tehnologije, posmatramo situacije u kojima je ona bila upotrebljena, poredimo stare načine sa novim načinima i određujemo (koristeći ankete, studije slučajeva, eksperimente, analize karakteristika i druge tehnike) da li je

dokaz protivrečan, konsistentan ili objektivan. Želimo da znamo koliko „solidan" slučaj imamo, da bismo se uverili da će nova tehnologija rešti poslovni problem za koji je namenjena.

Međutim, ubedljiv dokaz nije dovoljan da bi se obezbedilo usvajanje tehnologije. Svet marketinga nam kaže kako da upakujemo i podržimo tehnologiju da bismo je učinili „prijateljskom" i lakšom za razumevanje i upotrebu. Na primer, alati i pisani materijal za podršku prelaze dug put pri pomaganju korisniku da usvoji novu tehniku ili proces. Kada je potrebna infrastruktura prisutna kaže se da je tehnologija „spremna za prvi trenutak". Odnosno, ta upakovana tehnologija je najzad spremna za šire rasprostiranje, tako da mi, kao zajednica, možemo da procenimo brzinu njenog usvajanja i dokaz efektivnosti kada više organizacija izveštava o rezultatima na osnovu svojih iskustava.

U stvari, može nam zatrebati više modela rasprostiranja, uključujući iteraciju, na osnovu karakteristika tehnologije, usvajaoca i dokaza. Ti modeli, uz marketinške koncepte i dokaze koje smo opisali u ovom poglavlju, nude nam rečnik pomoću koga možemo da raspravljamo o ključnim pitanjima koja utiču na uspeh usvajanja i prostiranja tehnologije i da ih razumemo.

Sledeći koraci u poboljšavanju prenosa tehnologije

Više ne možemo sebi da dozvolimo ulaganja u tehnologiju čije je usvajanje nepoželjno ili verovatno neće biti uspešno. Praktičari mogu da pomognu istraživačima da razumeju prirodu uspešnog prenosa tehnologije i da koriste to novo shvatanje da bi izgradili modele i podršku za efektivnu procenu i usvajanje. Posebno, neke od tekućih empirijskih studija tehnologija softverskog inženjerstva su labavo povezane i slabo planirane, pa ne predstavljaju baš dobar zajednički generator koherentnog skupa dokaza. Ako bi empirijski istraživači softverskog inženjerstva organizovali studije tako da svaki deo doprinosi jasnom i značajnom rezultatu celine, rezultujući skup dokaza bio bi mnogo ubedljiviji. Odnosno, praktičari i istraživači moraju zajedno da planiraju šta da rešavaju u svakoj studiji i kako će rezultati studije doprinositi ukupnom skupu dokaza.

U isto vreme, praktičari zainteresovani za upotrebu nove tehnologije mogu efektivno da nauče iz mnogih već izvedenih studija i da primene to znanje na sopstvene projekte. Kočničari i promoteri prenosa tehnologije spadaju u tri kategorije: tehnološku, organizacionu i dokaznu. Na primer, prenos tehnologije se koči nedostatkom pratećih elemenata, nedostatkom odnosa prema pritisku tehničkog ili poslovnog problema i teškoćama u korišćenju i razumevanju. Suprotno od toga, nova tehnologija može da se promoviše alatima, dobro shvaćenim kontekstom i razumevanjem jasne moguće koristi za one koji su sada na projektu ili koji će kasnije koristiti njegove rezultate. Drugim rečima, lakše je preneti tehnologiju ako je ona jednostavna za upotrebu i ako je lako uočiti da će biti od pomoći.

Slično tome, praktičari mogu da procene postojeći dokaz i pomognu istraživačima da zasnuju nove studije na postojećim nalazima. Bilo da praktičari projektuju nove studije ili samo učestvuju u njima, oni bi trebalo da imaju poverenje u nosioce informacija o tehnologiji, da budu sigurni da će tehnologija jasno biti uzrok predviđenih efekata i da primene tehnologiju na realan problem na terenu, a ne samo na problemu-igrački koji nema nikakvog značaja za stvarni rad.

Ne znamo sve što bi trebalo o tome koja tehnologija softverskog inženjerstva najbolje funkcioniše u kojoj situaciji. Ne možemo da čekamo godinama da bismo to saznali, imajući u vidu munjevitu brzinu promene tehnologije. Ali, postoje lekcije koje treba da se nauče, kako iz istraživanja softverskog inženjerstva, tako i iz drugih disciplina usredsređenih na procenu i prenos tehnologije. Sada možemo da preduzmemo korake da ubrzamo usvajanje i izgradimo verodostojan skup podataka o efektivnosti naše prakse.

14.3 DONOŠENJE ODLUKA U SOFTVERSKOM INŽENJERSTVU

Uz potrebu za dokazom o novim tehnologijama, povezana je i potreba da se o njima donesu odluke. Kako odlučujemo koje tehnologije da upotrebimo u svom sledećem projektu? Kako dodeljujemo timu prave resurse? I kako odmeravamo rizik jednog izbora u odnosu na drugi? Kao kod prenosa tehnologije, možemo da pogledamo i druge discipline, da bismo videli kako istraživanje procesa donošenja odluka može da pomogne u poboljšanju opredeljenja vezanih za inženjerstvo softvera.

Mnogo odluka

Ponekad može da izgleda kao da softversko inženjerstvo predstavlja skup spregnutih aktivnosti povezanih donošenjem odluka i procenama. Projekte planiramo procenjujući resurse i rizik. Projekte ocenjujemo odlučivanjem da li su procesi bili efektivni, resursi odgovarajući i proizvodi zadovoljavajući. Da bismo testirali proizvode, kada ne postoji mogućnost testiranja svih elemenata, pristupamo oceni alternativa. Zahtevi za promenu i održavanje traže ocenjivanje alternativnih akcija, procenu potrebnih resursa i analiziranje rizika.

Nema potrebe da svoje odluke donosimo u vakuumu. Postoje teorije koje podržavaju proces donošenja odluka sa dve tačke gledišta: opisne i normativne. Opisne teorije obezbeđuju dokaz o tome kako se odluke stvarno donose, dok normativne teorije daju okvir i metode koji pomažu onima koji donose odluke da poboljšaju performansu u otkrivanju i rešavanju problema, uz data stvarna ograničenja. Na slici 14.3 ilustrovano je koliko drugih disciplina doprinosi informacijama o procesu donošenja odluka.

Proces donošenja odluka često obuhvata najmanje dva različita koraka. Prvo, pravimo pojedinačne izbore. Predviđamo ili zaključujemo, dodeljujemo vrednost različitim alternativama. Drugo, prilažemo naše nalaze procesu grupnog odlučivanja. Na primer, da bi procenio napor koji je potreban za izradu posebne vrste softvera, svako od nas može da napravi sopstveno predviđanje pre kombinovanja tih procena sa ciljem formulisanja grupnog predviđanja. Štaviše, ta „grupa" mogu u stvari da budu projekti, organizacije ili čak i društva; svaka takva odluka ima uticaj u odnosu na grupu koju će odražavati ili na koju će uticati.

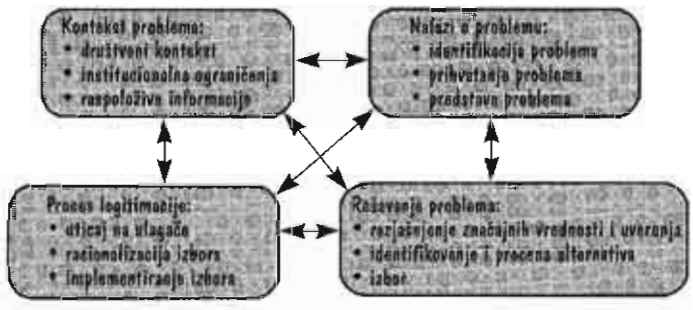
Slika 14.4 pokazuje da više elemenata može da utiče na način donošenja odluke. Kontekst situacije ograničava i naše razumevanje i naše izbore. Unutar tog konteksta, moramo da shvatimo i predstavimo problem, pre nego što pokušamo da ga rešimo. Svaki izbor treba da se proveri na više načina, da bi se odredili njegovi verovatni efekti na ulagače i stepen njegove racionalnosti i realizma. „Legitimacija" je nešto što je i veoma

značajno i što se donekle previđa u kontekstu softverskog inženjerstva. Shvatljivo je da procenitelji i oni koji donose odluke mogu da imaju svoje poželjne izbore za procene i odluke koji mogu lakše da se opravdaju. Ti poželjni izbori nagoveštavaju naginjanje ka nekom posebnom pristupu, kao što je npr. upotreba algoritamskih modela u odnosu na ekspertsko rasuđivanje. Moguća rešenja treba da se sagledaju kroz filtar naših vrednosti i uverenja.

	Opisne teorije	Normativne teorije
Pojedinac	Psihologija	Teorija odlučivanja
	Marketing	Ekonomija
	Psihijatrija	Operaciona istraživanja
	Književnost	Filozofija i logika
Grupa	Socijalna psihologija	Teorija igara
	Organizaciono ponašanje	Organizaciono ponašanje
	Antropologija	Klinička psihologija i terapija
	Sociologija	Finansije i ekonomija
Organizacija	Teorija organizacije	Planiranje i strategija
	Sociologija	Teorija upravljanja i kibernetika
	Industrijske organizacije	Organizacioni dizajn
	Političke nauke	Teorija tima i ekonomija
Društvo	Sociologija	Filozofija prava
	Antropologija	Političke nauke
	Makroekonomija	Socijalni izbor

SLIKA 14.3 Koreni nauka o odlučivanju (prilagođeno iz Kleindorfera i dr. 1993)

Da biste videli kako koristimo te elemente pri donošenju odluka, razmotrite problem biranja novog kancelarijskog prostora. U tabeli 14.1 predstavljeno je pet opcija. Svaku alternativu karakterišu najam u dolarima po mesecu, rastojanje u kilometrima od stana, površina u kvadratnim metrima i opšta, subjektivna ocena kvaliteta prostora. (Na primer, prostor visokog kvaliteta ima više svetla ili višu tavanicu od onog sa nižim kvalitetom.)



SLIKA 14.4 Aspekti donošenja odluka (prilagođeno iz Kleindorfera i dr. 1993)

TABELA 14.1 Opcije kancelarijskog prostora

Opcija kancelarije	Mesečni najam	Rastojanje od stana	Površina (m^2)	Kvalitet
1	450 USD	10 kilometara	4000	srednji
2	475 USD	15 kilometara	2500	visok
3	460 USD	14 kilometara	1500	prosečan
4	500 USD	5 kilometara	1750	visok
5	510 USD	7 kilometara	2500	visok

Postoji više pravila koja možemo da primenimo da bismo izabrali najbolju opciju. Na primer, možemo da izaberemo kancelariju sa najnižim najmom. Ili, možemo da odaberemo onu koja je najbliža našem stanu. Ta pravila odražavaju naše vrednosti; neko ko koristi prvo pravilo umesto drugog više vrednuje novac nego vreme. Alternativno, možemo da koristimo složenija pravila. Na primer, možemo da definišemo „vrednost kancelarije" kao kombinaciju najma i površine i da je uravnotežimo sa vremenom najkraćeg puta. Ili, možemo da primenimo pristup u više koraka, gde prvo koristimo nivoe eliminacije na osnovu najma i rastojanja (na primer, ne više od 500 USD za najam i ne dalje od deset kilometara od stana), a zatim uravnotežujemo preostale atribute.

Naravno, naš proces izbora može da bude savršeniji. Na primer, možemo da koristimo procedure dominacije, gde odstranjujemo alternative koje su „prevladane" boljim izborima. Međutim, ta vrsta pravila može da dovede do podoptimizacije; možemo da odstranimo dosta dobar izbor ako su naše granične vrednosti proizvoljne, ili nisu pažljivo razmotrene. Ili možemo da koristimo konjunkciju (svaka dimenzija je u saglasnosti sa standardom) ili disjunkciju (svaka dimenzija je zadovoljavajuće visoka). U tim situacijama, nema popuštanja kada su vrednosti karakteristika blizu praga; možemo da odbacimo izbor zato što je najam preko 500 USD, ali u stvari ostale karakteristike izbora od 501 USD su daleko bolje od onih u drugim opcijama.

Druga strategija je da se koristi odstranjivanje po aspektima. Tu svaki atribut ima unapred dodeljenu vrednost kriterijuma i svakom atributu je dodeljen prioritet. Atributi se zatim analiziraju u skladu sa njihovim relativnim značajem. Možemo da formalizujemo taj pristup korišćenjem modela aditivne vrednosti, gde dodeljujemo težinske koeficijente ili prioritete (w_i) svakom atributu (x_i), a zatim sabiramo proizvode težinskih koeficijenata i vrednosti atributa ($v(x_i)$):

$$V = \sum_{j=1}^{n} w_j v(x_j)$$

Ponekad se težinski koeficijenti i poređenja lakše naprave usvajanjem pristupa po parovima, kao kod Saatyjevog analitičkog hijerarhijskog procesa, jednog drugog primera pomoći za višekriterijumsko odlučivanje.

Svaki od tih pristupa predlaže „pravi" izbor, ali to može i da ne bude uvek optimalni izbor. Ili, može da obuhvati mnogo proračuna ili poređenja. U stvarnosti, možemo da koristimo heuristički pristup koji nam daje dovoljno dobar odgovor.

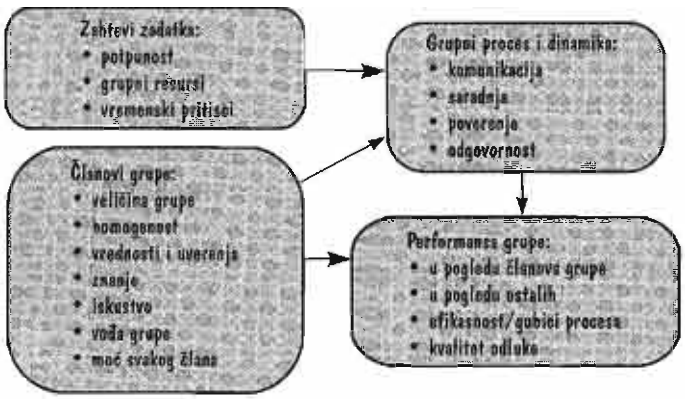
Grupno odlučivanje

Do sada smo govorili o karakteristikama u vezi sa samim problemom. Grupno odlučivanje je u izvesnom smislu teže, zato što aspekti ponašanja grupe utiču na to kako se odluču-

je. Na slici 14.5 ilustrovana su neka od pitanja koja moraju da se razmatraju kada više ljudi pokušava da izabere između ponuđenih alternativa. Na primer, poverenje, komunikacija i saradnja mogu da utiču na rezultat; ništa od toga nije činilac kada izbor vrši pojedinac.

Međutim, postoji više strategija grupnog odlučivanja koje rešavaju ta pitanja. Na primer, dijalektičke strategije mogu da dopuste jednoj strani da iznese argument, a zatim drugoj strani da govori. Treća strana može da se angažuje na mirenju različitih gledišta. Alternativno, može da se koristi slobodno iznošenje ideja da bi se identifikovala potpuna lista mogućnosti, uključujući povoljne prilike i opasnosti. Tehnike nominalne grupe obuhvataju tiho generisanje ideja koje se posle kružno razmatraju, po jedna u jednom trenutku, a zatim spontano ocenjuju koristeći tajno glasanje. Ili, mogu da se koriste pristupi socijalnog rasuđivanja, da se razdvoje činjenice od rasuđivanja, ili da se napravi razlika između nauke i vrednosti socijalnog rasuđivanja.

Kada je grupna organizacija, oni koji donose odluke moraju da razlikuju strategijske od taktičkih i rutinskih odluka. Strategijske odluke utiču na dobrobit i prirodu organizacije; one po pravilu obuhvataju nove proizvode, usluge ili tržišta i tu više rukovodstvo mora da ima značajnu ulogu. Procene troškova mogu da budu deo strategijskih odluka, posebno ako se koriste da bi se proizvod plasirao na tržište. Taktičke odluke utiču na određivanje cene, angažovanje zaposlenih, interakciju sa kupcem ili operacije, ali one ne utiču u istom stepenu na finansijsku osnovicu organizacije ili na komercijalno upravljanje. Taktička procena troškova može da se koristi da bi se uspostavila cena novog proizvoda, gde nije u pitanju deo tržišta; na primer, kada jedan sektor kompanije razvija proizvod za drugi sektor, konkurencija i određivanje cene nisu od strategijskog značaja.



SLIKA 14.5 Pitanja grupnog donošenja odluka

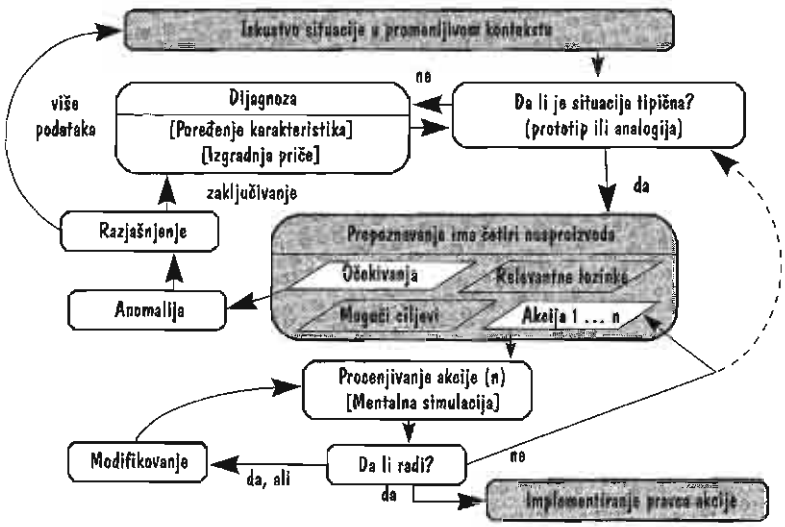
Rutinsko odlučivanje je obično prizemnije: po prirodi ponavljajuće, po opsegu lokalno i vođeno organizacionim pravilima ili politikama. Na primer, pretpostavite da kompanija podržava sopstveno skladište za ponovno korišćenje, gde se softverski inženjeri nagrađuju za „ulaganje" ponovno upotrebljive komponente i „naplaćuje im se" za korišćenje komponente koja već postoji u skladištu. Određivanje „cene" komponente može da bude rutinski zadatak zasnovan na zajedničkim smernicama.

Kako stvarno odlučujemo

Nauka o odlučivanju i literatura o operacionim istraživanjima je puna primera tehnika odlučivanja. Koje od tih tehnika stvarno koristimo u procesu donošenja odluke? Pregled o kome je izvestio Forgionne (1986) nagoveštava da smo skloni da koristimo statistiku i simulaciju, ali veoma malo složenijih procesa koji se predlažu u literaturi.

Postoji više razloga zbog kojih često izbegavamo tehnički savršenije pristupe. Najveće prepreke njihovom korišćenju su teškoća izvođenja proračuna i kombinatorna eksplozija mogućnosti. Umesto da formuliše hipoteze o najboljem načinu odlučivanja, Klein (1998) je posmatrao one koji donose odluke u radu, pod pritiskom. U jednoj studiji od 156 posmatranja, on je otkrio da niko od njih nije koristio unapred izabrane opcije (gde neko drugi postavlja šta možete da uradite i onda birate između tih mogućnosti). Osamnaest ispitanika je napravilo komparativnu procenu, gde je bila izabrana početna opcija a onda su sve ostale opcije upoređene sa njom, da bi se videlo da li je ona bila najbolja; tu su oni optimizovali svoju akciju. Jedanaest ispitanika je formulisalo novu opciju. Ali ostali su koristili ono što Simon zove zadovoljavajuća strategija: oni su procenjivali svaku opciju na osnovu njenih sopstvenih prednosti, dok nisu pronašli prvu koja je zadovoljila njihove kriterijume prihvatljivog rešenja.

Posle posmatranja i intervjua sa vatrogascima, tehničarima medicinske hitne pomoći, vojnicima i drugima koji donose važne odluke pod pritiskom, Klein je predložio „model odluke na osnovu prepoznavanja", kao što je prikazano na slici 14.6, da bi opisao kako stvarno donosimo odluke. On ističe da smo skloni da u svojim glavama skladištimo primere i da te situacije izvlačimo kada smo suočeni sa izborom. Mi se mentalno obraćamo situaciji iz prošlosti i poredimo je sa sadašnjom, da bismo videli da li je slična. Umesto da poredimo sve opcije, mi grabimo onu koja je „bliska" i prolazimo kroz proces u svojim glavama, da bismo sebi opravdali da je ona „dovoljno bliska" da bi mogla da se upotrebi za sadašnju odluku. U isto vreme, koristimo mentalnu simulaciju da bismo odredili da li će akcije koje predlažemo da se preduzmu zaista funkcionisati u toj situaciji; kada mislimo da neće, vraćamo se nazad, biramo različit scenario i opet izvodimo mentalnu simulaciju. Na kraju, kada smo zadovoljni svojim izborom, delujemo.



SLIKA 14.6 Model odluke na osnovu prepoznavanja (prilagođeno prema Kleinu 1998)

Ali, donošenje odluke i procena nisu tako jednostavni kao što model nagoveštava. Hershey, Kunreuther i Schoemaker (1982) su pokazali da kontekst odluke može da polarizuje izbor. Da biste videli kako, razmotrite ova dva pitanja:

Pitanje 1: Imate 50% šanse da izgubite 200 USD i 50% šanse da ne izgubite ništa. Da li biste platili 100 USD da izbegnete ovu situaciju?

Pitanje 2: Možete da platite 100 USD da izbegnete situaciju u kojoj možete da izgubite 200 USD ili ništa. Da li biste platili ako bi bilo 50% šanse da izgubite?

U njihovoj studiji, oni su tražili od svakog iz dve ekvivalentne grupe da odgovori na pitanje sa „da", „ne" ili „svejedno". Čak iako su dva pitanja ekvivalentna, 6% grupe koja je odgovarala na pitanje 1 je bilo spremno da plati, ali 32% od grupe koja je odgovarala na pitanje 2 je bilo spremno da plati.

Tversky i Kahneman (1981) su ilustrovali istu vrstu polarizacije u donošenju odluke prilikom analize rizika. Oni su opisali situaciju gde je retka bolest ugrozila 600 ljudi u selu. Službenici javnog zdravstva imaju količinu vakcine koja je dovoljna za 200 ljudi i razmatraju dve različite mogućnosti. Oni mogu da daju punu vakcinu za 200 ljudi (program A), ili mogu da je razblaže vodom i nadaju se da će ona zaštititi svih 600 seoskih stanovnika (program B). Istraživači su tražili od grupe da izabere između dva programa, i to:

Program A: biće spašeno tačno 200 života.

Program B: jedna trećina šanse je da će se spasti svih 600, a dve trećine šanse da se neće spasti niko.

U alternativnom okviru, istraživači su tražili od ekvivalentne grupe da napravi isti izbor, ali ovog puta da se programi prikažu pomoću života koji će se izgubiti:

Program C: biće izgubljeno tačno 400 života.

Program B: jedna trećina šanse da neće niko umreti, a dve trećine da će umreti 600 ljudi.

Čak iako su problemi matematički identični, razlika u odgovorima je bila dramatična. Sedamdeset dva procenta učesnika je izabralo A u prvom okviru; samo 22% je izabralo C u drugom okviru.

U sličnoj studiji, ljudi su povezali situacije iz prošlosti sa sadašnjim odlukama. Da biste videli kako, razmotrite situaciju u kojoj idete u pozorište da biste videli komad. Kada dođete do biletarnice, otkrivate da ste izgubili novčanicu od 10 USD. Da li biste platili 10 USD za kartu za komad? Osamdeset osam procenata ispitanika je odgovorilo potvrdno. Ali ako, alternativno, kada dođete do biletarnice otkrijete da ste izgubili svoju kartu vrednu 10 USD, da li biste platili 10 USD za novu kartu? U toj situaciji, 46% bi platilo, čak iako su situacije matematički identične.

Dakle, važno je razmotriti kontekstualnu sklonost, posebno kad se procenjuje napor ili rizik. Način na koji se uokviri problem može da proizvede radikalno drugačije rezultate.

U Steeleovom radu (1999) opisuje se taj fenomen. On napominje da očekivanje performanse ili rezultata može da dovede do „stereotipne pretnje". Na primer, ako se po-

sebnoj grupi ljudi kaže da ona obično ne radi dobro u datoj situaciji, onda taj stereotip stvarno može da za rezultat ima slabiju performansu.

Drugi izvori sklonosti mogu da se uvuku u donošenje odluka. Na primer, ljudi obično precenjuju ono što već poseduju (to se zove *„status quo* polarizacija"). Taj fenomen može da utiče na procenjivača da bude preterano optimističan u vezi sa produktivnošću u poznatoj razvojnoj situaciji. Slično tome, verovatnoća i isplativost mogu međusobno da reaguju: ako su izgledi mali, ljudi prvo gledaju isplativost; ako su izgledi veliki, ljudi gledaju verovatnoću, a zatim isplativost.

Pojedinci pokazuju zapaženu sklonost ka informacijama koje su specifične za slučaj ili pojedinačne, nasuprot statističkim ili distribucionim informacijama. Busby i Barton (1996) su se usredsredili na tu sklonost kada su opisivali procenjivače koji su upotrebili pristupe odozgo nadole ili pristupe prestanka rada da bi predviđali. Nažalost, taj pristup nije uspeo da prihvati neplaniranu aktivnost, pa su predviđanja bila dosledno potcenjena za 20%. Po definiciji, dokaz specifičan za slučaj je da svaki projekat neće uspeti da uračuna neplanirane aktivnosti, ali statistički dokazi mnogih projekata nagoveštavaju da se neplanirane aktivnosti vrlo verovatno događaju. I pored toga, menadžeri su davali prednost pojedinačnim dokazima i ne bi uključili faktor neplaniranih aktivnosti u vlastiti proces procene.

Pored toga, moramo da zapamtimo da na podsećanje utiče i nedavna prošlost i živost iskustva. Što faktor seže dalje u prošlosti, to je veća sklonost onog koji ga se priseća da smanji njegov značaj. Sa tog stanovišta, to umanjenje značaja može da bude razumno, imajući u vidu da se način na koji razvijamo softver značajno promenio tokom godina. S druge strane, mnogi rizici, kao što su npr. zahtevi koji su izmenjeni ili pogrešno shvaćeni, veoma malo su se promenili .

Usidrenje i podešavanje je druga uobičajena tehnika koju procenjivači koriste Tu procenjivač bira analognu situaciju i zatim je podešava da bi odgovarala novim okolnostima. Međutim, postoji značajan dokaz koji nagoveštava da su procenjivači preterano oprezni kada podešavaju. Drugim rečima, preovladuje sidrenje posle koga se prave nedovoljna prilagođavanja. Na taj pristup može da utiče podsećanje, time što najpogodnije analogije mogu da se previde zato što nisu skorašnje.

Strah da se ne ostavi negativan utisak može takođe da utiče na ekspertsko rasuđivanje. DeMarco (1982) nas podseća da „realizam može pogrešno da se tumači kao nelojalnost", što dovodi do neumesnog optimizma u predviđanju, kako pojedinačnog tako i grupnog.

Kako grupe stvarno donose odluke

Mnoge organizacije koriste grupne tehnike za donošenje odluka da bi izvele značajne projekcije ili procene. Na primer, Delfi tehnika (pogledajte podsetnik 14.1) omogućava da više procenjivača kombinuje svoje različite procene u jednu kojom su svi zadovoljni.

PODSETNIK 14.1 DELFI TEHNIKE

Delfi tehniku je prvobitno razvila Rand Corporation krajem 1940-ih godina, kao metod za strukturisanje grupnih komunikacionih procesa za rešavanje složenih problema. Vlada SAD koristila je Delfi da predvidi budućnost naftne industrije. Delfi tehnika je namenjena da se dođe do saopštene procene držanjem pojedinačnih predviđanja u tajnosti uz više iteracija procene. Konsenzus nije od suštinske važnosti. U stvari, Delfi tehnika može takođe da se upotrebi da bi obrazovala učesnike zaključivanjem na osnovu multidisciplinarnih ili različitih ulaza.

Tehniku je kasnije pročistio i popularizovao Barry Boehm (1981.) za specifičniji zadatak procene softverskog projekta. Glavni koraci Delfi procesa uključuju sledeće:

1. Grupa eksperata dobija specifikaciju i formular za procenu.
2. Eksperti raspravljaju o proizvodu i pitanjima procene.
3. Eksperti prave pojedinačne procene.
4. Procene se tabeliraju i vraćaju ekspertima.
5. Ekspert je svestan samo svoje sopstvene procene; izvori ostalih procena ostaju anonimni.
6. Eksperti se sastaju da bi raspravljali o rezultatima.
7. Procene se revidiraju.
8. Eksperti kruže kroz korake od 1 do 7 dok se ne dobije prihvatljiv nivo konvergencije.

Dok je ta tehnika relativno dobro poznata i objavljena u mnogim knjigama o softverskom inženjerstvu, ima malo objavljenih iskustava o upotrebi Delfi tehnike za predviđanje softvera.

Ali dinamika grupe može da utiče na kvalitet odluke ili procene. Na primer, Foushee i dr. (1986) su pronašli da članovima tima treba vremena da nauče da budu zajedno produktivni. Timovi su bolje radili na kraju svog zadatka nego na početku, zato što su naučili da efektivno rade zajedno.

Dinamika grupe može takođe da ima i negativne efekte. Asch (1956) je ispitivao efekat kolega na odluku pojedinca o predstavljanju predmeta sa linijama prikazanim na slici 14.7. Kada su pojedinci upitani koja je od tri linije sa desne strane bila iste dužine kao testna linija, skoro svi ispitanici su odgovorili tačno: linija B. Ali, kada su u prostoriji bili prisutni i ostali i dali pogrešan odgovor, broj grešaka se dramatično povećao, kao što je prikazano u tabeli 14.2.



SLIKA 14.7 Primer korišćen u procene grupnih efekata

TABELA 14.2 Rezultati Aschove studije (1956)

Uslov	Procenat grešaka
Ispitanik je sam	1%
Sa jednom osobom koja kaže A	3%
Sa dve osobe koje kažu A	13%
Sa tri osobe koje kažu A	33%
Sa šest osoba koje kažu A i jednom koja kaže B	6%

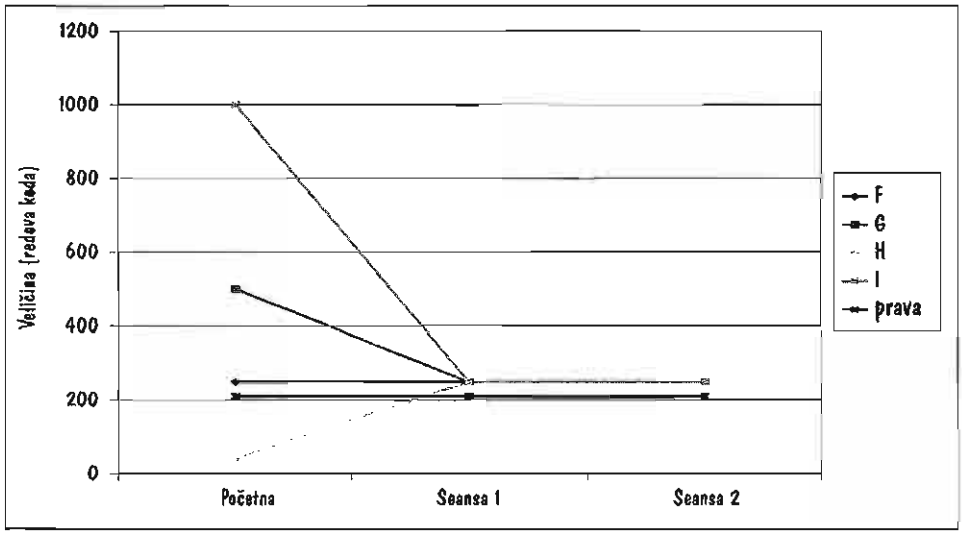
Skromna studija posmatranja

Da bi ispitali grupno donošenje odluka u softverskom inženjerstvu, Pfleeger, Shepperd i Tesoriero (2000) su istraživali pristupe za procenu radnog napora. Oni su počeli sa pilot istraživanjem na Univerzitetu Bournemouth. Dvanaest postdiplomaca je organizovano u četiri tima koji su imali od dva do četiri člana. Kao deo rada na kursu, od svakog tima se tražilo da dođe do zahteva i razvije prototip jednostavnog informacionog sistema. Zatim je od tima traženo da predvidi veličinu prototipa u broju linija koda. (Da bi minimizovali probleme brojanja, istraživači su definisali liniju koda kao ograničavač iskaza. Linije koda su odabrane zato što ih je moguće lako verifikovati, a ne zbog bilo kog drugog posebnog razloga.) Učesnici nisu bili ograničeni na bilo koju posebnu tehniku, mada su u praksi bili skloni da koriste subjektivnu procenu. Neposredno posle toga, učestvovali su na Delfi sastancima koji su doveli do dve dodatne procene.

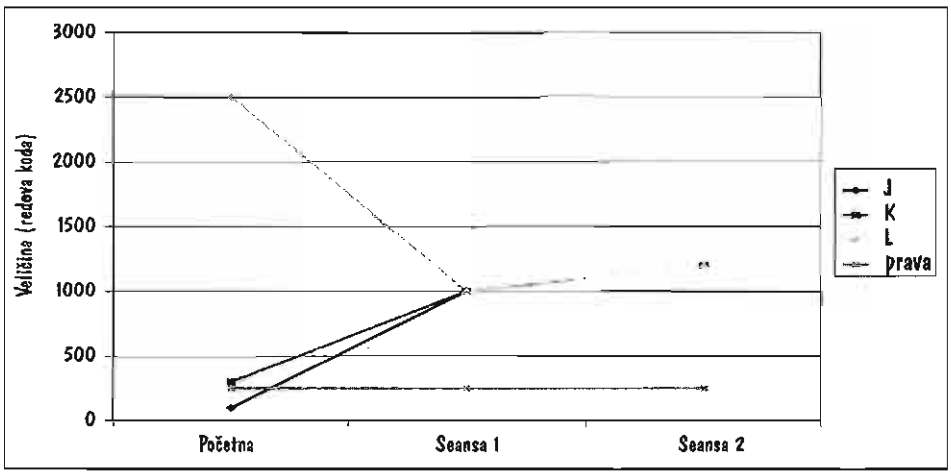
U tabeli 14.3 je pokazano da su i medijana greške i širina opsega grešaka najveće za početnu procenu. Drugim rečima, greška je bila najveća pre korišćenja Delfi tehnike; kasnije Delfi seanse su dovele do smanjivanja razlika između predviđenih i stvarnih veličina prototipova. Kao što je prikazano na slikama 14.8 i 14.9, tri od četiri grupe su pokazale jasno poboljšanje, ali četvrta grupa je odstupila od prave vrednosti kada se proces nastavio. To odstupanje je bilo delom posledica prevlasti jednog člana u grupi. Iako Delfi tehnika dozvoljava anonimnost za pojedinačne procene, ona je ranjiva od strane nasilnih pojedinaca. Zanimljivo je da je taj rezultat u saglasnosti sa teorijama ponašanja koje smo ranije opisali. Kao što smo videli, u literaturi o performansi grupe (Sauer i dr. 2000) nagoveštava se da je glavna odrednica ishoda izbor šeme odlučivanja, kao i to kako grupa koristi svoja ekspertska znanja.

TABELA 14.3 Procena grešaka iz tri seanse predviđanja veličine

Procena	Medijana greške	Minimalna greška	Maksimalna greška
Početna	160,5	23	2249
Seansa 1	40	23	749
Seansa 2	40	3	949



SLIKA 14.8 Procene konvergentnih grupa



SLIKA 14.9 Procene divergentnih grupa

Ohrabrujuće je da je slična studija sa postdiplomcima na Univerzitetu Maryland dala uporedive rezultate, u tome da su uzastopne seanse Delfi tehnike dovele do značajnog smanjenja u opsegu procena. U ovom slučaju, zadatak procene je bio teoretski, pa tačnost procena nije mogla da se oceni. Na Marylandu, svi studenti su bili zaposleni praktičari sa značajnim iskustvom; oni su se upisali ma magistarske studije softverskog inženjerstva. Razred od deset je podeljen na tri tima od po tri ili četiri studenta. Kao deo opštijeg projekta koji je obuhvatao postavljanje zahteva, analizu i procenu, pojedinci iz svakog

tima su koristili dva proizvoda, Data Salvage (analogan alatu razvijenom na Univerzitetu Bournemouth) i Revic (alat zasnovan na COCOMO-u, razvijen za potrebe Ministarstva odbrane SAD), da bi napravili početne procene za projekat svog tima. Zatim, posle časa na kome su naučili da koriste Delfi metod, studenti su posmatrani i snimani na traci kada su radili na dve 20-minutne Delfi seanse, da bi konvergirali u timskoj proceni. Odnosno, svakom timu su dati rezultati svih pojedinačnih procena, ne samo od njihovog tima, nego od svih u razredu. Od svakog tima se tražilo da zapiše svoj stepen poverenja u novu procenu i da dokumentuje pretpostavke koje je napravio u izvođenju procene. U vreme studije, studenti nisu bili svesni da su istraživači bili zainteresovani za dinamiku njihove rasprave, a ne za stvarne brojeve koje su napravili.

Pfleeger, Shepperd i Tesoriero su zapazili nekoliko značajnih težnji u timovima. Prvo, raspon od najmanje do najveće procene se dramatično smanjivao u toku vremena. On je išao od 16239 na početku prve seanse, preko 10181 na početku druge, do 1520 na kraju.

Drugo, postojao je opšti porast poverenja kako su studenti napredovali kroz Delfi rasprave. Odnosno, svi studenti su izvestili o povećanju poverenja u svoje procene posle grupne rasprave, bez obzira na nivo svog iskustva. Međutim, nije bilo dokaza o bilo kakvom odnosu između iskustva i poverenja.

S obzirom na to da ličnost može da igra tako značajnu ulogu u Delfi raspravama, svaki od studenata na Marylandu je uradio Myers-Briggsov test. Myers-Briggsov test klasifikuje svakog ispitanika u jednu od četiri skale: ekstrovertan/introvertan, rezonovanje/intuicija, mišljenje/osećanje i prosuđivanje/zapažanje. Dakle, o rezultatima se izveštavalo kao o kombinaciji četiri slova, na primer IRMP za introvertan-rezonovanje-mišljenje-prosuđivanje, da bi se opisao način na koji student obično odgovara na situaciju. Rezultati za drugu grupu su bili posebno zanimljivi. Dva člana te grupe bila su klasifikovana kao IRMP, treći je bio ERMP, a četvrti IROP. Ljudi sa tipom ličnosti IRMP teže da budu precizni i tačni, prate pravila i procedure i imaju odličnu koncentraciju. Međutim IRMP-ovi ne vole promene i skloni su da budu kruti. IROP kao i IRMP su skloni tačnosti, iscrpnosti i pažljivi su sa detaljima, a pružaju i otpor promeni. Zanimljivo je da su ličnosti IROP sklone da potcenjuju sopstvenu vrednost. Suprotno od toga, ERMP-ovi su skloni da budu praktični i orijentisani ka rezultatima. Oni se usredsređuju na cilj, postajući nestrpljivi sa neefikasnošću i sa onima koji ne prate procedure. Delfi rasprave grupe 2 su se usredsredile na detalje parametara alata Revic, a ne na globalnija pitanja o projektu uopšte. Možda su tipovi ličnosti orijentisani ka detaljima naveli članove te grupe da gledaju više u alat nego u sopstvena iskustva. Kao što može da se vidi u tabeli 14.4, grupa 2 je imala najmanje poverenja u svoju konačnu procenu, možda zbog tipova ličnosti svojih članova.

Iz upitnika koji je bio urađen posle studije, izlazi da je deset ispitanika imalo povoljan stav prema Delfi metodi jer su svi izvestili o tome da je tehnika povećala poverenje u njihove procene. Kada su upitani da li bi svako od njih razmotrio upotrebu Delfi tehnika za procenu u radnoj situaciji, pet je reklo da, četiri možda, a samo jedan ne.

TABELA 14.4 Poverenje grupe sa Marylanda u konačnu procenu

Grupa	Broj u grupi	Medijana poverenja u procenu	Opseg procena
1	2	91	2
2	4	65	15
3	3	80	0
4	3	80	0

U tabeli 14.5 izlistane su pozitivne i negativne stavke koje su ispitanici identifikovali odgovarajući na upitnik. One su sortirane po opadajućoj učestalosti, tako da se one, koje se najčešće pominju, pojavljuju prve. Zapazite da su problemi koji se odnose na dominantnog pojedinca i nedostatak ekspertize bili najčešće navedeni nedostaci, dok su koristi od različitih gledišta i vrednost grupne diskusije bile najčešće navedene prednosti.

TABELA 14.5 Zapažene snage i slabosti tehnike Delfi

Slabosti	Snage
Može pogrešno da utiče na pojedinca i uticaj dominantnog pojedinca	Eksperti sa različitom osnovom/perspektivama
Zavisi od znanja/ekspertize pojedinaca	Grupna rasprava može da ispravi greške
Rizik od pogrešnih pretpostavki	Ponovno razmatranje
Grupne rasprave prave malu razliku u rezultatu (konsenzus grupa)	Koristi ekspertsko rasuđivanje Medijana bolja od srednje vrednosti
Velika promenljivost u predviđanjima	Obezbeđuje poređenje sa drugim procenama
Neodgovarajući cilj, trebalo bi da se koristi za detaljnije probleme	Anonimnost/nezavisnost kombinovana sa grupnim koristima

Naučene lekcije

Iz dve studije o Delfi proceni proistekao je izvestan broj lekcija. Prva je da su subjekti imali široko pozitivan stav prema tehnici. Štaviše, iz studije na Bournemouthu jasno je da je tehnika dovela do poboljšanih procena. U obe studije tehnika je jasno smanjila opseg procena, čak i ako je šest od deset studenata sa Marylanda izvestilo o negativnim efektima grupne diskusije. Kao što su istraživači poput Kleina istakli, postoje druge, indirektne koristi, uključujući obrazovanje i razvoj zajedničkog rečnika.

Druga jasna lekcija je da ličnosti mogu da preovlađuju u raspravi, čak i kada dominantni učesnik nije korektan, vrlo slično onome što je Asch pronašao nekoliko decenija ranije. Štaviše, pretpostavke pojedinaca koje su uobličile osnovu početnih promena (odnosno, činioci zahtevani kao ulaz za korišćene alate) bile su nebitne u većini narednih grupnih rasprava. Taj rezultat je u paraleli sa nalazima u izveštajima o istraživanjima grupnih sastanaka za preglede i inspekcije, gde su mnogi pojedinačni nalazi izgubljeni kada nisu bili potvrđeni od ostalih članova grupe. Posebno, usredsređenje Delfi rasprava se okreće ka samim brojevima (a ne tamo odakle dolaze ti brojevi) i ka opravdavanju

unutrašnjih osećanja, kako nagoveštava Klein. Drugim rečima, sidrenje i podešavanje je živo i zdravo u Delfi tehnici .

Mi često pretpostavljamo da će oni sa najvećim iskustvom najjače uticati na grupne diskusije, dovodeći do realnih procena. Ali u studiji Pfleegera, Shepperda i Tesorieroa, čak i najiskusnija grupa (Maryland grupa 3) je gledala medijanu da bi se uverila. A svi studenti su izvestili o porastu poverenja koje nije bilo ni u kakvom odnosu sa iskustvom članova tima. Dakle, ličnosti mogu da dominiraju nad iskustvom u Delfi raspravama, što je situacija koja ne dovodi često do realističnih i tačnih krajnjih projekcija.

Većina postojećih istraživanja tehnika procene usredsredila se na tačnost pojedinačnih procena. Međutim, većina praktičara pravi svoje procene u grupi, eksplicitno se oslanjajući na tehniku kao što je Delfi, ili implicitno dolazeći do mišljenja svojih kolega. Iz tog razloga, važno je priznati ulogu grupne dinamike u procesu procene i, zaista, u grupnom donošenju odluka uopšte. Mi često pretpostavljamo da će ekspertiza i iskustvo dominirati, da imamo visoko kvalitetne podatke u istorijskim zapisima sličnih projekata odakle možemo da izvučemo svoje analogije i da znamo kako da kažemo kada su dva projekta slični jedan drugom. Nažalost, mnoge od tih pretpostavki su pogrešne u izvesnom obimu, a praktičari moraju da se oslone na pojedinačne i grupne alate i tehnike pomoću kojih mogu da naprave projekciju ili donesu odluku. Mada tvrdimo da je objektivnost suštinska, u stvari smo prisiljeni da se oslonimo na subjektivne i često nejasne podatke i alate da bismo doneli svoje odluke. Studije posmatranja, kombinovane sa solidnim razumevanjem grupne dinamike, mogu da nam pomognu da skrojimo svoj proces donošenja odluka prema realnosti ličnosti i procesa i tako povećamo svoje poverenje u rezultate.

Štaviše, ono što naučimo o donošenju odluka u postavkama kao što je procena može da se primeni u širem kontekstu. Možemo da naučimo kako da budemo efektivniji u kombinovanju pojedinačnih pretpostavki i mišljenja u grupni konsenzus i kako da pripremimo i nepristrasno predstavimo dokaze.

14.4 PROFESIONALIZACIJA SOFTVERSKOG INŽENJERSTVA: LICENCIRANJE, SERTIFIKACIJA I ETIKA

Softversko inženjerstvo je do sada prešlo dugačak put. Ali, ono mora da ide i dalje ako treba da se smatra zrelim kao druge inženjerske discipline. Više država u SAD insistira na tome da softverski „inženjeri" budu obučeni i licencirani, isto kao i ostali inženjeri. Okupilo se više grupa da odredi šta sačinjava „osnovna znanja" softverskog inženjerstva – odnosno skup znanja i veština koje je razumno očekivati od svakog diplomskog programa softverskog inženjerstva na univerzitetskom nivou, ili od svakog licenciranog profesionalca. Ti radovi moraju da daju odgovore na teška pitanja o razlici između softverskog inženjerstva i drugih računarskih disciplina. U ovom odeljku, ispitujemo napore da se formalizuje, kodifikuje i ispita dobra praksa softverskog inženjerstva. Istražujemo takođe potrebu za etičkim kodeksom da bi se obezbedilo da softverski inženjeri rade ispravno kada razvijaju i održavaju softver.

Usredsređenje na ljude

Istraživači softverskog inženjerstva su prvo pokušali da poboljšaju kvalitet softvera ispitivanjem samo proizvoda razvoja softvera. Kao što smo videli u prethodnim poglavljima, zahtevi, dizajn, kôd i dokumenti koji se na njih odnose mogu da se pregledaju i verifikuju, da bi se odredilo da li proizvodi zadovoljavaju standarde kvaliteta koje su postavili kupci i praktičari. Ali teškoće i troškovi procenjivanja proizvoda (posebno krajnjeg proizvoda koji je u radu) i varijacije proizvoda u domenima čine takvo obezbeđenje kvaliteta zastrašujućim. Sledeći talas pokušaja poboljšanja obuhvata softverski proces. Modeli zrelosti kao što je CMM, međunarodni standardi procesa kao što je ISO 9000 i druge aktivnosti ne gledaju samo u ono što je proizvedeno, nego i na način na koji je to urađeno. Veoma slično kako se najbolji šefovi kuhinja obučavaju da gledaju recepte i tehnike kuvanja kao i ukus spremljenih jela, istraživači softverskog inženjerstva i praktičari su videli poboljšanje procesa kao način za poboljšanje proizvoda.

Još skorije, zajednica softverskog inženjerstva je okrenula svoju pažnju na ljude koji prave softver. Posebno, čine se napori da se poboljša obrazovanje iz softverskog inženjerstva i istraži da li licenciranje ili sertifikacija može da dovede do značajnih poboljšanja procesa i proizvoda. U najvećem delu, ti napori su reakcija na veliki broj ljudi koji su u softversku profesiju ušli bez bilo kakvog formalnog obrazovanja u oblasti softvera (odnosno, samouki studenti programeri, baš po izlasku iz srednje škole) i koji se samima sebi, klijentima i potencijalnim poslodavcima predstavljaju kao softverski inženjeri. Napori koji se usmeravaju ka tom pitanju verovatno neće poboljšati kvalitet bezbedno kritičnog softvera, zato što organizacije koje razvijaju takve proizvode već zapošljavaju dobro kvalifikovane ljude i znaju da procene kvalifikacije svojih potencijalnih zaposlenih. Umesto toga, napori da se „poboljša praktičar" namenjeni su da se poboljša prosečan kvalitet softverskih proizvoda i procesa, da bi se obezbedilo da praktičari ispod proseka zadovolje minimum standarda softverskog znanja i kompetencije i da se javnost zaštiti od nekvalifikovanih praktičara.

Napori u obrazovanju u oblasti softverskog inženjerstva imaju za cilj povećanje skupa dobro kvalifikovanih projektanata softvera; oni su takođe namenjeni za izgradnju skupa licenciranih profesionalnih inženjera koji razumeju osobine, mogućnosti, ograničenja i složenosti softvera i koji poseduju ekspertska znanja da naprave kvalitetne softverske komponente za inženjerske proizvode (na primer, automobile). Namera koja postoji iza sertifikovanja ili licenciranja softverskih praktičara je da pomogne da se obezbedi da praktičari zadovolje minimum standarda kompetencije. Govorićemo redom o svakom od tih napora.

Obrazovanje u oblasti softverskog inženjerstva

Istorijski posmatrano, softversko inženjerstvo se smatralo relativno uskim aspektom računarskih nauka. U najboljem slučaju, univerzitetski program iz računarskih nauka je uključio kurs iz softverskog inženjerstva, obično na prvoj ili drugoj godini. U najmanju ruku, on je obuhvatio nekoliko predavanja o značenju i nameni softverskog inženjerstva. Sa povećanjem naglaska na kvalitetu i bezbednosti softvera i sa nedavnim impulsom da se profesionalizuju projektanti softvera, univerziteti su sve više gledali na softversko inženjerstvo kao na posebnu akademsku disciplinu. Ima mnogo načina na koje studenti mogu da se kreću kroz univerzitetski program i završe sa obrazovanjem iz softverskog inženjerstva:

- specijalizujući se u softverskom inženjerstvu, kao delu glavnog smera računarskih nauka;
- specijalizujući se u softverskom inženjerstvu, kao delu glavnog smera računarskog inženjerstva;
- specijalizujući se u softverskom inženjerstvu, kao delu glavnog smera inženjerstva;
- završivši glavni smer softversko inženjerstvo, sa posebnom diplomom različitom od računarskih nauka ili računarskog inženjerstva.

Kao što je prikazano u tabeli 14.6, program softverskog inženjerstva pozajmljuje predmete i od programa računarskih nauka i od programa inženjerstva. Kursevi računarskih nauka obezbeđuju tehnička i ekspertska znanja koja su potrebna da se softverski proizvodi izgrade i da se njima upravlja: principi programiranja; strukture podataka; paradigme algoritama; programski jezici; gramatike, sintaksna analiza, prevođenje; konkurentnost i upravljačke strukture i teoretske granice izračunljivosti. Pored toga, kursevi računarskih nauka i računarskog inženjerstva obuhvataju ključne komponente sistema (na primer, računarsku arhitekturu, operativni sistem, srednji sloj, mreže) i ključne softverske komponente (na primer, baze podataka, korisničke interfejse) i kako dizajn i implementacija tih komponenti može da utiče na performansu, upotrebljivost i kvalitet softvera.

Iz inženjerstva, program softverskog inženjerstva pozajmljuje i prilagođava principe i prakse koji se odnose na velike sisteme i koji su namenjeni pravljenju uspešnih proizvoda:

- rano planiranje i razvojne aktivnosti koje pomažu da se greške otkriju rano u procesu razvoja, kada je njihova popravka jeftinija i jednostavnija;
- sistematske, predvidljive procese projektovanja i razvoja koji pomažu da se obezbedi da se softverski proizvod razvije ekonomično i da odgovara nameni;
- razmatranje nefunkcionalnih osobina, kao što su performansa, mogućnost održavanja, upotrebljivost, ekonomičnost i vreme do pojave na tržištu, koje često određuju da li je softverski proizvod prihvatljiv.

Tradicionalno inženjersko obrazovanje takođe uključuje solidne osnove iz matematike i nauke i izlaganje drugim inženjerskim disciplinama. Naučna osnova je važna zato što inženjeri moraju da shvate kako da reaguju na fizički svet i kako njime da upravljaju. Izlaganje drugim inženjerskim disciplinama pomaže inženjeru da razume inženjere iz drugih oblasti i da uspešnije radi sa njima kao i da gradi komponente koje bolje sarađuju sa komponentama iz drugih tehnika.

TABELA 14.6 Softversko inženjerstvo obuhvata i računarske nauke i inženjerstvo

Računarske nauke	Inženjerstvo
Upravljanje podacima	Disciplinovani procesi
Šabloni podataka	Veliki, integrisani sistemi
Transformacija podataka	Koordinirani timovi
Algoritamske platforme	Nefunkcionalne osobine (npr. performansa, pouzdanost, mogućnost održavanja, lakoća upotrebe)
Programski jezici	
Interakcija čovek-računar	

Kursevi softverskog inženjerstva se grade na temeljima računarstva i inženjerstva i uvode koncepte koje mi objašnjavamo u ovoj knjizi, uključujući specifikaciju zahteva, principe projektovanja softvera, tehničku dokumentaciju, upravljanje projektom, testiranje i verifikaciju. Pored toga, program softverskog inženjerstva će verovatno uključiti netehničke kurseve iz komunikacije, humanih i društvenih nauka. Projekti softverskog inženjerstva su skloni da obuhvate veliki broj ljudi, pa softverski inženjeri moraju da razumeju dinamiku grupne interakcije da bi znali kako da motivišu pojedince da slede zajednički cilj. Oni treba da razmotre uticaj njihovih tehnologija na korisnike i na društvo u celini i mogu da donose odluke koje uzimaju u obzir ljudske i društvene vrednosti. Oni takođe treba da rade sa specijalistima iz drugih disciplina koji imaju različite nivoe softverskih ekspertskih znanja. Taj rad zahteva jaku komunikaciju, poslovne veštine i veštine rasuđivanja.

Kao takvo, softversko inženjerstvo je opštije, primenljivije i ima širi opseg od računarskih nauka. U svom jezgru, računarske nauke se usredsređuju na podatke, transformaciju podataka i algoritme; napredniji kursevi predstavljaju tehnike projektovanja i programiranja za specifične domene primene. Nasuprot tome, softversko inženjerstvo se usredsređuje na izgradnju softverskih proizvoda. Ono posmatra sve aktivnosti obuhvaćene razvojem softverskog sistema, od početne zamisli do krajnjeg proizvoda. Štaviše, koncepti projektovanja u softverskom inženjerstvu teže da se usredsrede na principe projektovanja opšte namene, oblike i kriterijume; napredniji kursevi predstavljaju tehnike projektovanja i analize koji se primenjuju na velike softverske sisteme.

Mada postoji iskušenje da se pokuša klasifikovanje softverskog inženjerstva kao i svake druge grane inženjerstva, to „prirodno" preslikavanje sadrži fatalnu grešku: dok većina inženjera proizvodi fizičke produkte koji se razmeštaju u stvarnom svetu i podležu fizičkim zakonima, softverski inženjeri prave softverske proizvode koji se razmeštaju u računarima (koji su veštački svet) i podležu zakonima i ograničenjima računarstva. Ono što se najviše zapaža, fizički proizvod ima kontinualno ponašanje, što znači da možemo da ispitujemo proizvod u jednoj situaciji i ekstrapoliramo kako će se on ponašati u odgovarajućim situacijama. Softverski proizvodi nemaju kontinualno ponašanje, što znači da slični ulazi mogu za rezultat da imaju veoma različite izlazne vrednosti. Takođe, softver se implementira koristeći promenljive diskretnih vrednosti, a uticaj koji se predstavlja pomoću takvih promenljivih može da se nagomilava u dugim računanjima. Greške „isključeno zbog jednog" u drugim inženjerskim disciplinama su bez posledica, ali takve greške u softveru su prava propast. Prema tome, mnoge od tradicionalnih tehnika za modelovanje i procenu iz standardnog inženjerskog arsenala ne odgovaraju za softver. Fleksibilnost i slojevitost softvera impliciraju njegovu složenost; čak i male promene mogu da imaju velike efekte, pa su projektovanje i testiranje često daleko teži nego u drugim domenima inženjerstva.

Osnovne studije softverskog inženjerstva ne mogu da budu sveobuhvatne, zato što postoji više potrebnih kurseva nego što ih je moguće ostvariti u četvorogodišnjem programu. Idealno, savršen program bi uključio ozbiljnu dubinu računarskih nauka, dovoljno diskretne matematike za softver za modelovanje i analizu, tradicionalna inženjerska ekspertska znanja u kontinualnoj matematici i nauci i znanja i tehnike softverskog inženjerstva, dok bi i dalje održavao deljeno iskustvo sa drugim granama inženjerstva. Dakle, kreatori programa softverskog inženjerstva su morali da naprave teške izbore u odlučivanju kako da kombinuju predmete da bi uobličili odgovarajući i izvodljiv program za osnovne obrazovne studije softverskog inženjerstva.

U isto vreme, ljudi ulaze u softversku industriju preko neakademskih putanja, kao što su učenje programiranja posle studiranja neke druge discipline kao što je poslovanje, fizika, ili matematika. S obzirom na to da praktičari sa tako raznovrsnim akademskim osnovama i radnim iskustvima sebe nazivaju „softverskim inženjerima", došlo je do nedavnog impulsa da se ustanove minimalni zahtevi za takvo označavanje. Kao rezultat, pojavio se izvestan broj inicijativa da se definiše osnovni „skup znanja" iz domena softverskog inženjerstva: detaljan opis onoga što bi neko trebalo da zna da bi mogao da se nazove softverskim inženjerom.

Korpus znanja softverskog inženjerstva

Namera iza definisanja osnovnog skupa znanja je da se unutar zajednice softverskog inženjerstva ustanovi sporazum o znanju, veštinama i ekspertnosti koje bi svaki softverski inženjer trebalo da poseduje. Neke od inicijativa za definisanje osnovnog skupa znanja iz domena softverskog inženjerstva su:

- *Computing Curricula – Software Engineering (SE2004) (IEEE-CS i ACM 2004).* Taj zajednički napor Instituta za inženjere elektrotehnike i elektronike - Računarsko društvo (IEEE-CS) i Udruženja za računarske mašine (ACM) namenjen je da ponudi smernice kako da se projektuje program za osnovne studije i specijalizacije iz softverskog inženjerstva. Njegovi saveti obuhvataju osnovni skup znanja, koji se zove Software Engineering Education Knowledge (SEEK), a sadrži suštinska i poželjna znanja i veštine koje bi svaki smer softverskog inženjerstva trebalo da pokuša da uključi u svoj nastavni plan. Njegov osnovni skup znanja pored softverskog inženjerstva uključuje i poznavanje disciplina koje su sa njim u vezi (na primer, matematike, računarskih nauka, inženjerstva, ekonomije).
- *Software Engineering Body of Knowledge (SWEBOK) (IEEE-CS 2004).* Taj napor, koji podržava IEEE-CS i jedan broj industrijskih partnera, namenjen je definisanju šta praktični profesionalac u domenu softverskog inženjerstva treba da zna. On se prvenstveno usredsređuje na znanja iz oblasti softverskog inženjerstva i znanja koja praktičar treba da prikupi iz kombinacije akademskih studija i četiri godine radnog iskustva.
- *Software Engineering Syllabus (CEQB 1998).* Taj napor organizacije Canadian Engineering Qualifications Board (CEQB) nudi smernice lokalnim inženjerskim udruženjima kako da ocene akademska uverenja podnosilaca zahteva koji žele da postanu licencirani profesionalni inženjeri. Njegov osnovni skup znanja je *de facto* skup nacionalnih smernica za procenjivanje osnovnih studija podnosilaca zahteva koji nisu diplomirali na akreditovanom kanadskom inženjerskom programu. Svako provincijsko inženjersko udruženje definiše sopstvene kriterijume specifične za disciplinu za procenjivanje kandidata koji imaju neinženjersku akademsku osnovu, ali koriste programe CEQB kao ulaze za definisanje kriterijuma.

S obzirom na to da ti napori imaju različite ciljeve (na primer, jezgro osnovnih studija prema znanju koje se očekuje od ustanovljenog profesionalca), oni imaju i različita očekivanja širine i dubine znanja i veština koje sačinjavaju osnovni skup znanja iz domena softverskog inženjerstva. U tabeli 14.7 prikazan je pogled visokog nivoa na znanje u obrazovanju iz softverskog inženjerstva (SEEK) (IEEE-CS i ACM 2004) i *minimalni* broj časova koji treba da se posvete svakoj oblasti i jedinici znanja. Kao što može da se

vidi, SEEK uključuje ne samo principe za koje se smatra da su proverene smernice za dobru praksu, nego i tehničko znanje iz računarskih nauka, inženjerstva i softverskog inženjerstva koje podržava i pojačava principe. Drugi delovi SEEK (koji nisu prikazani) uzimaju jedinice znanja prikazane u tabeli i razlažu ih na pojedinačne teme. Svaka tema se ocenjuje kao suštinska, poželjna ili opciona za osnovni program i označava se stepenom u kome se od studenata očekuje da njom ovladaju (na primer, da se podsete tog znanja, razumeju ga, ili primene u rešavanju problema). Na primer, u okviru modelovanja i analize softvera, jedinica znanja osnovi modelovanja se razlaže na sledeće teme:

TABELA 14.7 SEEK oblasti i jedinice znanja (© IEEE-CS i ACM 2004)

Naslov	časova	Naslov	časova
Suština računarstva	172	**Verifikacija i validacija softvera**	42
Osnovi računarskih nauka	140	Terminologija i osnovi verifikacije i validacije	5
Tehnologije konstrukcije	20	Pregledi	6
Alati za konstrukciju	4	Testiranje	21
Formalni metodi za konstrukciju	8	Testiranje i procena korisničkog interfejsa u interakciji čovek-računar	6
Matematički i inženjerski osnovi	89	**Evolucija softvera**	10
Matematički osnovi	56	Procesi evolucije	6
Inženjerski osnovi softvera	23	Aktivnosti evolucije	4
Inženjerska ekonomija softvera	10		
Profesionalna praksa	35	**Softverski proces**	13
Dinamika grupe/psihologija	5	Koncepti procesa	3
Komunikacione veštine (specifične za SI)	10	Implementacija procesa	10
Profesionalizam	20		
Modelovanje i analiza softvera	53	**Kvalitet softvera**	16
Osnovi modelovanja	19	Koncepti i kultura kvaliteta softvera	2
Vrste modela	12	Standardi kvaliteta softvera	2
Osnovi analize	6	Procesi kvaliteta softvera	4
Osnovi zahteva	3	Osiguranje procesa	4
Formulisanje zahteva	4	Osiguranje proizvoda	4
Specifikacija i dokumentovanje zahteva	6		
Validacija zahteva	3		
Projektovanje softvera	45	**Upravljanje softverom**	19
Koncepti projektovanja	3	Koncepti upravljanja	2
Strategije projektovanja	6	Planiranje projekta	6
Projektovanje arhitekture	9	Osoblje i organizacija projekta	2
Projektovanje interfejsa čovek-računar	12	Kontrola projekta	4
Detaljno projektovanje	12	Upravljanje konfiguracijom softvera	5
Alati za podršku projektovanju i proceni	3		

1. Principi modelovanja (na primer, apstrakcija, dekompozicija, pogledi)
2. Pred i postuslovi i invarijante
3. Matematički modeli i jezici za specifikaciju
4. Osobine jezika za modelovanje
5. Razlika između sintakse i semantike notacija
6. Značaj modela koji objašnjavaju sve informacije

Sve te teme su ocenjene kao suštinske, ali samo prvom od njih treba da se ovlada do nivoa „primene", na kome bi studenti trebalo da znaju kako da primene te principe modelovanja u konstruisanju sopstvenih modela.

Ne podrazumeva se da je osnovni skup znanja potpun. On radije prikuplja prakse i principe koji su primenljivi na većinu softverskih proizvoda, mogu da se na odgovarajući način objasne na nivou osnovnih studija i obezbeđuju izvesno osiguranje kvaliteta. Međutim, primena tih praksi i principa ne garantuje da će rezultat biti savršen softver. U tom smislu, osnovni skup znanja odražava najbolju poznatu praksu, a ne savršenstvo.

Licenciranje softverskih inženjera

Kao što smo ranije pomenuli, jedna od namera koja stoji iza definisanja osnovnog skupa znanja je da se formira osnova za procenjivanje kvalifikacija softverskih inženjera praktičara. Ta procena može da bude u obliku licenciranja ili sertifikacije. **Licenciranje** je zakonsko ograničenje o tome kome je dozvoljeno da radi u zakonom regulisanoj profesiji, a proces licenciranja obično uključuje procenu znanja i sposobnosti kandidata. Na primer, u najvećem delu severne Amerike, provincijske i državne uprave propisuju licenciranje nekih profesionalaca, kao što su lekari, pravnici i profesionalni inženjeri, od kojih se zakonski zahteva da rade na nivou u saglasnosti sa javnom bezbednošću i dobrobiti. Nasuprot toga, **sertifikacija** je dobrovoljna procena koju praktičar može da odabere da prođe, da bi pokazao kompetenciju. O sertifikaciji softverskih inženjera govorimo u sledećem odeljku.

Većina zemalja reguliše praksu u nekim profesijama, da bi zaštitila javnost od nekvalifikovanih praktičara. Regulisane profesije su one koje mogu da utiču na javnu bezbednost (kao što su lekari, farmaceuti, inženjeri, vozači kamiona) ili one koje prodaju svoje usluge direktno javnosti (kao što su advokati, računovođe). S obzirom na to da softver može da utiče na zdravlje i dobrobit i zato što projektanti softvera često prodaju svoje usluge i proizvode klijentima koji ne mogu da procene njihove kvalifikacije, mnogi tvrde da softverska profesija treba da se zakonski reguliše. Ta priča o regulaciji dovodi do pitanja ko bi tačno trebalo da se licencira i kako. Udruženja inženjera tvrde da je softversko inženjerstvo grana inženjerstva i da bi praktičari u toj oblasti trebalo da se licenciraju kao profesionalni inženjeri (američka ili kanadska titula) ili kao dokumentovani inženjeri (britanska titula). U suštini, oni tvrde da načini i autoritet za licenciranje softverskih inženjera već postoje, u obliku mnogih državnih odbora i udruženja profesionalnih inženjera, koji trenutno licenciraju profesionalne inženjere. Računarska udruženja tvrde da je softversko inženjerstvo suviše različito od tradicionalnih grana inženjerstva za svoje praktičare da bi se oni procenjivali prema kriterijumima inženjerstva i da bi trebalo da se stvore drugačije organizacije ili ovlašćeni autoritet da reguliše softversku profesiju. Dalje u tekstu, dajemo pregled kriterijuma za licenciranje softverskih inženjera u SAD i Ka-

nadi, a zatim ponovo razmatramo argumente za licenciranje profesionalnih softverskih inženjera i protiv njega.

U Severnoj Americi, inženjersku profesiju regulišu države, provincije i teritorije (ovde ćemo ih sve zvati „državama"; to znači, svaka država je donela zakone koji označavaju autoritativno telo (na primer, odeljenje vlade, nezavisna državna agencija, ili samoregulišuće profesionalno udruženje) koje reguliše inženjersku profesiju u toj jurisdikciji. Normalno, profesionalni inženjer ne može da radi u državi bez licence koju mu je izdala njegova regulišuća ustanova, iako većina tih ustanova ima recipročne sporazume koji dozvoljavaju licenciranom inženjeru da radi lokalno na privremenoj osnovi. (Taj reciprocitet je uporediv, mada ne baš isti, sa vozačkom dozvolom, koja se izdaje u državi u kojoj vozač živi, ali se priznaje u ostalim državama.)

U SAD, potreba za licencom zavisi od države u kojoj inženjer radi. Uopšte, licenciranje je obavezno za svakog profesionalca koji nudi inženjerske usluge direktno javnosti i koji učestvuje u projektovanju struktura i proizvoda (odnosno, puteva, građevina, nuklearnih postrojenja) čiji projekti i planovi moraju da se podnesu na odobrenje državnim agencijama. Većina inženjera iz SAD nisu licencirani. Oni često rade za kompaniju ili za saveznu vladu, pa ne nude svoje usluge direktno javnosti. Ta razlika rešava pitanje odgovornosti; kada inženjer projektuje proizvod ili pruži uslugu koja nije uspela, on ne mora da ima licencu ako je poslodavac odgovoran za efekte otkaza. Štaviše, većina inženjera ne koristi titulu „inženjer" izvan posla, za razliku od lekara, kojima se uvek obraća sa „doktor".

U Kanadi, svaki inženjer koji radi mora da bude licenciran kao profesionalni inženjer. Mnoge jurisdikcije definišu „inženjerstvo" po *praksi* (šta inženjer radi) a ne po *tituli* (kako inženjer sebe naziva). U Ontariju se „praksa profesionalnog inženjerstva" definiše u **Aktu profesionalnih inženjera** i podrazumeva tri testa:

- „svaki čin projektovanja, sastavljanja, procenjivanja, davanja saveta, izveštavanja, upravljanja ili nadgledanja;
- gde se stara o očuvanju života, zdravlja, imovine ili javne dobrobiti i
- zahteva primena inženjerskih principa, ali ne uključuje rad u svojstvu prirodnog naučnika" (Ontario 2000).

Dakle, svako ko je odgovoran za inženjerski rad koji bi mogao da utiče na bezbednost, zdravlje ili dobrobit javnosti bavi se inženjerstvom. Svako ko obavlja inženjerstvo bez stalne ili privremene licence kriv je zbog prekršaja i provincija može da ga kazni.

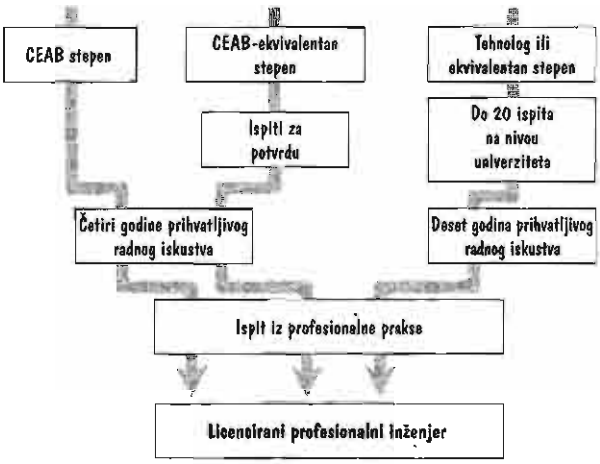
Zakoni o licenciranju u SAD i Kanadi se retko primenjuju u pogledu softverskih inženjera. Do tako labavog odnosa dolazi delimično zato što se primena zakona često pokreće putem pritužbe, pa se kršenja zakona ne otkrivaju dok se ne dovedu u pitanje kvalifikacije praktičara. On takođe odražava nedostatak podrške za licenciranje od strane vlade i zajednice razvoja softvera, a taj otpor će se verovatno nastaviti sve dok postoji nedovoljan broj softverskih inženjera na raspolaganju da popuni potrebe tekućih poslova.

I u SAD i u Kanadi profesionalni inženjer se procenjuje unutar određene inženjerske discipline, ali on ili ona se licenciraju kao opšti „inženjer" bez ograničenja u tituli. Sve-

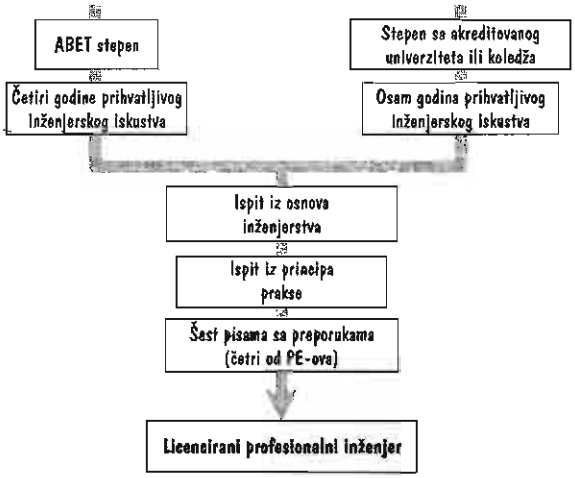
jedno, profesionalni inženjer koji radi izvan oblasti svoje kompetencije, može da bude predmet disciplinske akcije regulišućeg odbora ili profesionalnog udruženja. Ta situacija je analogna onoj u medicinskoj profesiji, gde se doktor licencira kao opšti „doktor medicine" (MD), ali mu se dozvoljava da radi samo unutar specifične specijalističke oblasti kao što je hirurgija, ili interna medicina.

Na slici 14.10 ilustrovana su tri puta da se postane profesionalni inženjer u Kanadi. Prvi korak u procesu je da se zadovolje akademski zahtevi. Izbor puta zavisi od toga koliko specijalističkih znanja se dobilo na akademskim studijama, nasuprot radnom iskustvu. Staza krajnje levo se oslanja na diplomiranje na inženjerskom programu koji je akreditovao Canadian Engineering Accreditation Board (CEAB), odbor organizacije Canadian Council of Profesional Engineers. Univerziteti akredituju svoje inženjerske programe da bi obezbedili da diplomci počnu svoj profesionalni život sa obrazovanjem potrebnim da bi efektivno radili. Diplomiranje na akreditovanom inženjerskom programu je najlakši način zadovoljavanja akademskih zahteva za licenciranje. Srednja staza uključuje diplomiranje na ekvivalentnom univerzitetskom programu (odnosno, programu primenjenih nauka ili inženjerskom programu u drugoj zemlji), gde „ekvivalentnost" mora da se pokaže na ispitima za potvrđivanje. Staza krajnje desno zahteva uspešno ispunjavanje specijalnog ispitnog programa, koji obuhvata do 20 ispita.

Drugi korak je da se zadovolje zahtevi inženjerskog iskustva. Oni koji tek postaju profesionalci treba da primenjuju svoja znanja i veštine u praksi pre nego što postanu spremni da preuzmu glavnu odgovornost za izvođenje radova u svojoj oblasti, pa sve tri staze do licence kanadskog profesionalnog inženjera zahtevaju inženjersko iskustvo. Staza koja se oslanja na strogu akademsku osnovu zahteva četiri godine inženjerskog iskustva. Kandidat koji nije diplomirao na akreditovanom inženjerskom ili ekvivalentnom programu, mora da prikupi deset godina iskustva pre nego što se razmatra za licencu.



SLIKA 14.10 Putevi kako se postaje profesionalni inženjer (P.Eng.) u Kanadi



SLIKA 14.11 Proces aplikacije za profesionalnog inženjera (PE) u SAD

Do jedna godina kredita daje se za postdiplomske studije ili vojno iskustvo. (Nasuprot tome, u SAD se od lekara zahteva da imaju tri godine nadgledanog iskustva, što se naziva rezidentnošću pre nego što mogu da se licenciraju, a sertifikovane javne računovođe moraju da rade jednu godinu za organizaciju koju je ovlastio odbor pre nego što dobiju svoje licence.) Najzad, svi kandidati moraju da pišu i polože ispit iz profesionalne prakse koji obuhvata relevantni lokalni zakon, profesionalnu praksu, etiku i pravnu obavezu.

Proces licenciranja u SAD je teži, u tome što svi kandidati moraju da prođu kroz ispite potvrđivanja. Štaviše, nema staze koja se zasniva samo na iskustvu (slika 14.11). Svi profesionalni inženjeri moraju da imaju akademske kvalifikacije, najbolje one koje uključuju diplomiranje na akreditovanom programu organizacije Accreditation Board for Engineering and Technology (ABET) ili njegovom ekvivalentu. (Program koji je akreditovao CEAB smatra se prihvatljivim ekvivalentom.)

Podnosiocima molbe koji imaju stepen sa programa koji je akreditovao ABET zahteva se četiri godine inženjerskog iskustva; ostalim podnosiocima se zahteva osam godina iskustva. Iskustvo mora da pokaže jasnu upotrebu inženjerskog znanja, obrazovanja i prosuđivanja, pokazujući na taj način da je kandidat kompetentan da preuzme odgovornost inženjerskog rada. Rad mora takođe da bude progresivan, što znači da u toku vremena postoji sve veći standard kvaliteta i odgovornosti u jednoj dominantnoj disciplini. Mada je rad pod nadgledanjem licenciranog profesionalnog inženjera preporučljiv, to nije zahtev za dobijanje licence.

Bez obzira na to kojom stazom ide do licence, kandidat mora da prođe osmočasovni ispit iz osnova inženjerstva. Taj ispit se obično polaže za vreme studentove starije godine na univerzitetu, zato što se ispit vrlo teško prolazi ako se to radi znatno kasnije od završavanja odgovarajućih kurseva. Prepodnevni deo ispita obuhvata materijal zajednički za sve inženjerske discipline, obično iz prve dve godine programa koji je akreditovao ABET, uključujući hemiju, računare, dinamiku, električna kola, ekonomiju inženjerstva, etiku, mehaniku fluida, nauku o materijalima, matematiku, mehaniku materijala, statiku

i termodinamiku. Namera iza ovog zajedničkog ispita je da se obezbedi da svaki inženjer zna nešto o drugim inženjerskim disciplinama, da bi se olakšalo komuniciranje između inženjera iz različitih disciplina; to znanje takođe pomaže da se obezbedi da inženjer ceni nivo ekspertize potreban u različitim organizacijama, tako da ne pokušava da radi u specijalnosti za koju nije obrazovan.

Na popodnevnom delu ispita testira se kompetencija u posebnoj disciplini, što odražava poslednje dve godine programa koji je akreditovao ABET. Postoje posebni testovi specifični za discipline za građevinsko, hemijsko, industrijsko, mašinsko i elektro inženjerstvo, ili kandidati mogu da polažu drugi opšti test koji pokriva isti materijal kao i prepodnevni ispit, samo sa više detalja. Namera iza ispita specifičnog za disciplinu je da testira kompetenciju kandidata u oblasti specijalizacije. U trenutku pisanja ovog teksta, nije postojao ispit specifičan za disciplinu softverskog inženjerstva; nema čak nijednog iz računarskog inženjerstva. Test specifičan za disciplinu pravi se samo kada postoji više od 100 programa koje je akreditovao ABET u toj disciplini, tako da se ne očekuje nikakav ispit iz softverskog inženjerstva u bliskoj budućnosti.

Posle ne više od četiri godine iskustva, kandidat za licencu mora da položi drugi ispit, koji se ovog puta odnosi na principe i praksu inženjerstva u predmetu specifičnom za disciplinu. Tom prilikom takođe nema ispita iz principa prakse softverskog inženjerstva.

Pokret ka licenciranju softverskih inženjera se pojavljuje na razne načine. U Kanadi, neke provincije, uključujući Ontario, Britansku Kolumbiju i Albertu, licenciraju softverske inženjere kao profesionalne inženjere. U SAD, Teksas i Tenesi zahtevaju dodelu zvanja profesionalnog inženjera pre nego što praktičar može sebe da nazove „inženjerom", uključujući tu i softverske inženjere. Međutim, druge jurisdikcije ne rade ništa. A u Velikoj Britaniji, neko može da bude imenovan za dokumentovanog inženjera jednostavno kada se pridruži organizaciji British Computer Society.

Još uvek nema snažnog momenta za podsticanje licenciranja. Razlog ide dalje od nedostatka projektanata softvera i uključuje razlike u mišljenju o stanju u oblasti softverskog inženjerstva: da li mi, kao profesija, znamo kako da razvijemo softver koji će se potvrditi kao korektan i pouzdan softver? Pristalice iznose više argumenata za licenciranje softverskih inženjera:

- Praksa softverskog inženjerstva pada pod statute kao što je Professional Engineers Act koji zahteva vladino ili institucionalno uverenje za praksu (u Kanadi) ili za prodaju usluga javnosti (u SAD).
- Licencirani softverski inženjeri bi poboljšali kvalitet softvera podizanjem prosečne kompetencije praktičara.
- Licenciranje bi podstaklo projektante softvera da dobiju solidnu obrazovnu osnovu za svoju praksu.
- Licenciranje bi podstaklo da najbolje prakse koriste čak i oni koji nisu licencirani.
- Licenciranje bi poboljšalo inženjerstvo softvera i obrazovanje softverskih inženjera, unapređujući na taj način profesionalizaciju softverskog inženjerstva.

Međutim, neki praktičari tvrde da profesija nije (i može nikada da neće ni biti) spremna za licenciranje. Njihovi razlozi uključuju sledeće:

- Nema dokaza da licencirani softverski inženjeri proizvode i održavaju najbolji softver.
- Licenciranje može da pruži lažnu sigurnost javnosti da je softver koji su napravili profesionalci visokog kvaliteta.
- Nema široko prihvaćenog osnovnog skupa znanja čijim bi se ovladavanjem definisala kompetencija u softverskom inženjerstvu.
- Javna bezbednost bi se najbolje osigurala sertifikovanjem proizvoda, a ne procesa ili proizvođača. Drugim rečima, dokaz valjanosti pudinga je kada se on jede.

Pored toga, računarski naučnici i inženjeri nastavljaju da se prepiru oko ograničenja kome bi trebalo da se dozvoli da razvija softver. Na primer, licenciranje u Kanadi se zahteva kada se primenjuju principi inženjerstva, ali gde se zaustavljaju računarske nauke a gde počinje softversko inženjerstvo? Verovatno će licenciranje na kraju biti primenjeno samo na one projektante softvera koji grade i održavaju bezbedno kritične sisteme ili sisteme sa kritičnom misijom, kao što su medicinski uređaji, softver za nuklearna postrojenja i softver koji uključuje modelovanje fizičkog sveta. Međutim, da li bi diplomiranje na akreditovanom programu računarskih nauka trebalo smatrati dovoljnom akademskom osnovom za licenciranje? Pored toga, koji softver bi trebalo smatrati kritičnim? Greške u naizgled nevažnom telefonskom softveru mogu da obore kritičnu informacionu infrastrukturu, a otkazi u običnom bankarskom softveru mogu da imaju štetne ekonomske efekte.

Teško je predvideti u kom pravcu će ići softversko licenciranje u bližoj i daljoj budućnosti. Potražnja dobro kvalifikovanih softverskih specijalista će verovatno prevazilaziti dobijanje licenciranih softverskih inženjera u predvidljivoj budućnosti. Vlade i institucije ne mogu da postanu ozbiljne u pogledu licenciranja softverskih profesionalaca dok javnost ne bude zahtevala softver boljeg kvaliteta. Međutim, licenciranje verovatno neće čekati da softverska praksa sazri do te mere da možemo da garantujemo uspeh i bezbednost naših proizvoda. U tom pogledu, zakonska odgovornost softverskih inženjera može na kraju da bude kao ona koju imaju lekari i advokati, gde će se od nas očekivati da sledimo „najbolju praksu", ali ne i da obavezno proizvodimo najbolji softver.

U međuvremenu, vlade mogu da ozakone uloge koje izdvajaju računarske naučnike od softverskih inženjera, vrlo slično tome kako se arhitekti izdvajaju od građevinskih inženjera. Praktičari mogu da se izuzmu od propisa kao što je Professional Engineers Act, ako sistemi koje oni prave imaju malo međusobnog dejstva sa fizičkim svetom, kao što su finansijske, informacione, poslovne, naučne i zabavne aplikacije. Pored toga, pojedincima koji imaju deset ili više godina specijalizovanog iskustva može da se izda ograničena licenca; onima koji imaju takvu licencu moglo bi da se dozvoli da rade u uskom aspektu inženjerstva (uskom u pogledu i discipline i aktivnosti) u kome mogu da pokažu kompetenciju.

Sertifikacija

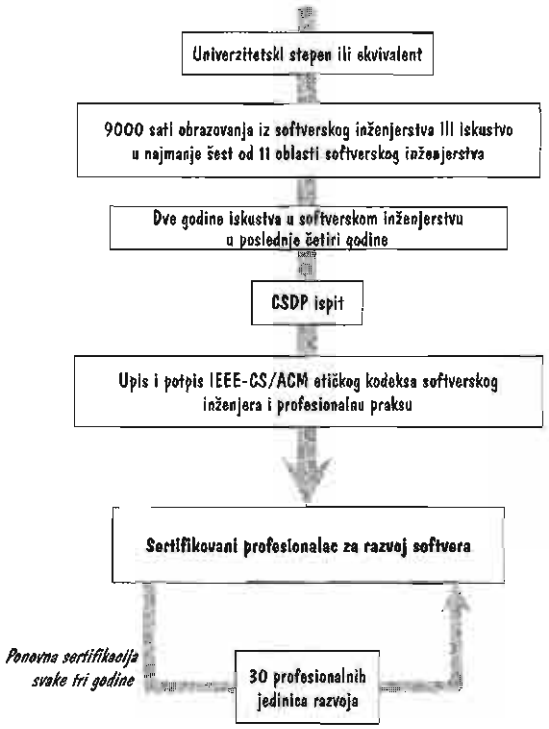
Sertifikacija se razlikuje od licenciranja. **Licenca** je zakonski dokument kojim se potvrđuje da je onaj ko je ima ispunio uslove koje je postavila vlada. Sertifikacija je davanje

dokumenta kojim se potvrđuje praktičareva kompetencija na osnovu položenih ispita, godina iskustva ili i jednog i drugog; ona je namenjena da podstakne javno poverenje u profesionalca koji je poseduje. Licenca se zahteva za rad; sertifikat je dobrovoljan.

Sertifikacijom često upravlja i daje je profesionalno udruženje. Na primer, IEEE Computer Society nudi sertifikaciju za sertifikovanog profesionalca za razvoj softvera (CSDP), a Computer Information Processing Society (CIPS) ima sertifikat za profesionalca u informacionim sistemima (ISP). Oba sertifikata su namenjena da pokažu da praktičar zadovoljava minimum akademskih i praktično iskustvenih kvalifikacija, i oba zahtevaju ponovnu sertifikaciju svake tri godine. Da bi se ponovo sertifikovali, praktičari moraju da pokažu da su oni svoje veštine održali ažurnim, kroz kurseve, objavljivanje, konferencije, strukovne časopise itd.

Proces sertifikacije udruženja Certified Software Development Society, prikazan na slici 14.12, zahteva da kandidati imaju univerzitetski stepen i da imaju dovoljno iskustva (9000 sati) proučavanja ili primene koncepata i tehnika softverskog inženjerstva. To iskustvo mora da se stekne u najmanje šest od sledećih jedanaest oblasti:

- Profesionalizam i ekonomija inženjerstva
- Softverski zahtevi
- Projektovanje softvera
- Konstrukcija softvera

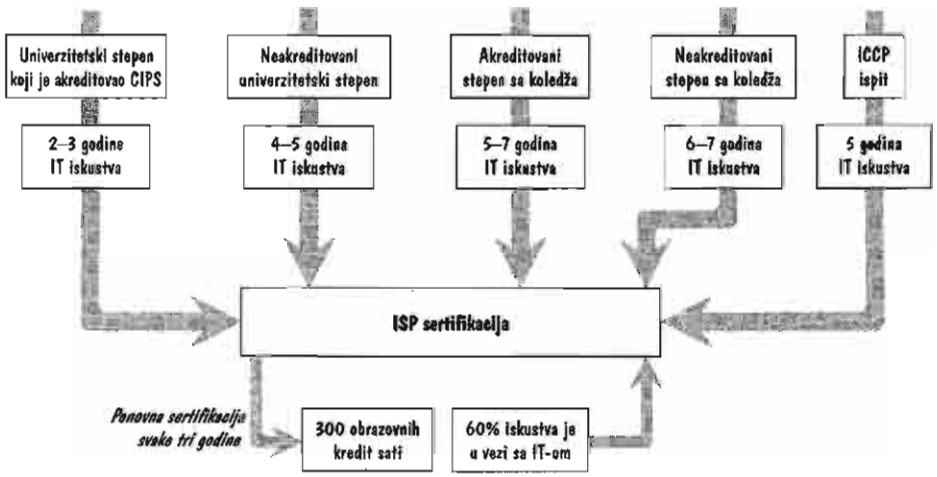


SLIKA 14.12 Proces sertifikacije profesionalca za razvoj softvera

- Testiranje softvera
- Održavanje softvera
- Upravljanje inženjerstvom softvera
- Upravljanje konfiguracijom softvera
- Proces softverskog inženjerstva
- Metodi i alati softverskog inženjerstva
- Kvalitet softvera

Pored toga, kandidat mora da položi ispit koji pokriva navedenih 11 oblasti i mora da ima skorašnje radno iskustvo na polju povezanom sa softverskim inženjerstvom. Da bi ostao sertifikovan, potrebno je da dokumentuje profesionalne jedinice razvoja (PDU) koje su obavljene u poslednje tri godine. PDU mogu da se zatraže za obrazovne aktivnosti (na primer kurseve, konferencije, seminare), napisane članke ili knjige, date prezentacije, tehničke ili profesionalne usluge, sopstvene studije ili aktivnosti zapošljavanja. Profesionalac koji traži ponovnu sertifikaciju posle roka od tri godine, može biti izložen ponovnom polaganju CSDP ispita.

Proces ISP sertifikacije se dosta razlikuje. On zahteva iskustvo koje koristi značajan nivo znanja o informacionoj tehnologiji (IT) i da se poseduje visok nivo nezavisnog rasuđivanja i odgovornosti. Na slici 14.13 prikazane su različite kombinacije zahteva stepena i iskustva koje vode do ISP sertifikacije. Kandidat koji nema obrazovanje na nivou univerziteta ili koledža u oblasti u vezi sa IT-om, može da zadovolji zahteve stepena položivši ispit koji nudi Institute for Certification of Computer Professionals (ICCP). Ispit ICCP uključuje pitanja o osnovnim predmetima, specijalnostima i programskim jezicima. ISP sertifikacija zahteva od svojih profesionalaca da prođu kroz ponovnu sertifikaciju svake tri godine. Kandidati moraju da pokažu da njihov rad nastavlja da bude povezan sa IT-om. Kandidati takođe moraju da pokažu stalne napore da nadgrađuju znanje IT-a kroz predavanja, konferencije, čitanja tehničke literature, objavljivanje, mentorstvo itd.; ti napori moraju da se sumiraju u najmanje 300 sati profesionalnog razvoja u trogodišnjem periodu.



SLIKA 14.13 Proces sertifikacije ISP

Profesionalna sertifikacije ne bi trebalo da se pomeša sa sertifikatima koje nude softverske kompanije. Na primer, i Microsoft i Cisco imaju obimne programe sertifikacije za procenjivanje korisnikove kompetencije u instaliranju i upotrebi njihovih operativnih sistema i mrežnih programa. Takve sertifikacije specifične za proizvod ili proizvođača mogu da pokažu korisnikovu veštinu u korišćenju određenog proizvoda kompanije, ali ne potvrđuju korisnikovo opšte znanje kao softverskog profesionalca.

Etički kodeks

Inženjere, a posebno softverske inženjere, ponekad kritikuju da su toliko usredsređeni na tehnički problem da zaboravljaju ljudski kontekst u kome je on ugrađen. Iz tog razloga, etički kodeks može da bude koristan, ako ne i od suštinske važnosti, u obezbeđenju da razumemo i prihvatamo implikacije naših profesionalnih radnji i aktivnosti. U etičkom kodeksu opisuju se etičke i profesionalne obaveze prema kojima pojedinci, javnost i zakonska tela mogu da mere ponašanje profesionalca. Kodeks ima tri glavne funkcije:

- On stimuliše etičko ponašanje.
- On podstiče javno poverenje u našu profesiju.
- On nudi formalnu osnovu za procenjivanje akcija i disciplinovanje profesionalaca koji su se složili da se pridržavaju kodeksa.

Zaista, etički kodeks može da podstakne dobro ponašanje i da se upotrebi da se prozovu akcije drugih koje odstupaju od kodeksa. On može da se dopuni zakonskom snagom, tako da se kršenja koda mogu kazniti suspenzijom ili oduzimanjem licence.

Ima više načina da se odredi i definiše etičko ponašanje; Baase (2002) o njima govori detaljno. Etički kodeks može da bude propisan ili opisan. Propisan kodeks se izražava u terminima prava i obaveza. On precizno nabraja šta može, a šta ne može da se uradi. Propisan kodeks se izražava u terminima vrlina koje kodeks namerava da podržava. On se zasniva na pretpostavci da će izražene vrednosti delovati kao vodič za odgovarajuće ponašanje.

O etičkom kodu možemo da mislimo kao o skupu osnovnih pravila integriteta i nepristrasnosti u našoj profesiji. Na primer, on može da nas usmerava da ostanemo lojalni prema našim saradnicima, poslodavcima, klijentima i potčinjenima. Kodeks je takođe skup profesionalnih pravila koja opisuju naše odgovornosti prema onima koje služimo. Na primer, on može da definiše pravila poverljivosti ili otkrivanja koja pomažu da uobličimo svoja profesionalna prosuđivanja o tome kada i kako da otkrivamo informacije. Etički kodeks takođe može da imenuje najbolje prakse, kao što je, na primer, koliko ispitivanja je dovoljno. Većina profesionalaca prihvata pojam „priznate marljivosti", gde kodeks opisuje kada za profesionalca može da se kaže da je primenio dobro prosuđivanje i opšte prihvaćenu dobru praksu u korist klijenta ili društva.

Udruženje profesionalnih inženjera Ontarija (PEO) objavljuje etički kodeks koji profesionalnim inženjerima nalaže više dužnosti:

- Dužnost prema društvu
- Dužnost prema poslodavcima
- Dužnost prema klijentima
- Dužnost prema kolegama i zaposlenima

- Dužnost prema inženjerskoj profesiji
- Dužnost prema samom sebi

U isto vreme PEO je definisalo karakteristike lošeg profesionalnog ponašanja:

- Nemarnost
- Uznemiravanje
- Neuspeh u zaštiti bezbednosti, zdravlja ili vlasništva korisnika
- Neuspeh u pridržavanju statuta, propisa, standarda i pravila
- Potpisivanje ili stavljanje pečata na dokument koji profesionalac nije pripremio ili proverio
- Neuspeh u otkrivanju sukoba interesa
- Izvršavanje zadatka izvan nečije oblasti ekspertize

Razlika između etičkog kodeksa i definicije lošeg profesionalnog ponašanja je stvar stepena; jedno nije negacija drugog. Etički kodeks je visok standard etičkog ponašanja, koji ide iznad i dalje od onoga što zahteva zakon. Nasuprot tome, loše profesionalno ponašanje je kažnjiv prekršaj, zakonski ili od strane profesionalne organizacije (na primer, suspendovanjem ili oduzimanjem prekršiočeve licence ili sertifikata). U podsetniku 14.2 naveden je Etički kodeks profesionalnog inženjerstva koji su objavile organizacije Association for Computing Machinery (ACM) i Institute for Electrical and Electronics Engineers (IEEE).

Etički izbori ponekad nisu jasni, ali oni često obuhvataju teške odluke. Etičke dužnosti mogu da se sukobe i da ne postoji pravi odgovor - može čak da ne postoji dobar izbor. Na primer, ako vaš kolega ima problem sa pićem koji može da utiče na inženjersko prosuđivanje, da li ćete to prijaviti vašem nadređenom? Ako vaš kolega ima sukob interesa, da li o tome izveštavate? A ako proizvod koji projektujete može da ima užasne posledice (na primer, da je rezultat bolje nuklearno oružje), da li ćete da ga napravite?

Profesionalni razvoj

Bez obzira na to da li smo licencirani ili sertifikovani, svako od nas ima odgovornost da shvati najnovija otkrića o proizvodima, procesima i praksama softverskog inženjerstva. Čitanje ove knjige je samo jedan delić te veće aktivnosti profesionalnog razvoja. Stalni profesionalni razvoj održava ili poboljšava naše znanje i veštine pošto smo započeli profesionalnu praksu; postajemo bolji praktičari, pravimo bolje proizvode i pružamo bolje usluge.

Postoji više organizacija koje nude pomoć u našoj potrazi da saznamo ono što je najnovije u oblasti rada i praksi. Među njima su međunarodne organizacije, kao što su Association for Computing Machinery (ACM) i Institute for Electrical and Electronics Engineers (IEEE). Regionalne ili nacionalne organizacije uključuju Canadian Society for Electrical and Computer Engineeering (CSECE), British Computer Society i Brazilian Computer Society. Te i druge organizacije pomažu u održavanju naše kompetencije pomoću:

- razvijanja standarda;
- objavljivanja istraživačkih časopisa koji proširuju naše znanje;
- objavljivanja praktičarskih časopisa koji olakšavaju prenos tehnologije između istraživača i praktičara;

- održavanja tehničkih konferencija koje olakšavaju komunikaciju sa kolegama;
- delovanja u svojstvu javnog predstavnika naših interesa;
- formiranja specijalnih interesnih grupa za istraživanje fokusiranih tema.

PODSETNIK 14.2 ACM/IEEE ETIČKI KODEKS PROFESIONALNE PRAKSE ZA SOFTVERSKO INŽENJERSTVO (KRATKA VERZIJA, © ACM/IEEE-CS 2004)

Kratka verzija kodeksa sumira ambicije na visokom nivou apstrakcije. Klauzule koje su uključene u potpunu verziju daju primere i detalje o tome kako te ambicije menjaju način na koji delujemo kao profesionalci u softverskom inženjerstvu. Bez ambicija, detalji mogu da postanu legalistički i dosadni; bez detalja, ambicije mogu da postanu zvučne, ali prazne; zajedno, ambicije i detalji daju kohezivan kodeks.

Softverski inženjeri treba samo da se angažuju da učine analizu, specifikaciju, projektovanje, razvoj, ispitivanje i održavanje softvera korisnom i cenjenom profesijom. U saglasnosti sa njihovim posvećenjem zdravlju, bezbednosti i dobrobiti javnosti, softverski inženjeri treba da se pridržavaju sledećih osam principa:

1. Softverski inženjeri treba dosledno da deluju u skladu sa javnim interesom.
2. Softverski inženjeri treba da deluju na način koji je u najboljim interesima njihovog klijenta i poslodavca, u skladu sa javnim interesom.
3. Softverski inženjeri treba da obezbede da njihovi proizvodi i izmene u vezi sa njima zadovoljavaju najviše moguće profesionalne standarde.
4. Softverski inženjeri treba da održavaju integritet i nezavisnost u svojim profesionalnim prosuđivanjima.
5. Menadžeri i vođe softverskog inženjerstva treba da prihvate i promovišu etički pristup u upravljanju razvojem i održavanjem softvera.
6. Softverski inženjeri treba da unapređuju integritet i dobar glas profesije, u skladu sa javnim interesom.
7. Softverski inženjeri treba da budu nepristrasni prema svojim kolegama i da ih podržavaju.
8. Softverski inženjeri treba da učestvuju celog života u učenju o praksi svoje profesije i treba da promovišu etički pristup profesionalnoj praksi.

Na primer, IEEE ima blizu 360000 članova. U tabeli 14.11 navedena su 42 tehnička društva koja sačinjavaju IEEE. Jedna od podružnica, IEEE Computer Society, posebno se usredsređuje na pitanja računarskih nauka i softverskog inženjerstva; ona je najveća od tehničkih društava IEEE, sa 98000 članova i najveća je organizacija računarskih profesionalaca. Društvo IEEE Computer Society je sponzor desetina tehničkih konferencija svake godine, samo i u saradnji sa drugim organizacijama, omogućavajući članovima da saznaju o najnovijem istraživanju. Njegove mnogobrojne tehničke publikacije, navedene u tabeli 14.12, informišu članstvo o svakom aspektu računarstva. Pored toga, IEEE Com-

puter Society Press objavljuje knjige i zbornike radova sa konferencija. Druge organizacije, kao što je ACM, obezbeđuju slične proizvode i usluge.

TABELA 14.11 IEEE Tehnička društva

IEEE Aerospace and Electronic Systems Society	IEEE Instrumentation and Measurement Society
IEEE Antennas and Propagation Society	IEEE Lasers & Electro-Optics Society
IEEE Broadcast Technology Society	IEEE Magnetics Society
IEEE Circuits and Systems Society	IEEE Microwave Theory and Techniques Society
IEEE Communications Society	IEEE Nanotechnology Council
IEEE Components Packaging and Manufacturing Technology Society	IEEE Nuclear and Plasma Sciences Society
IEEE Computational Intelligence Society	IEEE Oceanic Engineering Society
IEEE Computer Society	IEEE Power Electronics Society
IEEE Consumer Electronics Society	IEEE Power Engineering Society
IEEE Control Systems Society	IEEE Product Safety Engineering Society
IEEE Council on SuperConductivity	IEEE Professional Communication Society
IEEE Dielectrics and Electrical Insulation Society	IEEE Reliability Society
IEEE Education Society	IEEE Robotics & Automation Society
IEEE Electromagnetic Compatibility Society	IEEE Sensors Council
IEEE Electron Devices Society	IEEE Signal Processing Society
IEEE Engineering Management Society	IEEE Society on Social Implications of Technology
IEEE Engineering in Medicine and Biology Society	IEEE Solid-State Circuits Society
IEEE Geoscience & Remote Sensing Society	IEEE Systems, Man, and Cybernetics Society
IEEE Industrial Electronics Society	IEEE Ultrasonics, Ferroelectrics, and Frequency Control Society
IEEE Industry Applications Society	IEEE Vehicular Technology Society
IEEE Information Theory Society	
IEEE Intelligent Transportation Systems Society	

Sledeći koraci u istraživanju i praksi

Budućnost softverskog inženjerstva je u našim rukama. Moramo da proučimo načine na koje smo slični drugim inženjerima, tako da možemo da učimo iz njihovih iskustava. I moramo da proučimo načine na koje smo različiti, tako da možemo da skrojimo naše strategije, tehnike i alate za jedinstvene probleme sa kojima se susrećemo. Opštije, moramo da obezbedimo da sagledavamo softversko inženjerstvo u njegovom širem postavljanju, prepoznajući da kvalitetne softverske proizvode i procese stvaraju kreativni ljudi koji rade u timovima, a ne proizvodni proces koji „štancuje" naše igračke. Do tada, trebalo bi da prihvatimo druge discipline, uključujući i društvene nauke, tako da se naši procesi skroje da koriste ono najbolje što svaki inženjer može da ponudi, a naši proizvodi da budu što je moguće korisniji i od veće pomoći našim kupcima. Nazad, moramo da obratimo veću pažnju na posledice naših odluka u softverskom inženjerstvu. Ko je odgovoran kada naš softver otkaže? Ko snosi etičku i zakonsku odgovornost za naše promašaje u zahtevima, dizajnu, implementaciji i ispitivanju? Kao u svakoj zreloj disciplini, moramo da naučimo da preuzimamo odgovornost za svoje akcije i proizvode.

TABELA 14.12 Tehničke publikacije IEEE Computer Society

IEEE Transactions on Computers	IEEE Transactions on Visualization and Computer Graphics Computing in Science & Engineering
IEEE/ACM Transactions on Computational Biology & Bioinformatics	IEEE Annals of the History of Computing
IEEE Transactions on Dependable & Secure Computing	**IEEE Computer**
IEEE Transactions in Information Technology in Biomedicine	IEEE Computer Graphics & Applications
IEEE Transactions on Knowledge and Data Engineering	IEEE Design & Test of Computers
IEEE Transactions on Mobile Computing	IEEE Intelligent Systems
IEEE Transactions on Multimedia	IEEE Internet Computing
IEEE Transactions on Nanobioscience	IEEE Micro
IEEE Transactions on Networking	IEEE MultiMedia
IEEE Transactions on Parallel and Distributed Systems	IEEE Pervasive Computing
IEEE Transactions on Patterns Analysis and Machine Intelligence	IEEE Security & Privacy
IEEE Transactions od Software Engineering	**IEEE Software**
IEEE Transactions on Very Large Scale Integration (VLSI) Systems	IT Professional

14.5 SEMINARSKI RAD

Vaš seminarski rad je završen. Razmotrite završen proizvod sa tačke gledišta prenosa tehnologije. Kako biste predstavili Organizator pozajmica analitičaru kredita u FCO? Gde se analitičari kredita i njihovi menadžeri uklapaju u kontinuum usvajanja koji je opisao Rogers? Kako njihova klasifikacija nagoveštava strategije za poboljšanje korisničkog interfejsa? Obuke? Podrške? Evolucije proizvoda?

14.6 KLJUČNE REFERENCE

Opštim pitanjima u vezi sa prenosom tehnologije obraćaju se mnoge konferencije, uključujući International Conference on Software Engineering. Članak The Redwine and Riddle (1985) ostaje do sada najiscrpniji pregled o softverskoj tehnologiji. Institut SAD za softversko inženjerstvo (SEI) objavio je više tehničkih izveštaja o prenosu tehnologije. U nedavnim člancima se govori o prenosu softverske tehnologije uključujući one čiji su autori Pfleeger (1999a), Pfleeger (1999b) i Pfleeger i Menezes (2000).

Literatura iz društvenih nauka je bogata informacijama o grupnom donošenju odluka. Klein (1998) je napisao vrlo čitljiv uvod u model donošenja odluke pripremljen prepoznavanjem. Izdanje časopisa *IEEE Softver* za januar/februar 2000. godine bavi se opštijim pitanjem šta možemo da naučimo iz drugih disciplina.

Za informacije o Delfi tehnici, pročitajte reference Linstone i Turoff (1975) i Turoff i Hiltz (1995).

Diskusijom o licenciranju softverskog inženjerstva bavi se izdanje *IEEE Softver* za novembar/decembar 1999. godine i izdanje *IEEE Computer* za maj 2000. godine. Kaner i Pels (1998) razmatraju ko je odgovoran kad softver otkaže i šta možete da uradite kao potrošač kada vaš softverski proizvod ne radi ispravno.

Druge odlične reference u vezi sa softverskom sertifikacijom i licenciranjem uključuju radove čiji su autori Allen i Hawthorn (1999), Canadian Engineering Quality Board (2001), Knight i Leveson (2000), Notkin (2000), Parnas (2000) i Shaw (1990). Kao što ćete videti, mnoge od smernica su u obliku nacrta, zato što su pitanja licenciranja i akreditacije još uvek predmet diskusija u zajednici softverskog inženjerstva. Professional Engineers Act, jedan od statuta provincije Ontario, može da se pronađe na stranici 28 materijala na veb lokaciji http://192.75.156.68/DBLaws/Statues/English/90p28e.htm. Opis programa iz softverskog inženjerstva na Univerzitetu Waterloo je na lokaciji http://www.softeng.uwaterloo.ca.

Odlična rasprava o etici i ulozi lične odgovornosti može da se pronađe u Baase (2002).

14.7 VEŽBE

1. Razmotrite aktivnosti menadžera softverskog projekta. Gde se donose odluke? Kada su one grupne odluke? Kada su one odluke pojedinaca?
2. Koje softverske tehnologije su promovisane u poslednjih deset godina? Koje od njih su široko usvojene, a koje nisu? Možete li da upotrebite Rogersov i Mooreov okvir da biste objasnili zašto?
3. Izdanje časopisa *IEEE Softver* za januar/februar 2000. godine sadrži uvodnik od McConnella o najboljim uticajima softverskog inženjerstva:

   - revije i inspekcije;
   - skrivanje informacija;
   - postepen razvoj;
   - angažovanje korisnika;
   - automatizovana kontrola revizije;
   - razvoj korišćenjem Interneta;
   - programski jezici: Fortran, COBOL, Turbo Pascal, Visual Basic;
   - model zrelosti sposobnosti za softver;
   - razvoj zasnovan na komponentama;
   - metrike i merenja.

   Za svaku od navedenih praksi, analizirajte njeno verovatno tehnološko usvajanje. Da li je to najbolja praksa? Koji dokaz podržava njeno usvajanje? Kom delu publike se obraća tim dokazom?

Bibliografija sa napomenama

Abdel-Ghaly, A. A., P. Y. Chan i B. Littlewood (1986), „Evaluation of competing software reliability predictions", *IEEE Transactions on Software Engineering*, SE 12(9): 950-967.

Analizira nekoliko modela pouzdanosti. Uvodi pojmove u-dijagrama, verovatnoće na osnovu prethodnih dogadaja, šuma.

Abdel-Hamid, Tarek (1989), „The dynamics of software project staffing: A system dynamics based simulated approach", *IEEE Transactions on Software Engineering*, 15(2) (februar): 109-119.

Koristi modele dinamike sistema za simuliranje efekata različitih nivoa osoblja u projektu razvoja softvera.

--- (1990), „Investigating the cost/schedule trade-off in software development", *IEEE Software*, 7(1): 97-105, januar.

--- (1996), „The slippery path to productivity improvement", *IEEE Software*, 13(4) (jul): 43-52.

Predlaže korišćenje dinamičkih sistema radi optimizalnog iskorišćavanja resursa u softverskim razvojnim projektima.

Abdel-Hamid, Tarek i Stuart Madnick (1991), *Software Project Dynamics: An Integrated Approach*. Englewood Cliffs, NJ: Prentice Hall.

Govori o modelima dinamičkih sistema u procesu razvoja softvera.

Ackerman, F., L. S. Buchwald i F. H. Lewski (1986), „Software inspections: An effective verification process", *IEEE Software*, 6(3) (maj): 31-36.

Adams, E. (1984), „Optimizing preventive service of software products", *IBM Journal of Research and Development*, 28(1): 2-14.

Analizira više velikih softverskih sistema i dokazuje da većina softverskih grešaka retko dovodi do otkaza, dok samo mali broj izaziva najčešće otkaze.

Agile Alliance (2001), Principles of the Agile Alliance, http://www.agilealliance.org/.

Akao, Yoji (1990), *Quality Function Deployment: Integrating Customer Requirements Into Product Design*, Productivity Press, 1990.

Alexander, Christopher (1979a), *The Timeless Way of Building*, New York: Oxford University Press.

Prva knjiga u kojoj je uveden pojam šablonski dizajn u svetlu arhitekture.

--- (1979b), *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press.

Alford, M. (1977), „A requirements engineering methodology for real-time processing requirements", *IEEE Transactions on Software Engineering*, SE 3(1) (januar): 60-69.

Članak u kojem se prvi put uvodi SREM.

--- (1985), „SREM at the age of eight: The distributed computing design system", *IEEE Computer*, 18(4) (april): 36-46.

Allen, Frances i Paula Hawthorn (1999), „ACM Task Force on Professional Licensing in Software Engineering", (maj). http://www.acm.org/serving/se_policy/report.html.

Ambler, Scott W. (2004a), „Agile Requirements Modeling", Official Agile Modeling Site, http://www.agilemodeling.com/essays/agileRequirements.htm.

--- (2004b), „Agile Software Development", Official Agile Modeling Site, http://www.agilemodeling.com/essays/agileSoftwareDevelopment.htm.

Andriole, Stephen J. (1993), *Rapid Application Prototyping: The Storyboard Approach to User Requirements Analysis*. New York: John Wiley.

Anonymous (1996), „In a hurry are we, sir?" *Pilot*.

Opisuje neočekivani problem sa radarskim softverom u mlaznom avionu Harrier.

Anthes, Gary H. (1997), „How to avoid killer apps", *Computerworld*, jul 7.

Dobar i pristupačan opšti pregled nekih posledica nebezbednog dizajna sistema. Takođe dostupan na Webu, na adresi http://www.computerworld.com/features/970707killer.html.

Arango, Guillermo, Eric Schoen i Robert Pettengill (1993), „Design as evolution and reuse", u *Proceedings of the Second International Workshop on Software Reusability* (Luka, Italija, mart 24-26), Los Alamitos, CA: IEEE Computer Society Press.

Opisuje korišćenje tehnoloških i projektnih evidencija da bi se sačuvane lekcije naučene na jednom projektu koristile u sledećem.

Ardis, Mark A., John A. Chaves, Lalita Jategaonkar Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas i James Von Olnhausen (1996), „A framework for evaluating specification methods for reactive systems", *IEEE Transactions on Software Engineering*, 22(6) (jun): 378-389.

Izveštaj o iskustvu, prvobitno izložen na 17. međunarodnoj konferenciji o softverskom inženjerstvu, koji daje kriterijume za izbor jedne od više tehnika za specifikaciju zahteva. Sadrži osnovne i važne kriterijume, a pokazuje i njihovu primenu tokom razvojnog ciklusa.

Arnold, Robert S. (1993), *Software Reengineering*. Los Alamitos, CA: IEEE Computer Society Press.

Arnold, Robert S. i Shawn Bohner (1996), *Software Change Impact Analysis*. Los Alamitos, CA: IEEE Computer Society Press.

Arthur, Lowell Jay (1997), „Quantum improvements in software system quality", *Communications of the ACM*, 40(6) (jun): 47-52.

Opisuje iskustva na poboljšavanju softvera u U. S. West Technologies. Ističe neke od učinjenih grešaka, uključujući i fokusiranje na pogrešne ciljeve.

Asch, Solomon (1956), „Studies of independence and submission to group pressure", *Psychological Monographs*, 70.

Associated Press (1996), „Pilot's computer error cited in plane crash", *Washington Post*, avgust 24, str. A4.

Novinski izveštaj o ulozi računarske greške u vazduhoplovnoj katastrofi u Kolumbiji.

Avizienis, A. i J. P. J. Kelly (1984), „Fault tolerance through design diversity: Concepts and experiments", *IEEE Computer*, 17(8): 67-80.

Opisuje pojam korišćenja nekoliko nezavisno dizajniranih softverskih sistema na bazi istih zahteva. Cilj je da se svi sistemi u isto vreme izvrše da bi se glasalo i postupilo po odluci većine. Ovo je filozofija na kojoj počivaju redundantni sistemi za svemirske šatle u SAD. Pročitajte izveštaj Knight i Levesona (1986) gde se iznosi suprotno mišljenje.

Baase, Sara (2002), *A Gift of Fire: The Social, Legal and Ethical Issues for Computers and the Internet*, 2. izdanje, Upper Saddle River, NJ: Prentice Hall.

Babich, Wayne (1986), *Software Configuration Management*. Reading, MA: Addison-Wesley.

Dobar opšti pregled ključnih pitanja vezanih za upravljanje konfiguracijom sa nekoliko studijskih primera na kraju knjige.

Bach, James (1997), „Test automation snake oil", u *Proceedings of the Fourteenth International Conference and Exposition on Testing Computer Software*, str. 19-24 (Washington, DC, jun 16-19).

Predlaže smernice za donošenje razumnih odluka o mogućnostima i nemogućnostima automatizacije prilikom testiranja. Može se nabaviti preko Frontier Technologies, Annapolis, MD.

Bailey, John W. i Victor R. Basili (1981), „A meta-model for software development resource expenditures", u Proceedings of the Fifth International Conference on Software Engineering, str. 107-116. Los Alamitos, CA: IEEE Computer Society.

Baker, F.T. (1972), „Chief programmer team management of production programming", *IBM Systems Journal*, 11(1).

Prvi rad u kojem se predlaže organizacija tima sa glavnim programerom za softverske razvojne projekte.

Balzer, Robert (1981a), „Transformational implementation: An example", *IEEE Transactions on Software Engineering*, SE 7(1) (januar): 3-14.

Opisuje transformacioni model softverskog razvojnog procesa.

--- (1981b), Gist Final Report (februar), Los Angeles: University of Southern California, Information Sciences Institute.

Banker R., R. Kauffman i R. Kumar (1992), „An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment", *Journal of Management Information Systems*, 8(3) (Zima 1992): str. 127-150.

Uvodi notaciju objektnih tačaka za merenje veličine sistema.

Barghouti, Naser S. i Gail E. Kaiser (1991), „Scaling up rule-based development environments", u *Proceedings of the Third European Software Engineering Conference* (Milano, Italija), *Lecture Notes in Computer Science*, no. 55, str. 380-395. Amsterdam: Springer-Verlag.

Barghouti, Naser S., David S. Rosenblum, David G. Berlanger i Christopher Alliegro (1995), „Two case studies in modeling real, corporate processes", *Software Process: Improvement and Practice*, 1(1): 17-32.

Barnard, J. i A. Price (1994), „Managing code inspection information", *IEEE Software*, 11(2) (mart): 59-69.

Opširna kvantitativna studija o uvođenju Fagan inspekcija u firmi AT&T. Koristi paradigmu cilj-pitanje-metrika u odlučivanju šta da se meri.

Barnes, Bruce i Terry A. Bollinger (1991), „Making reuse cost-effective", *IEEE Software*, 8(1) (januar): 13-24.

Divan izveštaj o anketi koja je pokazala neke od ključnih pitanja za uspešno višekratno korišćenje softvera.

Barron, D. W. i J. M. Bishop (1984), *Advanced Programming*. New York: John Wiley.

Basili, Victor R. (1990), „Viewing maintenance as reuse-oriented software development", *IEEE Software*, 7(1) (januar): 19-25.

Basili, Victor R., Lionel Briand i Walcelio Melo (1995), „A validation of object-oriented design metrics", tehnički izveštaj sa Odeljenja kompljuterskih nauka na Univerzitetu u Merilendu CS-TR-3443 (maj).

Basili, Victor R. i Scott Green (1994), „Software process evolution at the SEL", *IEEE Software*, 11(4) (jul): 58-66.

Opisuje upotrebu paradigme unapređenja kvaliteta, studijskih primera i eksperimenata na istraživanju efekata tehnika čitanja i čiste sobe. Posebno korisna za opisivanje načina upotrebe različitih vrsta studija za procenjivanje tehnologije.

Basili, Victor R. i Barry T. Perricone (1984), „Software errors and complexity: An empirical investigation", *Communications of the ACM*, 27(1): 42-52.

Analizira distribucije i odnose izvedene iz podataka o softverskim promenama.

Bates, Clive (1997), „Test it again-how long?", u *Proceedings of the Fourteenth International Conference and Exposition on Testing Computer Software* (Washington, DC, jun 16-19).

Može se nabaviti preko Frontier Technologies, Annapolis, MD.

Beck, Kent (1999), *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley.

Beck, Kent et al. (2004), „Manifesto for Agile Software Development", oktobar.

http://www.agilemanifesto.org.

Beizer, Boris (1990), *Software Testing Techniques*, 2. izdanje, New York: Van Nostrand.

Drugo izdanje jednog od nekoliko sveobuhvatnih tekstova o testiranju softvera. Sadrži detaljne opise mnogih specifičnih strategija i ozbiljno razmatra problem merenja.

--- (1995). *Black-Box Testing*. New York: John Wiley.

--- (1997), „Cleanroom process model: A critical examination", *IEEE Software*, 14(2) (mart/april): 14-16.

Sugeriše da se pristupom čiste sobe zanemaruju osnovi teorije testiranja i da čista soba predstavlja neodgovoran postupak.

Belady, L. i M. M. Lehman (1972), „An introduction to growth dynamics", u W. Freiberger (urednik), *Statistical Computer Performance Evaluation*. New York: Academic Press.

Uvodi jedan od prvih matematičkih modela za procenu troškova održavanja softvera.

Bentley, Jon (1986), *Programming Pearls*. Reading, MA: Addison-Wesley.

--- (1989), *More Programming Pearls*. Reading, MA: Addison-Wesley.

Berard, Edward V. (2000), „Abstraction, encapsulation and information hiding,"

http://www.itmweb. com/essay550.htm.

Dobar rad o razlikama između apstrahovanja, enkapsuliranja i sakrivanja informacija.

Berry, Daniel M. (2002a), „The inevitable pain of software development, including of extreme programming, caused by requirements volatility", *Proceedings of the Workshop on Time-Constrained Requirements Engineering (T-CRE)*. Esen, Nemačka (septembar): 9-19. Los Alamitos, CA: IEEE Computer Society Press.

Berry, Daniel M. (2002b), „The inevitable pain of software development: Why there is no silver bullet", u *Proceedings of Monterey Workshop 2002: Radical Innovations of Software and Systems Engineering in the Future*. Venecija, Italija (oktobar): 28-47. Los Alamitos, CA: IEEE Computer Society Press.

Bertolino, A. i L. Strigini (1996), „On the use of testability measures for dependability assessment", *IEEE Transactions on Software Engineering*, 22(2) (februar): 97-108.

Autori ispituju pojam mogućnosti testiranja koji predlaže Voas i ističu potencijalne opasnosti. Ako povećavate mogućnost testiranja programa možda zaista povećavate verovatnoću da u programu ne bude grešaka, ali istovremeno povećavate verovatnoću da će preostale greške dovesti do otkaza. Oni predlažu unapređeni model mogućnosti testiranja.

Beyer, Hugh i Karen Holtzblatt (1995), „Apprenticing with the customer: A collaborative approach to requirements definition", *Communications of the ACM*, 35(8) (maj 1995): 45–52.

Bieman, James M. i Linda M. Ott (1993), Measuring Functional Cohesion, tehnički izveštaj TR CS-93-109. Fort Collins: Colorado State University, Computer Science Department.

Binder, Robert V. (1997), „Can a manufacturing quality model work for software?" *IEEE Software*, 14(5) (septembar/oktobar): 101–105.

Kolumna Quality Time u kojoj se opisuje zašto se pojam prvoklasnog kvaliteta ne može primeniti na softver.

Binder Robert. (2000), *Testing Object-Oriented Systems*. Reading, MA: Addison-Wesley.

Bodker, K. i J. Pedersen (1991), „Workplace cultures: Looking at artifacts, symbols and practices", u J. Greenbaum i M. Kyng (urednici), *Design at Work: Cooperative Design of Computer Systems*, str. 121–136. Hillsdale, NJ: Lawrence Erlbaum.

Boehm, Barry (2000), „Requirements that handle IKIWISI, COTS i rapid change", *IEEE Computer*, 33(7) (jul): 99–102.

Boehm, B. W. (1981), *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.

Ovo je jedna od prvih knjiga u kojoj se softversko inženjerstvo posmatra iz „inženjerskog" ugla. Boehm opisuje izvođenje modela COCOMO i njegovu primenu na softver i na ocenu rokova. Knjiga je posebno značajna zato što Boehm zasniva COCOMO na velkom uzorku dobijenom od organizacije TRW koja radi za vojsku.

--- (1988), „A spiral model for software development and enhancement", *IEEE Computer*, 21(5) (maj): 61–72.

Opisuje model za uklapanje procedura upravljanja rizikom u model životnog ciklusa softvera.

--- (1989), *Software Risk Management*. Los Alamitos, CA: IEEE Computer Society Press.

Izvrstan udžbenik za rešavanje pitanja rizika u projektima razvoja softvera.

--- (1990), „Verifying and validating software requirements and design specifications", u *System and Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press.

--- (1991), „Software risk management: Principles and practices", *IEEE Software*, 8(1) (januar): 32–41.

Dobar kratak pregled terminologije, modela i primera upravljanja rizikom.

--- (1996), „Anchoring the software process", *IEEE Software*, 13(4) (jul): 73–82.

--- (2000), „Software cost management with COCOMO II", u *Proceedings of the International Conference on Applications of Software Measurement*. Orange Park, FL: Software Quality Engineering.

Boehm, B. W., J. R. Brown, J. R. Kaspar, M. Lipow i G. MacCleod (1978), *Characteristics of Software Quality*. Amsterdam: North Holland.

Predlaže definicije i mere za niz atributa kvaliteta. U ovoj knjizi se opisuje „model" kvaliteta softvera koji se od tada naziva Boehmov model kvaliteta.

Boehm, B. W., C. Clark, E. Horowitz, C. Westland, R. Madachy i R. Selby (1995), „Cost models for future life cycle processes: COCOMO 2.0", *Annals of Software Engineering*, 1(1) (novembar): 57–94.

U ovom referatu opisuju se problemi uočeni pri korišćenju originalnog modela COCOMO i tehnika primenjenih za njihovo rešavanje u revidiranoj verziji modela COCOMO.

Boehm, B. W., T. E. Gray i T. Seewaldt (1984), „Prototyping versus specifying: A multi-project experiment", *IEEE Transactions on Software Engineering*, SE 10(3) (mart).

Boehm, B. W. i C. Papaccio (1988), „Understanding and controlling software costs", *IEEE Transactions on Software Engineering*, 14(10) (oktobar): 14–66.

Böhm, C. i G. Jacopini (1966), „Flow diagrams, Turing machines and languages with only two formation rules", *Communications of the ACM*, 9(5) (maj).

Klasičan tekst u kome se dokazuje da se dizajn može pisati isključivo pomoću sekvence, odlučivanja i iteracije – to jest, bez iskaza goto.

Bohner, Shawn A. (1990), Technology Assessment on Software Reengineering, tehnički izveštaj, CTC-TR-90-001P. Chantilly, VA: Contel Technology Center.

Divan, jasan opis međusobnih veza reinženjeringa, reverznog inženjeringa, restrukturisanja. Na žalost, može se dobiti isključivo od autora.

Bollinger, Terry B. i Clement L. McGowan (1991), „A critical look at software capability evaluations", *IEEE Software*, 8(4) (jul): 25–41.

Jasan, pronicljiv referat koji dovodi u pitanje podloge zrelosti procesa. Obavezna literatura za svakog ko razmišlja o izboru strategije za unapređenje procesa.

Bollinger, Terry B. i Shari Lawrence Pfleeger (1990), „The economics of software reuse: issues and alternatives", *Information and Software Technology*, 32(10) (decembar): 643–652.

Booch, Grady (1994), *Object-Oriented Analysis and Design with Applications*, 2. izdanje, San Francisko, CA: Benjamin-Cummings.

Braun, Christine i Rubén Prieto-Díaz (1990), Technology Assessment of Software Reuse, tehnički izveštaj CTC-TR-90-004P. Chantilly, VA: Contel Technology Center.

Brealey R. i S. Myers (1991), *Principles of Corporate Finance*. New York: McGraw-Hill.

Brettschneider, Ralph (1989), „Is your software ready for release?" *IEEE Software*, 6(4) (jul): 100–108.

Opisuje jednostavan model koji se koristi u Motoroli, a naziva se testiranje na nula grešaka.

Briand, Lionel C., Victor R. Basili i William M. Thomas (1992), „A pattern recognition approach for software engineering data analysis", *IEEE Transactions on Software Engineering*, 18(11): 931–942.

Koristi optimalno smanjivanje skupa za procenu troškova.

Briand, Lionel C., Prem Devanbu i Walcelio Melo (1997), „An investigation into coupling measures for C++", u *Proceedings of the Nineteenth International Conference on Software Engineering* str. 412–421 (Boston, MA), maj, 1997. Los Alamitos, CA: IEEE Computer Society Press.

Predlaže se niz mera za objektno orijentisane dizajne. Prikazuje kako neke od tih mera mogu biti korisne za otkrivanje grešaka.

Briand, Lionel C., Sandro Morasca i Victor R. Basili (1994), Defining and Validating High-Level Design Metrics, technical Report CS-TR-3301. College Park, MD: University of Maryland, Department of Computer Science.

Brodman, Judith G. i Donna L. Johnson (1995), „Return on investment (ROI) from software process improvement as measured by U.S. industry", *Software Process-Improvement and Practice*, 1(1): 35-47.

U ovom referatu ispituju se načini na koji 33 organizacije izračunavaju povraćaj od investicija u softver. Prikazuju se velike nekonsistentnosti zbog kojih je nemoguće objediniti njihove rezultate.

Brooks, Frederick P., Jr. (1975), *The Mythical Man-Month*. Reading, MA: Addison-Wesley.

Klasična knjiga eseja koja se zasniva na autorovim iskustvima pri izgradnji operativnog sistema OS-360. Mnoga njegova zapažanja odnose se na organizacionu dinamiku, kao što je ubacivanje dodatnog osoblja na projekat koji kasni. Postoji novo izdanje iz 1995.

Brooks, Frederick P., Jr. (1987), „No silver bullet: Essence and accidents of software engineering", *IEEE Computer*, 20(4) (april): 10-19.

Busby, J. S. i S. C. Barton (1996) „Predicting the cost of engineering: Does intuition help or hinder?" *Engineering Management Journal*. 6(4):177-182.

Canadian Engineering Qualifications Board (CEQB) (2004), „Software Engineering Syllabus-1998", http://www.ccpe.ca/e/files/syllabus_4_17.pdf.

Canadian Engineering Quality Board (2001), „Core and Supplementary Bodies of Knowledge for Software Engineering: A report prepared for the CCPE as part of the CEQB Body of Knowledge Development Pilot Project", nacrt verzije 0.4 (septembar 5).

Card, David N. (1992), „Capability evaluations rated highly variable", *IEEE Software*, 9(5) (septembar): 105-107.

Card, David N., V. E. Church i William W. Agresti (1986), „An empirical study of software design practices", *IEEE Transactions on Software Engineering*, 12(2) (februar): 264-271.

Card, David N. i Robert L. Glass (1990), *Measuring Software Design Quality*. Englewood Cliffs, NJ: Prentice Hall.

Zanimljiv opis razvojnog procesa jednog sistema merenja na osnovu ciljeva koji se žele postići tim merenjem i ponašanja.

Cashman, P. M. i A. W. Holt (1980), „A communication-oriented approach to structuring the software maintenance environment", *ACM SIGSOFT Software Engineering Notes*, 5(1) (januar).

Cavano, Joseph P. i Frank S. LaMonica (1987), „Quality assurance in future development environments", *IEEE Software*, 7(5) (septembar): 26-34.

Chen, Peter (1976), „The entity-relationship model: Toward a unified view of data", *ACM Transactions on Database Systems*, 1(1): 9-36.

Chidamber, S. R. i C. F. Kemerer (1994), „A metrics suite for object-oriented design", *IEEE Transactions on Software Engineering*, 20(6): 476-493.

Chillarege, Ram, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray i Man-Yuen Wong (1992), „Orthogonal defect classification: A concept for in-process measurements", *IEEE Transactions on Software Engineering*, 18(11) (novembar): 943-956.

Coad, Peter i Edward Yourdon (1991), *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Cobb, R. H. i H. D. Mills (1990), „Engineering software under statistical quality control", *IEEE Software*, 7(6) (novembar): 44-54.

Cockburn, Alistair (2002), *Agile Software Development*. Reading, MA: Addison-Wesley.

Cockburn, Alistair i Laurie Williams (2000), „The costs and benefits of pair programming," *Proceedings of Extreme Programming and Flexible Processes in Software Engineering* (XP2000), Kaljari, Italija (jun).

Coffee, Peter (1997), „Pathfinder made not so soft a landing", *PC Week*, jul 19.

Cohen, David, Siddhartha Dalal, Jesse Parelius i Gardner Patton (1996), „The combinatorial approach to automatic test generation", *IEEE Software*, 13(5) (septembar): 83–88.

Opisuje kako *se* kombinatornim pristupom skraćuje vreme potrebno za izradu plana testiranja sa mesec dana na manje od nedelju dana.

Cole, M. i P. Griffin (1990), „Cultural amplifiers reconsidered", u D. R. Olson (urednik), *The Social Foundations of Language and Thought*, str. 343–364. New York: W. W. Norton.

Coleman, Derek, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes i Paul Jeremaes (1994), *Object-Oriented Development: The Fusion Method*. Englewood Cliffs, NJ: Prentice Hall.

Opisuje metod za integrisanje i proširivanje najboljih objektnih metodologija OMT, CRC i Objectory. Sadrži mnogo dobro objašnjenih primera.

Coleman, Don, Dan Ash, Bruce Lowther i Paul Oman (1994), „Using metrics to evaluate software system maintainability", *IEEE Computer*, 27(8) (avgust): 44–49.

Opisuje kako se u Hewlett-Packardu pri odlučivanju o održavanju koriste merenja.

Collier, Bonnie, Tom DeMarco i Peter Fearey (1996), „A defined process for project postmortem reviews", *IEEE Software*, 13(4) (jul): 65–72.

Compton, B. T. i C. Withrow (1990), „Prediction and control of Ada software defects", *Journal of Systems and Software*, 12: 199–207.

Izveštaj o ponašanju softverskih grešaka u Unisysu.

Computer Weekly Report (1994), „Sources of errors", avgust 12.

Conklin, Peter F. (1996), „Enrollment management: Managing the Alpha AXP program", *IEEE Software* 13(4) (jul): 53–64.

Opisuje pristup upravljanju projektima koji je uspešno primenjen prilikom razvijanja čipa Alpha u kompaniji Digital.

Conte, S., H. Dunsmore i V. Shen (1986), *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin-Cummings.

Lep opšti pregled i istorija softverskih merenja. Posebno je dobar odeljak sa poređenjem modela za procenu troškova.

Courtney, R. E. i D. A. Gustafson (1993), „Shotgun correlations in software measures", *Software Engineering Journal*, 8(1): 5–13.

Curtis, Bill, W. E. Hefley i S. Miller (1995). People Capability Maturity Model, tehnički izveštaji SEI-CMU-TR-95-MM-001 and -002. Pittsburgh, PA: Software Engineering Institute.

Curtis, Bill, Marc I. Kellner i Jim Over (1992), „Process modeling", *Communications of the ACM*, 35(9) (septembar): 75–90.

Dobar referat sa opštim pregledom različitih načina za modelovanje procesa.

Curtis, Bill, Herb Krasner, Vincent Shen i Neil Iscoe (1987), „On building software process models under the lamppost", u *Proceedings of the 9th International Conference on Software Engineering*, str. 96–103. Monterey, CA: IEEE Computer Society Press.

Značajan rad koji sugeriše da često modeliramo ono što se lako modelira, a ne ono što nam treba.

Curtis, Bill, Herb Krasner i Neil Iscoe (1988), „A field study of the software design process for large systems", *Communications of the ACM*, 31(11) (novembar): 1268–1287.

Procena značajnih aktivnosti u 17 softverskih razvojnih projekata.

Cusumano, Michael i Richard W. Selby (1995), *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People*. New York: Free Press/Simon and Schuster.

--- (1997), „How Microsoft builds software", *Communications of the ACM*, 40(6) (jun): 53–61.

Zanimljiv opis toga kako se u Microsoftu primenjuju dobri postupci softverskog inženjeringa, a pri tom se održava izvestan hakerski mentalitet.

Davis, Alan M. (1993), *Software Requirements: Objects, Functions and States*, rev. izdanje. Englewood Cliffs, NJ: Prentice Hall.

Dawid, A. P. (1984), „Statistical theory: The prequential approach", *Journal of the Royal Statistical Society*, A147: 278–292.

De Almeida, Mauricio, Hakim Lounis i Walcelio Melo (1997), „An investigation on the use of machine learning models for estimating software correctability", tehnički izveštaj CRIM-97/08–81. Montreal: Centre de Recherche Informatique de Montréal.

DeMarco, T. (1978), *Structured Analysis and System Specification*. New York: Yourdon Press.

Jasna i ubedljiva argumentacija u korist strukturisane analize zahteva.

--- (1982), *Controlling Software Projects*. New York: Yourdon Press.

Zabavna i jasna argumentacija upotrebe merenja za razmatranje i usmeravanje softverskih projekata. Sadrži i definiciju „sistemskog praska" i dobar opis značaja procenjivanja.

--- (1997), *The Deadline: A Novel about Project Management*. New York: Dorset House.

DeMarco, T. i T. Lister (1985), „Programmer performance and the effects of the workplace", u *Proceedings of the Eighth International Conference on Software Engineering* (London), Los Alamitos, CA: IEEE Computer Society Press.

--- (1987), *Peopleware: Productive Projects and Teams*. New York: Dorset House.

Deming, W. Edwards (1989), *Out of Crisis*. Cambridge, MA: MIT Center for Advanced Engineering Study.

Denton, Lynn (1993), *Designing, Writing and Producing Computer Documentation*. New York: McGraw-Hill.

Department of Defense (1977), *Automated Data Systems Documentation Standards*. Washington, DC: DOD.

Department of Trade and Industry (1992), „TickIT guide to software quality management, system construction and certification using ISO 9001/EN 29001/BS 5750 issue 2.0", Available from TickIT project office, 68 Newman Street, London W1A 4SE, UK.

DeYoung, G. E. i G. R. Kampen (1979), „Program factors as predictors of program readability", u *Proceedings of the Computer Software and Applications Conference*, str. 668–673. Los Alamitos, CA: IEEE Computer Society Press.

Dion, Raymond (1993), „Process improvement and the corporate balance sheet", *IEEE Software*, 10(4) (jul): 28–35.

Opisuje uticaj poboljšanja procesa razvoja softvera i primene SEI CMM modela na poslovni dobitak ili gubitak korporacija.

Dixon, Rand (1996), *Client/Server and Open Systems*. New York: John Wiley.

Opšti pregled prednosti i mana klijent/server modela sa dodatnim informacijama o dobavljačima koji podržavaju klijent/server model i proizvode sa otvorenom arhitekturom.

Dressler, Catherine (1995), „We've got to stop meeting like this", *Washington Post*, decembar 31, str. H2.

Zanimljiv članak sa predlozima za unapređenje radnih sastanaka u sklopu projekata.

Drobka, Jerry, David Noftz i Rekha Raghu (2004), „Piloting XP on four mission-critical projects", *IEEE Software* 21(6) (novembar/decembar): 70–75.

Dromey, R. Geoff (1996), „Cornering the chimera", *IEEE Software*, 13(1) (januar): 33–43.

Model kvaliteta proizvoda gde se sve karakteristike nižeg nivoa definišu tako da mogu da se izmere, a zatim spoje u karakteristike višeg nivoa.

Dutertre, Bruno i Victoria Stavridou (1997), „Formal requirements analysis of an avionics control system", *IEEE Transactions on Software Engineering*, 23(5) (maj): 267–277.

Opisuje metod za specifikaciju i proveravanje sistema u realnom vremenu pomoću PVS-a. Sadrži formalnu specifikaciju funkcionalnih i bezbednosnih zahteva. Prikazuje konsistentnost i neka svojstva bezbednosti.

Easterbrook, Steve i Bashar Nuseibeh (1996), „Using viewpoints for inconsistency management", IEEE Software Engineering Journal, 11(1) (januar), BCS/IEE Press: 31–43.

Eckhardt, D. E. i L. D. Lee (1985), „A theoretical basis for the analysis of multiversion software subject to coincident errors", *IEEE Transactions on Software Engineering*, SE-11(12): 1511–17.

Ehn, P. (1988), *Work-Oriented Design of Computer Artifacts. Stockholm*: Almquist & Wiksell International.

El Emam, Khaled i N. H. Madhavji (1995), „The reliability of measuring organizational maturity", *Software Process Improvement and Practice*, 1(1):3–25.

Elmendorf, W. R. (1973), Cause-Effect Graphs in Functional Testing, tehnički izveštaj TR-00.2487. Pokipsi, NY: IBM Systems Development Division.

--- (1974), „Functional analysis using cause-effect graphs", u *Proceedings of SHARE XLIII*. New York: IBM.

Elssamadissy, A. i G. Schalliol (2002), „Recognizing and responding to 'bad smells' in extreme programming", u *Proceedings of the 24th International Conference on Software Engineering*, (maj), Los Alamitos, CA: IEEE Computer Society Press.

Engle, Charles, Jr. i Ara Kouchakdjian (1995), „Engineering software solutions using cleanroom", u *Proceedings of the Pacific Northwest Quality Conference*. Portland, OR.

Jedan primer koji Beizer citira kao ortodoksnost pristupa čiste sobe u kojoj se izbegava jedinično testiranje.

ESPI Exchange (1996), „Productivity claims for ISO 9000 ruled untrue", London: European Software Process Improvement Foundation. oktobar, str. 1.

Rasprava u britanskoj Organizaciji za poštovanje standarda u oglašavanju, na kojoj je zaključeno da se sertifikatom ISO 9000 ne osigurava kvalitet proizvoda.

Evans, M. i J. Marciniak (1987), *Software Quality Assurance and Management*. New York: John Wiley.

Fagan, M. E. (1976), „Design and code inspections to reduce errors in program development", *IBM Systems Journal*, 15(3):182-210.

Originalni izveštaj Faganovih inspekcija.

--- (1986), „Advances in software inspections", *IEEE Transactions on Software Engineering*, SE 12(7): 744-751.

Dorađen opis klasičnog Faganovog pristupa inspekcijama.

Favaro, John (1996), „Value based principles for management of reuse in the enterprise", u Proceedings of the Fourth International Conference on Software Reuse (Orlando, Florida), Los Alamitos, CA: IEEE Computer Society Press.

Favaro, John i Shari Lawrence Pfleeger (1998), „Making Software Development Investment Decisions", *ACM Software Engineering Notes*, 23(5)(septembar): 69-74.

Fenelon, P., J. A. McDermid, M. Nicholson i D. J. Pumfrey (1994), „Towards integrated safety analysis and design", *ACM Applied Computing Reviews* (jul).

Dobar opšti pregled tehnika za procenjivanje bezbednosti softvera.

Fenton, Norman E. i Shari Lawrence Pfleeger (1997), *Software Metrics: A Rigorous and Practical Approach*, 2. izdanje, London: PWS Publishing, 1997.

Fernandes, T. (1995), *Global Interface Design*. London: Academic Press.

Fewster, Mark i Dorothy Graham (1999), *Automated Software Testing*. Reding MA: Addison-Wesley.

Field, Tom (1997a), „A good connection", *CIO Magazine*, 1. februar.

Opis načina na koji su u kompaniji Bell Atlantic postupili sa zastarelim sistemima i unapredili usluge i proizvode.

--- (1997b), „Banking on the relationship", *CIO Magazine*, 1. februar.

Opisuje kako su u Chase Manhattan banci ažurirali zastareli sistem i usredsredili se na međuljudske probleme vezane za informacionu tehnologiju.

Fischer, G., K. Nakakoji i J. Ostwald (1995), „Supporting the evolution of design artifacts with representations of context and intent", u *Proceedings of DIS95, Symposium on Designing Interactive Systems* (Ann Arbor, MI), New York: ACM, str. 7-15.

Fjeldstad, R. K. i W. T. Hamlen (1979), „Application program maintenance study: A report to our respondents", u *Proceedings of GUIDE 48* (Philadelphia).

Star, ali zanimljiv opšti pregled 25 projekata za obradu podataka u kojima se razmatra podela na vreme za razvoj i vreme za održavanje.

Forgionne, G. A. (1986), *Quantitative Decision Making*. Belmont, CA: Wadsworth.

Forrester, J. (1991), „System dynamics and the lessons of 35 years", Working paper D-42241. Cambridge, MA: Massachusetts Institute of Technology, Sloan School of Management.

Opis upotrebe dinamike sistema od nastanka do danas.

Foushee, H.C., J. K. Lauber, M. M. Baetge i D. B. Acomb (1986), „Crew performance as a function of exposure to high-density short-haul duty cycles," NASA Technical Memorandum 99322, Mofet Fild, CA, NASA Ames Research Center.

Fowler, Martin (1999), *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

Fowler, M. i K. Scott (1999), *UML Distilled: Applying the Standard Object Modeling Language*, 2. izdanje, Reading, MA: Addison-Wesley.

Frakes, William B. i Sadahiro Isoda (1994), „Success factors of systematic reuse", *IEEE Software*, 11(5) (septembar): 15-19.

Uvod specijalnog izdanja posvećenog sistematičnoj primeni višekratne upotrebljivosti.

Frankl, Phyllis, Dick Hamlet, Bev Littlewood i Lorenzo Strigini (1997), „Choosing a testing method to deliver reliability", u *Proceedings of the Nineteenth International Conference on Software Engineering* (Boston, MA), str. 68-78. New York: ACM Press.

Zanimljiv rad o razlikama između testiranja za pronalaženje grešaka i testiranja za poboljšanje pouzdanosti.

Fujitsu Corporation (1987), prepiska sa Rubenom Prieto-Diazom, prema navodima u Braun i Prieto-Díaz (1990).

Fukuda, K. (1994), *The Name of Colors* (*Iro no Namae*) (na japanskom). Tokyo: Shufuno-tomo.

Objašnjava kulturološke konotacije različitih boja. Ove informacije mogu da budu korisne pri razmatranju dizajna korisničkih interfejsa.

Gabb, Andrew P. i Derek E. Henderson (1995), „Navy Specification Study: Report 1-Industry Survey," DSTO-TR-0190, Nacrt 2.0a. Kanbera: Australian Department of Defence, Defence Science and Technology Organisation.

Anketa kompanija uključenih u razvoj i isporuku složenih eksploatacionih sistema baziranih na računarima za potrebe australijske mornarice; sadrži povratne informacije o kvalitetu zahteva prema mornaričkoj specifikaciji.

Gamma, Erich, Richard Helm, Ralph Johnson i John Vlissides (1995), *Design Patterns: Elements of Object-Oriented Software Architecture*. Reading, MA: Addison-Wesley (ovlašćeni prevod: *Gotova rešenja: Elementi objektno orijentisanog softvera*, CET - Beograd, 2002. godina, ISBN 86-7991-153-4).

Zanimljiva i korisna knjiga koja opisuje arhitekturu u smislu šablona koje možemo da otkrijemo.

Gane, C. i T. Sarson (1979), *Structured Systems Analysis: Tools and Techniques*. Englewood Cliffs, NJ: Prentice Hall.

Klasičan tekst o upotrebi strukturne analize kod prikupljanja zahteva.

Garlan, David, Gail E. Kaiser i David Notkin (1992), „Using tool abstraction to compose systems", *IEEE Computer*, 25(6) (jun): 30-38.

Garvin, D. (1984), „What does 'product quality' really mean?" *Sloan Management Review*, (jesen): 25-45.

Opisuje kvalitet proizvoda iz pet perspektiva: transcendentalne perspektive, korisničke perspektive, iz perspektive proizvodnog procesa, iz perspektive proizvoda i iz perspektive vrednosti.

Gerlich, R. i U. Denskat (1994), „A cost estimation model for maintenance and high reuse", u *Proceedings of ESCOM* 1994 (Ivrea, Italija).

German Ministry of Defense (1992), V-Model: Software lifecycle process model, General Reprint No. 250. Bundesminister des Innern, Koordinierungs- und Beratungstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung.

Opis proizvodnog modela koji je upotrebljen u nemačkom ministarstvu odbrane.

Ghezzi, Carlo, Dino Mandrioli, Sandro Morasca i Mauro Pezze (1991), „A unified high-level Petri net formalism for time-critical systems", *IEEE Transactions on Software Engineering* (februar).

Gilb, Tom (1988), *Principles of Software Engineering Management*. Reading, MA: Addison-Wesley.

Gilb, Tom i Dorothy Graham (1993), *Software Inspections*. Reading, MA: Addison-Wesley.

Dobar priručnik o pojmu inspekcije i njenom uvođenju u organizacije.

Gomaa, Hassan (1995), *Software Design Methods for Concurrent and Real-Time Systems*. Reading, MA: Addison-Wesley.

Gordon, V. Scott i James M. Bieman (1995), „Rapid prototyping: Lessons learned", *IEEE Software*, 12(1) (januar): 85-95.

Grady, Robert B. (1997), *Successful Software Process Improvement*. Englewood Cliffs, NJ: Prentice Hall.

Divna knjiga koja se ne zadržava samo na merenjima softvera i objašnjava kako u Hewlett-Packardu rade na unapređenju softvera u celoj kompaniji.

Grady, Robert B. i Deborah Caswell (1987), *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, NJ: Prentice Hall.

Zanimljiva i korisna knjiga sa opisom korporacijskog mernog programa u Hewlett-Packardu.

Grady, Robert B. i Thomas van Slack (1994), „Key lessons in achieving widespread inspection use", *IEEE Software*, 11(4): 46-57.

Objašnjava kako inspekcije postaju standardna praksa u Hewlett-Packardu.

Graham, Dorothy R. (1996a), „Testing object-oriented systems", u *Ovum Evaluates: Software Testing Tools*, (februar). London: Ovum Ltd.

Dobar opšti pregled razlika između testiranja objektno orijentisanih i proceduralnih sistema.

--- (1996b), „Measuring the effectiveness and efficiency of testing", u *Proceedings of Software Testing '96* (Espace Champerret, Pariz, Francuska) (jun).

Greenbaum, J. i M. Kyng (urednici) (1991), *Design at Work: Cooperative Design of Computer Systems*. Hillsdale, NJ: Lawrence Erlbaum.

Griss, Martin i Martin Wasser (1995), „Making reuse work at Hewlett-Packard", *IEEE Software*, 12(1) (januar): 105-107.

Grudin, J. (1991), „Interactive systems: Bridging the gaps between developers and users", *IEEE Computer*, 24(4) (april): 59-69.

Gugliotta, Guy (2004), „Switches failed in crash of Genesis", *Washington Post*, subota 16. oktobar, str. A3.

Guindon, Raymonde, H. Krasner i B. Curtis (1987), „Breakdowns and processes during the early activities of software design by professionals", u *Empirical Studies of Programmers: Second Workshop*, str. 65-82. New York: Ablex.

U ovom istraživanju o projektantima na 19 projekata utvrđuju se uzroci zbog kojih je propao dizajn, koji su pomenuti u poglavlju 5 ove knjige.

Gunning, R. (1968). *The Technique of Clear Writing*. New York: McGraw-Hill.

Uvodi mernu jedinicu za razumljivost koja se zove indeks FOG.

Hall, J. Anthony (1996), „Using formal methods to develop an ATC information system", *IEEE Software*, 13(2) (mart): 66-76.

Opisuje kako se donose odluke o izboru odgovarajućih formalnih metoda za svaku fazu razvoja velikog sistema za kontrolu vazdušnog saobraćaja.

Halstead, Maurice (1977). *Elements of Software Science*. Amsterdam: Elsevier/North Holland.

Klasičan tekst sa (pogrešnom) primenom koncepata iz psihologije na razumevanje programa. Neke od njegovih mernih jedinica su bile korisne, ali neke nisu opravdale njegovu definiciju.

Hamlet, Dick (1992), „Are we testing for true reliability?" *IEEE Software*, 9(4) (jul): 21-27.

Provokativni referat u kojem se tvrdi da pretpostavke koje važe za običnu teoriju pouzdanosti ne važe za softver.

Harel, David (1987), „Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 8: 231-274.

Harrold, Mary Jean i John D. McGregor (1989). Incremental Testing of Object-Oriented Class Structures, tehnički izveštaj. Clemson, SC: Clemson University.

Prikazuje tehniku za testiranje klasa u kojoj se koristi hijerarhijska priroda relacije nasleđivanja i ponovno korišćenje informacija dobijenih testiranjem za testiranje roditeljske klase.

Hatley, D. i I. Pirbhai (1987). *Strategies for Real-Time System Specification*. New York: Dorset House.

Ovaj klasičan rad opisuje proširenja za rad u realnom vremenu kojima su pisci Hatley i Pirbhai proširili strukturnu analizu.

Hatton, Les (1995). *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*. New York: McGraw-Hill.

Izvrsna knjiga u kojoj se opisuju najbolji načini da se upotrebi C za razvoj sistema sa visokim integritetom u kojima je bezbednost kritična.

--- (1997), „Reexamining the fault density-component size connection", *IEEE Software*, 14(2) (mart): 89-97.

Izlaže dokaze da manje komponente sadrže više grešaka nego velike i nagoveštava da bi to mogao biti univerzalan princip softverskog inženjerstva. Konkretno, on smatra da možda postoji granica za najnižu dostižnu gustinu grešaka.

Hatton, Les i T. R. Hopkins (1989), „Experiences with Flint, a software metrication tool for Fortran 77", u *Proceedings of the Symposium on Software Tools* (Durham, UK).

Heimdahl, Mats P. E. i Nancy G. Leveson (1996), „Completeness and consistency in hierarchical state-based requirements", *IEEE Transactions on Software Engineering*, 22(6) (jun): 363-377.

Primenjuje analitičke algoritme i alatke na TCAS II, sistem za izbegavanje sudara u vazdušnom prostoru SAD.

Henry, Joel, Sallie Henry, Dennis Kafura i Lance Matheson (1994), „Improving software maintenance at Martin Marietta", *IEEE Software*, 9(4) (jul): 67-75.

Zanimljiva studija koja prikazuje kako je uz pomoć merenja promenjen način održavanja kod jednog velikog korisnika.

Herbsleb, James, Anita Carleton, James Rozum, J. Siegel i David Zubrow (1994), Benefits of CMM-Based Software Process Improvement: Initial Results, tehnički izveštaj SEI-CMU-94-TR-13. Pittsburgh, PA Software Engineering Institute.

Herbsleb, James, David Zubrow, Dennis Goldenson, Will Hayes i Mark Paulk (1997), „Software quality and the Capability Maturity Model", *Communications of the ACM*, 40(6) (jun): 31-40.

Sumira rezultate studijskih primera i anketa da bi se utvrdili efekti implementiranja modela CMM.

Hershey, John C., Howard C. Kunreuther i Paul Schoemaker (1982), „Sources of bias in assessment procedures for utility functions", *Management Science*, 28: 936-954.

Hetzel, William (1984), *The Complete Guide to Software Testing*. Wellesley, MA: QED Information Sciences.

Hillier, F. S. i G. J. Lieberman (1967), *Introduction to Operations Research*. San Francisko: Holden-Day.

Dobar osnovni tekst o tehnikama za istraživanje operacija koji obuhvata PERT i metod kritične putanje.

Hix, Deborah i H. Rex Hartson (1993), *Developing User Interfaces: Ensuring Usability through Product and Process*. New York: John Wiley.

Hughes, C. E., C. P. Pfleeger i L. Rose (1978), *Advanced Programming Techniques*. New York: John Wiley.

Dobri savĕti za pisanje čvrstog fortranskog koda, ali dobar deo saveta važi za sve jezike.

Hughes, R. T. (1996), „Expert judgment as an estimating method", *Information and Software Technology*, 38(2): 67-75.

Humphrey, W. S. (1989), *Managing the Software Process*. Reading, MA: Addison-Wesley.

--- (1995), *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley.

Humphrey, W. S., T. R. Snyder i R. R. Willis (1991), „Software process improvement at Hughes Aircraft", *IEEE Software*, 8(4) (jul): 11-23.

Hurley, Richard (1983), *Decision Tables in Software Engineering*. New York: John Wiley.

Opisuje kako je Hughes napredovao sa svojstvima i zrelošću nivoa 2 na nivo 3.

IEEE (1983), IEEE Standard 729: Glossary of Software Engineering Terminology. Los Alamitos, CA: IEEE Computer Society Press.

IEEE (1998), IEEE Standard 830-1998: IEEE Recommended Practice for Software Requirements Specification. Los Alamitos, CA: IEEE Computer Society Press.

IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices (2004), „Software Engineering Code of Ethics and Professional Practice," Version 5.2. http://www.computer.org/tab/seprof/code.htm.

International Function Point User Group (1994a), *Function Point Counting Practices Manual*, Release 4.0. Westerville, Ohio: IFPUG.

--- (1994b), *Guidelines to Software Measurement*, Release 1.0. Westerville, Ohio: IFPUG.

International Organization for Standardization (1987) „ISO 9001: Quality systems model for quality assurance in design, development, production, installation and servicing", ISO 9001. Ženeva: ISO.

Standard za merenje kvaliteta svih procesa.

--- (1990), „Quality management and quality assurance standards. Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software", ISO IS 9000-3. Ženeva: ISO.

Standard za merenje kvaliteta softverskih procesa.

--- (1991), „Information technology-Software product evaluation: Quality characteristics and guidelines for their use," ISO/IEC IS 9126. Ženeva: ISO.

Standard za merenje kvaliteta softverskih proizvoda uz pomoć šest karakteristika višeg nivoa.

International Telecommunication Union (1996), „Message Sequence Chart (MSC)", ITU-T Recommendation Z.120 (novembar).

International Telecommunication Union (2002), „Specification and Description Language (SDL)", ITU-T Recommendation Z.100 (avgust).

Ishii, H. (1990), „Cross-cultural communication and computer-supported cooperative work", *Whole Earth Review* (Zima): 48-52.

Isoda, Sadahiro (1992), „Experience report of software reuse project: Its structure, activities and statistical results", u *Proceedings of the Fourteenth International Conference on Software Engineering*. Los Alamitos, CA: IEEE Computer Society Press.

Opisuje okruženja CASE i višekratnu upotrebljivost u kompaniji Nippon Telephone and Telegraph.

Ito, M. i K. Nakakoji (1996), „Impact of culture in user interface design", u J. Nielsen i E. del Galdo (urednici), *International User Interfaces*. London: John Wiley.

Jackson, Michael (1995), *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. Reading, MA: Addison-Wesley.

Divna knjižica namenjena da podstakne na razmišljanje i da podući. Svako kratko poglavlje se odnosi na jedan element analize zahteva i primorava nas da dovedemo u pitanje svoje pretpostavke. Tom DeMarco kaže da je ovo „dosada najbolji rad Michael Jacksona."

Jackson, Michael i Pamela Zave (1995), „Deriving specifications from requirements: An example," u *Proceedings of the Seventeenth International Conference on Software Engineering*, 15-24. Los Alamitos, CA: IEEE Computer Society Press.

Jacky, Jonathan (1985), „The 'Star Wars' defense won't compute", *Atlantic Monthly* (jun): 18-30.

Opis problema koji su se javljali u izgradnji i testiranju softvera za inicijativu strateške odbrane SAD.

Jacobson, Ivar, M. Christerson, P. Jensson i G. Overgaard (1995), *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley.

Jelinski, Z. i P. B. Moranda (1972), „Software reliability research", u *Statistical Computer Performance Evaluation* (izdanje W. Freiburger), New York: Academic Press. 465-84.

Jézéquel, Jean-Marc i Bertrand Meyer (1997), „Design by contract: The lessons of Ariane", *IEEE Computer*, 30(1) (januar): 129-130.

Joint IEEE-CS/ACM Task Force on Computing Curricula (2004), „Computing Curricula-Software Engineering", http://sites.computer.org/ccse/SE2004Volume.pdf (oktobar).

Jones, C. (1997), „Programmer quality and programmer productivity," tehnički izveštaj TR-02.764. Jorktaun Haits, NY: IBM.

--- (1991), *Applied Software Measurement*. New York: McGraw-Hill.

Jones, S., C. Kennelly, C. Mueller, M. Sweezy, B. Thomas i L. Velez (1991), *Developing International User Information*. Bedford, MA: Digital Press.

Joos, Rebecca (1994), „Software reuse at Motorola", *IEEE Software*, 11(5) (septembar): 42-47.

Opisuje šta je uspelo, a šta nije u njihovom programu višekratne upotrebljivosti.

Joyce, Edward (1989), „Is error-free software possible?" *Datamation* (18. februar).

Jung, Carl (1959), *The Basic Writing of C. G. Jung*. New York: Modern Library.

Sadrži opis strukture ličnosti na dve skale: introvertne/ekstrovertne i intuitivne/racionalne. Ovaj okvir je koristan za tumačenje međusobnih odnosa osoblja na projektu.

Kaiser, Gail E., Peter H. Feiler i S. S. Popovich (1988), „Intelligent assistance for software development and maintenance", *IEEE Software*, 5(3): 40-49.

Opis jezika MARVEL za modelovanje procesa.

Kaner, Cem, Jack Falk i Hung Quoc Nguyen (1993), *Testing Computer Software*, 2. izdanje, London: International Thomson Press.

Kaner, Cem i David Pels (1998), *Bad Software*. New York: John Wiley.

Kaplan, R. i D. Norton (1992), „The balanced scorecard: Measures that drive performance", *Harvard Business Review* (januar-februar).

Kauffman, R. i R. Kumar (1993), „Modeling Estimation Expertise in Object-Based CASE Environments", New York: New York University, Stern School of Business Report.

Uvod u korišćenje tačaka na objektima za merenje dimenzija.

Kellner, Marc I. i H. Dieter Rombach (1990), „Comparisons of software process descriptions", u *Proceedings of the Sixth International Software Process Workshop: Support for the Software Process* (Hakodate, Japan) (oktobar).

Primer vežbe za modelovanje sa primenom 18 tehnika modelovanja procesa na isti problem.

Kemerer, C. F. (1989), „An empirical validation of software cost estimation models", *Communications of the ACM*, 30(5) (maj): 416-429.

Dobro vrednovanje nekoliko modela za procenu troškova i njihove (ne)preciznosti.

Kensing, F. i A. Munk-Madsen (1993), „PD: Structure in the toolbox", *Communications of the ACM*, 36(4) (jun): 78-85.

Kernighan, B. W. i P. J. Plauger (1976), *Software Tools*. Reading, MA: Addison-Wesley.

--- (1978), *The Elements of Programming Style*. New York: McGraw-Hill.

Kit, Ed (1995), *Software Testing in the Real World: Improving the Process*. Reading, MA: Addison-Wesley.

Kitchenham, Barbara A. i Kări Känsälа (1993). „Inter-item correlations among function points", u *Proceedings of the First International Symposium on Software Metrics* (Baltimore, MD), Los Alamitos, CA: IEEE Computer Society Press.

Kitchenham, Barbara A. i Steven Linkman (1997), „Why mixed VV&T strategies are important", *Software Reliability and Metrics Club Newsletter* (leto): 9-10.

Kitchenham, Barbara A., Stephen G. MacDonell, Lesley M. Pickard i Martin J. Shepperd (2000), „What accuracy statistics really measure", Bournemouth University tehnički izveštaj, jun.

Kitchenham, Barbara A. i Shari Lawrence Pfleeger (1996), „Software quality: The elusive target", *IEEE Software*, 13(1) (januar): 12-21.

Uvod za specijalno izdanje posvećeno kvalitetu softvera; ovaj članak sadrži pregled nekih uobičajenih modela za procenu kvaliteta softvera i postavlja pitanja o stvarnom značenju pojma „kvalitet softvera."

Kitchenham, Barbara A., Lesley Pickard i Shari Lawrence Pfleeger (1995), „Case studies for method and tool evaluation", *IEEE Software*, 12(4) (jul): 52-62.

Kitchenham, Barabara A. i N. R. Taylor (1984), „Software cost models", *ICL Technical Journal*, 4(3):73-102.

Klein, Gary (1998), *Sources of Power*. Cambridge, MA: MIT Press.

Kleindorfer, Paul, Howard Kunreuther i Paul Schoemaker (1993), *Decision Sciences: An Integrative Perspective*. Cambridge UK: Cambridge University Press.

Knight, John i Nancy Leveson (1986), „An empirical study of failure probabilities in multi-version software", u *Digest of the Sixteenth International Symposium on Fault-tolerant Computing*, str. 165-70. Los Alamitos, CA: IEEE Computer Society Press.

Procenjuje programiranje sa *n* verzija, dokazuje da *n* različitih dizajna može da sadrži mnogo istovetnih grešaka.

Knight, John, Nancy Leveson, et al. (2000), „ACM Task Force on Licensing of Software Engineers Working on Safety-Critical Software", Nacrt (jul).

http://www.acm.org/serving/se_policy/safety_critical.pdf.

Krasner, H., B. Curtis i N. Iscoe (1987), „Communication breakdowns and boundary-spanning activities on large programming projects", u *Empirical Studies of Programmers*: Second Workshop, str. 47-64. New York: Ablex Publishing.

Rezultati intervjua sprovedenih tokom 19 velikih softverskih razvojnih projekata istražujući probleme u vezi sa timovima i projektima u MCC-u (*Microelectronics and Computer Technology Consortium*). Opisuje tipične probleme u komunikaciji kada se rade veliki programerski projekti, kulturološke razlike i razlike u okruženjima koje sprečavaju efikasnu komunikaciju među grupama, i aktivnosti za premošćavanje granica da bi se postigla koordinacija pet aktuelnih komunikacionih mreža. Predlaže efikasnu koordinaciju u projektu, uključujući upotrebu alatki za zajednički rad na dizajniranju softvera uz pomoć računara.

Krasner, Herb, Jim Terrel, Adam Linehan, Paul Arnold i William H. Ett (1992), „Lessons learned from a software process modeling system", *Communications of the ACM*, 35(9) (septembar): 91-100.

Opisuje iskustva sa SPMS-om, sistemom za modeliranje softverskih procesa.

Krauss, R. M. i S. R. Fussell (1991), „Constructing shared communicative environments", u L. B. Resnick, J. M. Levine i S. D. Teasley (urednici), *Perspectives on Socially Shared Cognition*, str. 172-200. Washington, DC: American Psychological Association.

Kumar, Kuldeep (1990), „Post-implementation evaluation of computer-based information systems: Current practices", *Communications of the ACM*, 33(2) (februar): 203-212.

Kunde, Diana (1997), „For those riding technology's wave, a new managerial style", *Washington Post*, 9. februar, str. H5.

Preporučuje manje strukturisano upravljanje projektima da se ne bi gušila kreativnost dizajnera.

Lai, Robert Chi Tau (1991), Process Definition and Modeling Methods, tehnički izveštaj SPC-91084-N. Herndon, VA: Software Productivity Consortium.

Tehnički izveštaj sa definicijom procesa i njegovih sastavnih delova, koji opisuje jednu notaciju za modelovanje procesa i i zatim izlaže jedan opširan primer.

Lanergan, R. G. i C. A. Grasso (1984), „Software engineering with reusable designs and code", *IEEE Transactions on Software Engineering*, SE 10(4) (septembar): 498-501.

Opisuje jedan od prvih projekata višekratne upotrebljivosti u Raytheonu.

Lanubile, Filippo (1996), „Why software reliability predictions fail", *IEEE Software*, 13(4) (jul): 131-137.

Zanimljivo poređenje različitih tehnika predvidanja gde se pokazuje da nijedna nije bila mnogo uspešna.

Larman, Craig (1998), *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Upper Saddle River, NJ: Prentice Hall.

Lederer, Albert L. i Jayesh Prasad (1992) „Nine management guidelines for better cost estimating", *Communications of the ACM*, 35(2) (februar): 50-59.

Opisuje rezultate opsežne ankete o postupcima za procenu troškova. Obuhvata informacije o tome koliko često rukovodioci projektuju troškove koristeći alatke za procenu troškova.

Lee, Richard C. i William M. Tepfenhart (1997), UML and C++ : A Practical Guide to Object-Oriented Development. Upper Saddle River, NJ: Prentice Hall.

Lehman, M. M. (1980), „Programs, life cycles and the laws of software evolution", u *Proceedings of the IEEE*, 68(9)(septembar): 1060-1076.

Leveson, Nancy (1996), *Safeware*. Reading, MA: Addison-Wesley.

--- (1997), „Software safety in embedded computer systems", *Communications of the ACM*, 40(2) (februar): 129-131.

Leveson, Nancy G. i Clark S. Turner (1993), „An investigation of the Therac-25 accidents", *IEEE Computer*, 26(7) (jul): 18-41.

Konačna analiza čuvenog otkaza softvera koje je odneo ljudske živote.

Li, W. i S. Henry (1993), „Object-oriented metrics that predict maintainability", *Journal of Systems and Software*, 23(2) (februar): 111-122.

Liebman, Bonnie (1994), „Non-trivial pursuits: Playing the research game", *Nutrition Action Healthletter*. Center for Science in the Public Interest, 1875 Connecticut Avenue NW, Suite 300, Washington, DC 20009-5728 (oktobar).

Lientz, B. P. i E. B. Swanson (1981), „Problems in application software maintenance", *Communications of the ACM*, 24(11): 763-769.

Jedna od prvih anketa u kojima su se ispitivale karakteristike održavanja.

Lim, Wayne (1994), „Effects of reuse on quality, productivity and economics", *IEEE Software*, 11(5) (septembar): 23-30.

Opisuje rezultate primene višekratne upotrebljivosti u Hewlett-Packardu. Temeljna i zanimljiva studija.

Lindvall, Mikael i Kristian Sandahl (1996), „Practial implications of traceability", *Software: Practice and Experience*, 26(10) (oktobar): 1161-1180.

Pfleegerove i Bohnerove (1990) tehnike koje omogućuju praćenje primenjene na jednom sistemu u kompaniji Ericsson Radio Systems. Dokazuje da postoje različite vrste objekata i relacija koje se mogu pratiti i da samo formiranje veza već otkriva značajne informacije (a često i probleme).

Linger, Richard C. (nedatirano) „Cleanroom Software Engineering for Zero-Defect Software", tehnički izveštaj. Geitersburg, MD: IBM Cleanroom Software Technology Center.

Linger, Richard C. i R. Alan Spangler (1992), „The IBM Cleanroom software engineering technology transfer program", u *Proceedings of the Sixth SEI Conference on Software Engineering Education* (San Diego, CA).

Linstone, H. i M. Turoff (1975),*The Delphi Method: Techniques and Applications*. Reading, MA: Addison-Wesley.

Lions, J. L., et al. (1996), Ariane 5 Flight 501 Failure: Report by the Inquiry Board. European Space Agency.

Izveštaj postavljen na Web sa zaključcima istražne komisije o propasti leta Ariane-5. Zanimljiv opis softverskog dizajna, tehnika testiranja i predloženih rešenja.

Lipke, W. H. i K. L. Butler (1992), „Software process improvement: A success story", *Crosstalk: Journal of Defense Software Engineering*, 38 (novembar): 29-31.

Littlewood, Bev (1991), „Limits to evaluation of software dependability", u N. Fenton i B. Littlewood (urednici), *Software Reliability and Metrics*. Amsterdam: Elsevier.

Lorenz, M. i J. Kidd (1994), *Object-Oriented Software Metrics*. Prentice Hall, Upper Saddle River, NJ.

Lutz, Robyn R. (1993a), „Targeting safety-related errors during requirements analysis", Proceedings of SIGSOFT Symposium on the Foundations of Software Engineering, *ACM Software Engineering Notes*, 18(5): 99-105.

Lutz, Robyn (1993b), „Analyzing software requirements errors in safety-critical, embedded systems", u Proceedings of the IEEE International Symposium on Requirements Engineering, Los Alamitos, CA: IEEE Computer Society Press. 126-133.

Lyu, Michael (urednik) (1996), Handbook of Software Reliability Engineering. Los Alamitos, CA: IEEE Computer Society Press i New York: McGraw-Hill.

Odličan pregled najnovijih razmišljanja o merenjima, modelovanju i predviđanju pouzdanosti.

Manchester, William (1983), *The Last Lion*. Boston: Little Brown.

Prva od tri knjige sa biografijom Winstona Churchilla. Ova knjiga je značajna između ostalog po svom opisu Churchilla kao intuitivnog introverta.

Marca, David A. i Clement L. McGowan (1988), *SADT: Structured Analysis and Design Technique*. New York: McGraw-Hill.

Temeljiti uvod u SADT (strukturnu analizu i tehnike dizajna) sa mnogo primera tog procesa i korišćenja notacije.

Marcus, A. (1993), „Human communications issues in advanced user interfaces", *Communications of the ACM*, 36(4) (april): 101-109.

Martin, Robert C. (2000), „Extreme programming development through dialog", *IEEE Software* 17(4) (jul/avgust): 12-13.

Martin, Robert C. (2003), *Agile Software Development: Principles, Patterns and Principles*. Upper Saddle River, NJ: Prentice Hall.

MathSoft (1995), S-PLUS User's Manual, Version 3.3 for Windows. Seattle, WA: MathSoft Corporation.

Matos, Victor i Paul Jalics (1989), „An experimental analysis of the performance of fourth generation tools on PCs", *Communications of the ACM* (novembar).

Mays, R., C. Jones, G. Holloway i D. Studinski (1990), „Experiences with defect prevention." *IBM Systems Journal*, 29.

McCabe, T. (1976), „A software complexity measure", *IEEE Transactions on Software Engineering*, SE 2(4): 308-320.

U ovom referatu se prvi put pojavljuje ciklomatski broj (broj nezavisnih kontura).

McCabe, T. i C. W. Butler (1989), „Design complexity measurement and testing", *Communications of the ACM*, 32(12): 1415-1425.

McCall, J. A., P. K. Richards i G. F. Walters (1977), Factors in Software Quality, tomovi 1, 2 i 3, AD/A-049-014/015/055. Springfild, VA: National Technical Information Service.

Jedan od prvih radova u kojima se predlaže model kvaliteta, u ovom radu se za merenje kvaliteta softvera predlaže jedan pristup sa faktorima i kriterijumima.

McClure, Carma (1997), *Software Reuse Techniques*. Englewood Cliffs, NJ: Prentice Hall.

McConnell, Steve (1993), *Code Complete*. Redmond, WA: Microsoft Press.

Dobri i razložni saveti za dizajniranje i implementaciju.

McCracken, D. D. i M. A. Jackson (1981), „A minority dissenting opinion", u W. W. Cotterman et al. (urednici). *Systems Analysis and Design: A Foundation for the 1980s*, str. 551-553. New York: Elsevier.

McCue, G. (1978), „Architectural design for program development", *IBM Systems Journal*, 17(1).

Opisuje minimalnu količinu prostora koja je projektantu potrebna za efikasan rad.

McDermid, J. A. i D. J. Pumphrey (1995), A Development of Hazard Analysis to Aid Software Design, tehnički izveštaj. York, UK: University of York, Department of Computer Science, Dependable Computing Systems Centre.

Mellor, Peter (1992), Data Collection for Software Reliability Measurement, Software Reliability Measurement series, part 3. London: City University, Centre for Software Reliability.

Napomene uz komplet od tri video trake o pouzdanosti softvera koje se mogu nabaviti od Centra za pouzdanost softvera na adresi bi@csr.city.ac.uk.

Meyer, Bertrand (1988), *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall (ovlašćeni prevod: *Objektno orijentisano konstruisanje softvera*, CET - Beograd, 2003. godina, ISBN 86-7991-199-2).

--- (1992a), „Applying 'design by contract'", *IEEE Computer*, 25(10) (oktobar): 40-51.

--- (1992b), *Eiffel: The Language*. Englewood Cliffs, NJ: Prentice Hall.

--- (1993), „Systematic concurrent object-oriented programming", *Communications of the ACM*, 36(9) (septembar): 56-80.

Miller, Douglas R. (1986), „Exponential order statistical models of software reliability growth", *IEEE Transactions on Software Engineering*, SE 12(1): 12-24.

Mills, Harlan D. (1972), On the Statistical Validation of Computer Programs, tehnički izveštaj FSC-72-6015. Geitersburg, MD: IBM Federal Systems Division.

Uvodi ideju sejanja grešaka radi procenjivanja preostalih grešaka u kodu.

--- (1988), „Stepwise refinement and verification in box-structured systems", *IEEE Computer*, 21(6) (jun): 23-36.

Mills, Harlan, Michael Dyer i Richard Linger (1987), „Cleanroom software engineering", *IEEE Software*, 4(5) (septembar): 19-25.

Opšti pregled IBM-ovog pristupa čiste sobe.

Mills, Harlan, Richard Linger i Alan R. Hevner (1987), „Box-structured information systems", *IBM Systems Journal*, 26(4): 395-413.

Mills, Simon (1997), „Automated testing: Various experiences", u *Proceedings of the Fourteenth International Conference and Exposition on Testing Computer Software* (Washington, DC).

Zanimljiv opis problema koji su se javljali prilikom testiranja jednog sistema za ponudu automobilskog osiguranja.

Misra, Santosh i Paul Jalics (1988), „Third generation vs. fourth generation software development", *IEEE Software*, 5(4) (jul).

Miyazaki, Y. i K. Mori (1985), „COCOMO evaluation and tailoring", u *Proceedings of the Eighth International Software Engineering Conference* (London). Los Alamitos, CA: IEEE Computer Society Press.

Moad, Jeff (1995), „Time for a fresh approach to ROI", *Datamation* (februar 15).

Möller, Karl i Daniel Paulish (1993), „An empirical investigation of software fault distribution", u *Proceedings of CSR 93*, Amsterdam: Chapman and Hall.

Prikazuje rezultate iz kojih se vidi da manji moduli imaju veću gustinu grešaka od velikih modula.

Moore, Geoffrey (1991), *Crossing the Chasm*. New York: HarperBusiness.

Musa, John D. (1979), Software Reliability Data, tehnički izveštaj. Rome, NY: Rome Laboratories, Data Analysis Center for Software.

Musa, John D., Anthony Iannino i Kazuhira Okumoto (1990), *Software Reliability: Measurement, Prediction, Application*. New York: McGraw-Hill.

Myers, Glenford J. (1976), *Software Reliability*. New York: John Wiley.

Jedan od prvih tekstova u kojima se testiranje i pouzdanost softvera prikazuje pomoću empirijskih podataka.

--- (1979), *The Art of Software Testing*. New York: John Wiley.

I dalje jedna od najvrednijih knjiga u kojima se opisuje filozofija testiranja.

Nakakoji, K. (1994), „Crossing the cultural boundary", *Byte*, 19(6) (jun): 107–109.

Opisuje koliko je važno da se pri dizajnu interfejsa shvataju kulturološke razlike.

National Science Foundation (1983), *The Process of Technological Innovation*. Washington, DC: NSF.

Opisuje odnos između strukture upravljanja i karakteristika kod uspešnih projekata.

Netfocus: Software Program Manager's Network (1995), Washington, DC: Department of the Navy (januar).

U ovom biltenu se opisuje izrada „kontrolne table" sa nekoliko ključnih mera. Kontrolna tabla je namenjena rukovodiocima projekata radi „upravljanja" softverskim razvojnim projektom, po kojoj će rukovodilac znati kada je proizvod spreman za predaju kupcu.

Newsbytes Home Page (1996), „Computer blamed for $500 million Ariane explosion", Pariz (jun 6).

Nippon Electric Company (1987), u prepisci sa Rubenom Prieto-Diazom, prema navodima u Braun i Prieto-Díaz (1990) (jun).

Nosek, J. (1998), „The case for collaborative programming", *Communications of the ACM* 41(3) (mart): 105–108.

Notkin, David (2000), „Software Engineering Licensing and Certification," rad predstavljen na konferenciji Udruženja za računarska istraživanja (*Computing Research Association*) u Snowbirdu, Utah.

http://www.cra.org/Activities/snowbird/00/notkin-crawk3-5.pdf.

Ntafos, S. C. (1984), „On required element testing", *IEEE Transactions on Software Engineering*, 10: 795-803.

Upoređuje relativnu efikasnost u otkrivanju grešaka za nekoliko različitih strategija testiranja.

Nuseibeh, Bashar (1997), „Ariane 5: Who dunnit?" *IEEE Software*, 14(3) (maj): 15-16.

Analizira se eksplozija na Ariane-5 iz aspekta različitih aktivnosti u životnom ciklusu. Zaključak je da bi upravljanje rizikom bilo najbolji način da se osnovni problem otkrije u ranijim fazama.

Nuseibeh, Bashar, Joe Kramer i Anthony Finkelstein (1994), „A framework for expressing the relationships between multiple views in requirements specification", *IEEE Transactions on Software Engineering*, 20(10) (oktobar): 760-773.

Object Management Group (2003), „OMG Unified Modeling Language Specification," Version 1.5 (mart).

Olsen, Neil (1993), „The software rush hour", *IEEE Software*, 10(5) (septembar): 29-37.

Zanimljiva rasprava o tome kako merenja mogu da pomognu u upravljanju razvojnim procesom.

Oman, Paul i J. Hagemeister (1992), „Metrics for assessing software system maintainability", u *Proceedings of the International Conference on Software Maintenance*. Los Alamitos, CA: IEEE Computer Society Press. str. 337-344.

Opisuje hijerarhijski model održavanja, podeljen na tri dimenzije.

Oppenheimer, Todd (1997), „The computer delusion", *Atlantic Monthly* (jul): 45-62.

Dobar opis prednosti i mana učenja pomoću računara.

Osterweil, Leon (1987), „Software processes are software too", u *Proceedings of the Ninth IEEE International Conference on Software Engineering*, str. 2-13. Los Alamitos, CA: IEEE Computer Society Press.

Kontroverzni referat sa idejom da bi, uz dovoljno razumevanja, proces razvoja softvera mogao da se opiše procesnim jezikom i da bi zatim proces mogao da se izvrši kao program.

Padberg, Frank i Matthias M. Müller (2003), „Analyzing the cost and benefit of pair programming", u *Proceedings of the Ninth International Software Metrics Symposium*, Sidni, Australija (septembar), str. 166-177. Los Alamitos, CA: IEEE Computer Society Press.

Parikh, G. i N. Zvegintzov (1983), *Tutorial on Software Maintenance*. Los Alamitos, CA: IEEE Computer Society Press.

Mada ovo nije nova publikacija, ona sadrži informacije koje i dalje važe za projekte održavanja.

Parnas, David (1972), „On criteria to be used in decomposing systems into modules", *Communications of the ACM*, 15(12) (decembar).

Opisuje pojmove apstrakcije i sakrivanja informacija.

--- (1985), „Software aspects of strategic defense systems", *Datamation* , 28(12) (decembar).

Opisuje poteškoće u obezbedivanju da ova vrsta sistema funkcioniše prema zahtevima, uz prihvatljivu pouzdanost i bezbednost.

Parnas, David L. (1992), „Tabular Representation of Relations," tehnički izveštaj Communications Research Laboratory br. 260, CRL (oktobar).

Parnas, David L. (2000), „Two positions on licensing", stav iznet na panel diskusiji u *Proceedings of the 4th IEEE International Conference on Requirements Engineering* (jun).

Parris, Kathy V.C. (1996), „Implementing accountability", *IEEE Software*, 13(jul)(4): 83-93.

Opisuje upravljanje softverskim projektom za avion FX-16 u Ministarstvu odbrane SAD.

Parrish, Allen, Randy Smith, David Hale i Joanne Hale (2004), „A field study of developer pairs: Productivity impacts and implications", *IEEE Software* 21(5) (septembar/oktobar): 76-79.

Paulk, Mark, B. Curtis, M. B. Chrissis i C. V. Weber (1993a), „Capability maturity model for software, version 1.1," tehnički izveštaj SEI-CMU-93-TR-24. Pittsburgh, PA: Software Engineering Institute.

--- (1993b), „Key practices of the capability maturity model, version 1.1," tehnički izveštaj SEI-CMU-93-TR-25. Pittsburgh, PA: Software Engineering Institute.

Perkins, David (2001), *The Eureka Effect*. New York: W. W. Norton.

Perry, Dewayne E. i Gail E. Kaiser (1990), „Adequate testing and object-oriented programming", *Journal of Object-Oriented Programming*, 2 (januar/februar): 13-19.

Ispituje Weyukerove aksiome testiranja i dokazuje da postizanje višekratne upotrebljivosti objekata nije tako lako kao što izgleda. Neke objekte je neophodno temeljito ponovo testiranje.

Perry, Dewayne i Carol Steig (1993), „Software faults in evolving a large, real-time system: a case study," 4th European Software Engineering Conference (ESEC 93), Garmiš, Nemačka, septembar 1993.

Perry, William (1995), *Effective Methods for Software Testing*. New York: John Wiley.

Peterson, James (1977), „Petri Nets", *ACM Computing Surveys*, 9(3) (septembar): 223-252.

Petroski, Henry (1985), *To Engineer Is Human: The Role of Failure in Good Design*. New York: Petrocelli Books.

Pfleeger, Charles P. (1997a), „The fundamentals of information security", *IEEE Software*, 14(1) (januar): 15-16, 60.

Zanimljiv članak o tome kako se u većini zahteva podrazumeva dobronamerno okruženje, ali se za bezbednost podrazumeva da je okruženje zlonamerno. Objašnjava kako da se bezbednost uklopi u razvoj.

--- i Shari Lawrence Pfleeger (2003), *Security in Computing*, 3. izdanje. Upper Saddle River NJ: Prentice Hall.

Klasičan tekst koji obrađuje ključna pitanja u vezi sa računarskom bezbednošću danas, uključujući mreže i kriptovanje, i predlaže kako da se opisuju i implementiraju bezbednosni zahtevi.

Pfleeger, Shari Lawrence (2000), „Risky business: What we have yet to learn about software risk management", *Journal of Systems and Software*, 53(3) (septembar).

Razmatra kako se rizicima upravlja u drugim disciplinama, kao što su javni poslovi i poslovi zaštite životne sredine i predlaže načine za unapređenje upravljanja rizicima na softverskim proizvodima.

--- (1999a), „Albert Einstein and empirical software engineering", *IEEE Computer*, 32(10) (oktobar).

Bavi se potrebom da tehnologija ide napred dok mi razmatramo njenu efikasnost.

--- (1999b), „Understanding and improving technology transfer in software engineering", *Journal of Systems and Software*, 52(2) (jul).

--- (1996), „Measuring reuse: A cautionary tale", *IEEE Software*, 13(4) (jul): 118–127.

Opisuje iskustvo sa ponovnim korišćenjem u Amalgamated Inc., koji se sastoji od nekoliko organizacija koje su pokušale (sa različitim uspehom) da implementiraju višekratnu upotrebljivost. Dokazuje kako se važne poslovne odluke moraju doneti pre nego što se pređe na pitanja višekratne upotrebljivosti.

--- (1991), „Model of software effort and productivity", *Information and Software Technology*, 33(3) (april): 224–232.

Opis modela za procenu rada na osnovu broja objekata i metoda.

Pfleeger, Shari Lawrence i Shawn Bohner (1990), „A framework for maintenance metrics", u *Proceedings of the Conference on Software Maintenance* (Orlando, FL). Los Alamitos, CA: IEEE Computer Society Press.

Pfleeger, Shari Lawrence, Norman Fenton i Stella Page (1994), „Evaluating software engineering standards", *IEEE Computer*, 27(9) (septembar): 71–79.

Dokazuje kako većina standarda u softverskom inženjerstvu predstavlja samo smernice i prikazuje studijski primer iz Engleske u kojem se procenjivala efikasnost standarda za kodiranje i održavanje.

Pfleeger, Shari Lawrence i Les Hatton (1997), „Investigating the influence of formal methods", *IEEE Computer* 30(2) (februar): 33–43.

Studijski primer korišćenja formalnih metoda u sistemu za podršku kontrole vazdušnog saobraćaja. Opisuje kako su formalne metode uticale na kvalitet proizvoda, ali takođe prikazuje iskustva u načinu sprovođenja ovakvih studija.

Pfleeger, Shari Lawrence i Clement L. McGowan (1990), „Software metrics in the process maturity framework", *Journal of Systems and Software*, 12(1): 255–261.

Dokazuje uticaj različitih stepena zrelosti procesa na nivoe vidljivosti i merljivost

Pfleeger, Shari Lawrence i Winifred Menezes (2000), „Technology transfer: Marketing technology to software practitioners", *IEEE Software* 17(1) (januar/februar).

Pfleeger, Shari Lawrence, Martin Shepperd i Roseanne Tesoriero (2000), „Decisions and Delphi: The dynamics of group estimation", u *Proceedings of the Brazilian Software Engineering Symposium*; João Pessoa, oktobar.

Pfleeger, Shari Lawrence i David W. Straight (1985), *Introduction to Discrete Structures*. New York: John Wiley.

Uvodni tekst koji opisuje matematiku potrebnu za razumevanje računarskih nauka.

Polanyi, M. (1996), *The Tacit Dimension*. Garden Siti, NY: Doubleday.

Polyà, George (1957), *How to Solve It*, 2. izdanje, Prinston, NJ: Princeton University Press.

Porter, Adam i Richard Selby (1990), „Empirically-guided software development using metric-based classification trees", *IEEE Software*, 7(2) (mart): 46–54.

Opisuje upotrebu jedne statističke tehnike za utvrđivanje vrste merenja koje će najbolje predvideti ishod. Veoma korisno za svođenje velikog skupa prikupljenih mera na one koje obezbeduju najkorisnije informacije.

Porter, Adam, Harvey Siy, A. Mockus i Lawrence Votta (1998), „Understanding the sources of variation in software inspections", 7(1): 41–79.

Price, Jonathan (1984), *How to Write a Computer Manual*. Menlo Park, CA: Benjamin-Cummings.

Odličan opis kako treba pisati korisničku dokumentaciju.

Prieto-Díaz, Rubén (1987), „Domain analysis for reusability", u *Proceedings of COMPSAC 87*. Los Alamitos, CA: IEEE Computer Society Press.

--- (1991), „Making software reuse work: An implementation model", *ACM SIGSOFT Software Engineering Notes*, 16(3): str. 61-68.

--- (1993), „Status report: Software reusability", *IEEE Software*, 10(3) (maj): 61-66.

Prieto-Díaz, Rubén i Peter Freeman (1987), „Classifying software for reusability", *IEEE Software*, 4(1) (januar): 6-17.

Putnam, L. H. i Ware Myers (1992), *Measures for Excellence: Reliable Software on Time, within Budget*. Englewood Cliffs, NJ: Yourdon Press.

Queen's Printer for Ontario (2000), „Professional Engineers Act, R.R.O. 1990, Regulation 941, as amended", u *Revised Regulations of Ontario*, Toronto, Ont.

Redwine, S. i W. Riddle (1985), „Software technology maturation", u *Proceedings of the Eighth International Conference on Software Engineering*, str. 189-200. Los Alamitos, CA: IEEE Computer Society Press.

Reiss, S. P. (1990), „Connecting tools using message passing in the Field Environment", *IEEE Software*, 7(4) (jul): 57-66.

Opisuje sistem u kojem se u dizajnu koristi implicitno pozivanje.

Rensburger, B. (1985), „The software is too hard", *Washington Post National Weekly Edition*, novembar 11, str. 10-11.

Novinski izveštaj o razlozima zbog kojih je softver Zvezdanih ratova (inicijativa strateške odbrane) u SAD bio previše složen da bi se potpuno testirao.

Richards, F. R. (1974), Computer Software: Testing, Reliability Models and Quality Assurance, tehnički izveštaj NPS-55RH74071A. Monterey, CA: Naval Postgraduate School.

Opisuje tehniku za procenu poverenja u softver na osnovu istorije grešaka.

Rittel, H. W. J. i M. M. Webber (1984), „Planning problems are wicked problems", u N. Cross (izdanju), *Developments in Design Methodology*, str. 135-144. New York: John Wiley.

Robertson, James i Suzanne Robertson (1994), Complete Systems Analysis: The Workbook, the Textbook, the Answers. New York: Dorset House.

Komplet od dve knjige koji pruža temeljita osnovna znanja o glavnim tehnikama za analizu zahteva. Sadrži vežbe i potpune odgovore. (Ponovo izdato 1998. u jednoj knjizi.)

Robertson, Suzanne i James Robertson (1999), *Mastering the Requirements Process*. Reading, MA: Addison-Wesley.

U ovoj knjizi se nalazi kompletan opis obrade zahteva na osnovu modela obrade zahteva Robertson-Volere. Pošto Robertsonovi i dalje ažuriraju svoj model, trenutnu verziju možete da vidite na veb lokaciji Atlantic Systems Guilda, http://www.systemsguild.com.

Robinson, M. i L. Bannon (1991), „Questioning representations", u L. Bannon, M. Robinson i K. Schmidt (urednici), *Proceedings of the Second ECSCW'91*, str. 219-233. Amsterdam: Kluwar.

Rogers, Everett M. (1995), Diffusion of Innovations, 4. izdanje. New York: Free Press.

Klasičan tekst o transferu tehnologije. Opisuje pet tipova ljudi koji prihvataju tehnologiju i objašnjava kako treba motivisati svaki tip.

Rook, Paul (1993), *Risk Management for Software Development*, ESCOM Tutorial.

Dobra sinteza pristupa za upravljanje rizikom.

Rosenberg, Jarrett (1998), „Five easy steps to systematic data handling", *IEEE Software* 15(1) (januar): 75–77.

Ross, D.T. (1977), „Structured analysis (SA): A language for communicating ideas", *IEEE Transactions on Software Engineering*, SE 3(1) (januar): 16–34.

Prvi rad kojim je istraživačkoj zajednici predstavljen Softechov SADT.

--- (1985), „Applications and extensions of SADT", *IEEE Computer*, 18(4) (april): 25–34.

Rouquet, J. C. i P. J. Traverse (1986), „Safe and reliable computing on board the Airbus and ATR aircraft", u *Proceedings of the Fifth IFAC Workshop on Safety of Computer Control Systems*, W. J. Quirk (urednik), str. 93–97. Oxford: Pergamon Press.

Rout, T. P. (1995) „SPICE: A framework for software process assessment", *Software Process: Improvement and Practice*, 1(1) (avgust): 57–66.

Royce, W. E. (1990), „TRW's Ada process model for incremental development of large software systems", u *Proceedings of the Twelfth International Conference on Software Engineering*, str. 2–11. Los Alamitos, CA: IEEE Computer Society Press.

Izveštaji o korišćenju Boehmovih ključnih prekretnica i modela Theory W.

Royce, W. W. (1970), „Managing the development of large software systems: Concepts and techniques", u *Proceedings of WESCON* (avgust), tom 14, str. A–1 to A–9.

Prva publikacija u kojoj je pomenut model vodopada.

Rugg, D. (1993), „Using a capability evaluation to select a contractor", *IEEE Software*, 10(4) (jul): 36–45.

Ruhl, M. i M. Gunn (1991), Software Reengineering: A Case Study and Lessons Learned, NIST Special Publication 500–193. Geitersburg, MD: National Institute of Standards and Technology.

Izveštaji o rezultatima prepravljanja preko 13.000 redova koda u COBOL-u.

Rumbaugh, James, M. Blaha, W. Premerlani, F. Eddy i W. Lorenson (1991), *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall.

Sveobihvatan priručnik za OMT tehniku za upravljanje objektima razvijenu u korporaciji Rational.

Russo, P. i S. Boor (1993), „How fluent is your interface? Designing for international users", u *Proceedings of the Conference on Human Factors in Computing Systems* (INTERCHI'93), str. 342–347.

Sackman, H. H., W. J. Erikson i E. E. Grant (1968), „Exploratory experimental studies comparing online and offline programming performance", *Communications of the ACM*, 11(1) (januar): 3–11.

Interesantna studija koja pokazuje da među programerima produktivnost može da se kreće od 10 do 1.

Saiedian, H. i R. Kuzara (1995), „SEI capability maturity model's impact on contractors", *IEEE Computer*, 28(1) (januar): 16–26.

Sammet, Jean (1969), *Programming Languages: History and Fundamentals*. Englewood Cliffs, NJ: Prentice Hall.

Klasičan tekst sa objašnjenjem pitanja vezanih za dizajn programskih jezika.

Samson, B., D. Ellison i P. Dugard (1997), „Software cost estimation using an Albus Perceptron (CMAC)", *Information and Software Technology*, 39(1–2).

Primenjuje neuronsku mrežu na skup podataka COCOMO radi procene troškova.

Samuelson, Pamela (1990), „Reverse-engineering someone else's software: Is it legal?" *IEEE Software*, 7(1) (januar): 90-96.

Sauer, Chris, et al., (2000), „The effectiveness of software development technical reviews: A behaviorally motivated program of research", *IEEE Transactions on Software Engineering*, 26(1): 1-14.

Sawyer, K. (1985), „The mess at the IRS", *Washington Post National Weekly Edition*, novembar 11, str. 6-7.

Novinarski opis poteškoća u izgradnji novog softverskog sistema za Upravu prihoda SAD.

Scharer, L. (1983), „User training: Less is more", *Datamation* (jul): 175-182.

Scharer, Laura (1990), „Pinpointing requirements", u *System and Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press.

Scholtes, Peter R. (1995), *The Team Handbook*. Joiner Associates.

Opisuje organizaciju i uticaj timova uključujući i uputstva za postupanje sa nezgodnim članovima tima.

Schum, David A. (1994), *Evidential Foundations of Probabilistic Reasoning*, Wiley Series in Systems Engineering. New York: John Wiley.

Primenjuje pravničko razmišljanje na postupak sa dokazima. Uključuje Bayesovu verovatnoću u procenjivanju da li je dokazni materijal dovoljan.

Schwaber, Ken i Mike Beedle (2002), *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall.

Seddon, John (1996), ISO 9000 Implementation and Value-Added: Three Case Studies, tehnički izveštaj. London: Vanguard Consulting.

Seligman, Dan (1997), „Midsummer madness: New technology is marvelous except when it isn't", *Forbes*, septembar 8, str. 234.

Opisuje probleme sa fakturisanjem nastale zbog problema sa Nortelovim softverom.

Shaw, Mary (1990), „Prospects for an engineering discipline of software", *IEEE Software, 7(6)* (novembar), 15-24.

Shaw, Mary i David Garlan (1996), *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall.

Odlična knjiga u kojoj se opisuju neka od ključnih pitanja u procenjivanju dizajna softvera.

Shepperd, Martin (1997), „Effort and size estimation: An appraisal", *Software Reliability and Metrics Club Newsletter*, januar: 6-8. London: Centre for Software Reliability.

Shepperd, Martin, Chris Schofield i Barbara A. Kitchenham (1996), „Effort estimation using analogy", u *Proceedings of the Eighteenth International Conference on Software Engineering* (Berlin), Los Alamitos, CA: IEEE Computer Society Press.

Shneiderman, Ben (1997), *Designing the User Interface: Strategies for Effective Human-Computer Interface*, 3. izdanje. Reading, MA: Addison-Wesley.

Shooman, M. L. (1983), *Software Engineering*. New York: McGraw-Hill.

Shooman, M. L. i M. Bolsky (1975), „Types, distribution and test and correction times for programming errors", u *Proceedings of the 1975 International Conference on Reliable Software*. New York: IEEE Computer Society Press.

Shull, F., I. Rus i V. Basili (2000), „How perspective-based reading can improve requirements inspections", *IEEE Computer*, 33(7) (jul): 73-79.

Shumate, K. i M. Keller (1992), *Software Specification and Design: A Disciplined Approach for Real-Time Systems*. New York: John Wiley.

Simmons, Pamela L. (1996), „Quality outcomes: Determining business value", *IEEE Software*, 13(1) (januar): 25-32.

Simon, H. A. (1981), *The Sciences of the Artificial*. Cambridge, MA: The MIT Press.

Skowronski, Victor (2004), „Do agile methods marginalize problem solvers?" *IEEE Computer*, 37(10) (oktobar): 120-118.

Smith, Bill (1993), „Six sigma design", *IEEE Spectrum* 30(9) (septembar): 43-46.

Dobar članak koji opisuje kako se u proizvodnji koristi prvoklasni dizajn.

Smith, M. D. i D. J. Robson (1992), „A framework for testing object-oriented programs", *Journal of Object-Oriented Programming*, 5(3) (jun): 45-54.

Software Engineering Coordinating Committee of the ACM and IEEE (2004), „Guide to the Software Engineering Body of Knowledge", http://www.swebok.org (oktobar).

Spivey, J. M. (1992), *The Z Notation: A Reference Manual*, 2. izdanje Englewood Cliffs, NJ: Prentice Hall.

Opisuje jedan formalni jezik za specifikaciju zahteva.

Sreemani, Tirumale i Joanne M. Atlee (1996), „Feasibility of model checking software requirements: A case study", u Proceedings of the 11th Annual Conference on Computer Assurance 77-88.

Srinivasan, K. i D. Fisher (1995), „Machine learning approaches to estimating development effort", *IEEE Transactions on Software Engineering*, 21(2): 126-137.

Standish Group (1994), The CHAOS Report. Dennis, MA: The Standish Group.

--- (1995), *The Scope of Software Development Project Failures*. Dennis, MA: Standish Group.

Steele, Claude (1999) „Thin ice: Stereotype threat and black college students", *Atlantic Monthly* (avgust).

Steinberg, Daniel H. i Daniel W. Palmer (2004), *Extreme Software Engineering: A Hands-on Approach*. Upper Saddle River, NJ: Prentice Hall.

Stephens, Matt i Douglas Rosenberg (2003), *Extreme Programming Refactored: The Case Against XP*. Berkeley, CA: Apress.

Swanson, Mary i S. Curry (1987), „Results of an asset engineering program: Predicting the impact of software reuse", u *Proceedings of the National Conference on Software Reusability and Portability*. Los Alamitos, CA: IEEE Computer Society Press.

Opisuje korišćenje finansijskih podsticaja za višekratnu upotrebljivost u GTE Data Services u Tampi, Florida.

Swartout, W. R. i R. Balzer (1982), „On the inevitable intertwining of specification and implementation", *Communications of the ACM*, 25(7) (jul): 438-439.

Teasley, B., L. Leventhal, B. Blumenthal, K. Instone i D. Stone (1994), „Cultural diversity in user interface design: Are intuitions enough?" *SIGCHI Bulletin*, 26(1) (januar): 36-40.

Teichroew, D. i E. A. Hershey III (1977), „PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems", *IEEE Transactions on Software Engineering*, SE 3(1) (januar): 41-48.

Uvodi jedan jezik za evidentiranje zahteva.

Theofanos, Mary F. i Shari Lawrence Pfleeger (1996), „Wavefront: A goal-driven requirements process model", *Information and Software Technology*, 38(1) (januar): 507-519.

Prikazuje model za prikupljanje zahteva zasnovan na merenjima i ciljevima.

Trager, Louis (1997), „Net users overcharged in glitch", *Inter@ctive Week*, 8. septembar.

Opisuje probleme sa neadekvatnim testiranjem softvera u Nortelu.

Travassos, Guilherme H. i R. S. Andrade (1999), „Combining metrics, principles and guidelines for object-oriented design", u *Proceedings of the Workshop on Quantitative Approaches to Object-Oriented Software Engineering* (ECOOP99), Lisbon, Portugal.

Može se naći na: http://www.esw.inesc.pt/ftp/pub/esw/mood/ecoop99/.

Tsuda, M., et al. (1992), „Productivity analysis of software development with an integrated CASE tool", u *Proceedings of the International Conference on Software Engineering*. Los Alamitos, CA: IEEE Computer Society Press.

Opisuje program višekratnog korišćenja u Hitachiju.

Turoff, M i S. R. Hiltz (1995), „Computer-based Delphi processes," u M. Adler i E. Ziglio, urednici: *Gazing Into the Oracle: The Delphi Method and Its Application to Social Policy and Public Health*. London: Kingsley Publishers.

Tversky, Amos i Daniel Kahneman (1981), „The framing of decisions and the psychology of choice", *Science*, 211: 453-458.

University of Southern California (1996), *COCOMO II Model User's Manual*, Version 1.1. Los Angeles: USC.

U.S. Department of Defense (1994), *Military Standard: Software Development and Documentation*, MilStd-498. Washington, DC: DOD.

Ovo je trenutni standard za razvoj softvera koji važi za sisteme koji se grade u Ministarstvu odbrane SAD ili za njega.

Valacich, J. S., L. M. Jessup, A. R. Dennis i J. F. Nunamaker, Jr. (1992), „A conceptual framework of anonymity in group support systems", u *Proceedings of the 25th Annual Hawaii Conference on System Sciences*, tom III: *Group Decision Support Systems Track*, str. 113-125. Los Alamitos, CA: IEEE Computer Society Press.

Vartabedian, Ralph (1996), „IRS computer project has 'very serious problems,' Rubin Says", *Los Angeles Times*, 29. mart, str. D1.

Verner, June i Graham Tate (1988), „Estimating size and effort in fourth generation development", *IEEE Software*, 5(4) (jul).

Vinter, Otto (1996), „The prevention of errors through experience-driven test efforts," Delta Report D-259 (januar), Kopenhagen.

Opisuje retrospektivnu tehniku za procenu suprotstavljenih metoda testiranja.

Voas, J. M. i Michael Friedman (1995), *Software Assessment: Reliability, Safety and Testability*. New York: John Wiley.

Voas, J. M. i K. W. Miller (1995), „Software testability: The new verification", *IEEE Software*, 12(3) (maj): 17-28.

Walden, Kim i Jean-Marc Nerson (1995), *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*. Englewood Cliffs, NJ: Prentice Hall.

Walston, C. i C. Felix (1977), „A method of programming measurement and estimation", *IBM Systems Journal*, 16(1): 54-73.

Walz, Diane B., Joyce J. Elam, Herb Krasner i Bill Curtis (1987), „A methodology for studying software design teams: An investigation of conflict behaviors in the requirements definition phase", u *Empirical Studies of Programmers*: Second Workshop, str. 83–99. New York: Ablex Publishing.

Predstavlja jedan metod za analitičke postupke u projektovanju velikih sistema s računarskom podrškom, zasnovan na karakterizaciji procesa projektovanja tako da se priznaju različiti koncepti članova tima da bi se naglasak stavio na transformaciju apstraktnih ciljeva u konkretne sisteme i povukla razlika između neuspeha u procesu projektovanja koji su posledica funkcije dizajna i onih koji potiču od samog grupnog procesa (u kontekstu dizajna).

Ward, P. T. i S. J. Mellor (1986), *Structured Development for Real-Time Systems*, 3 toma New York: Yourdon Press.

Warmer, Jos i Anneke Kleppe (1999), *The Object Constraint Language: Precise Modeling with UMI*. Reading, MA: Addison-Wesley.

Wasserman, Anthony I. (1990), „Tool integration in software engineering environments", u F. Long (urednik), *Software Engineering Environments*, str. 138–150. Berlin: Springer-Verlag.

--- (1995), „Towards a discipline of software engineering: methods, tools and the software development process," uvodno Stevensovo predavanje o metodima razvoja softvera. U *Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering* (Toronto). Los Alamitos, CA: IEEE Computer Society Press.

Tekst Wassermanovog predavanja o nekoliko ključnih pitanja u današnjem softverskom inženjerstvu.

--- (1996), „Toward a discipline of software engineering", *IEEE Software*, 13(6) (novembar): 23–31.

Nadovezuje se na Wassermanovo i Stevensovo predavanje i objašnjava zašto se današnji razvoj razlikuje od onoga pre 10 ili 20 godina.

Watson, R. T., T. H. Ho i K. S. Raman (1994), „Culture: A fourth dimension of group support systems", *Communications of the ACM*, 37(10) (oktobar): 44–55.

Weiderhold, Gio (1988), *Database Design*, 3. izdanje. New York: McGraw-Hill.

Weinberg, Gerald M. (1971), *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

Seminarski rad o programerskom načinu razmišljanja i kako organizacije potpomažu ili ometaju kreativnost i produktivnost.

--- (1993), *Quality Software Management: First Order Measurement*. New York: Dorset House.

Deo ove knjige odnosi se na stilove rada i izgradnju timova.

Weller, E. F. (1992), „Lessons learned from two years of inspection data", u *Proceedings of the Third International Conference on Applications of Software Measurement*, str. 2.57–2.69.

--- (1993), „Lessons from three years of inspection data", *IEEE Software*, 10(5) (septembar): 38–45.

Izvrsna empirijska studija sa prikazom podataka iz preko 6 000 inspekcijskih sastanaka. Na primer, poredi se učestanost otkrivanja grešaka za različite veličine timova i za različite vrste koda.

--- (1994), „Using metrics to manage software projects", *IEEE Computer*, 27(9) (septembar): 27–34.

РЕПУБЛИКА СРПСКА
Природно-математички
факултет Бања Лука
Библиотека

Wetherbe, J. C. (1984), *Systems Analysis and Design: Traditional, Structured i Advanced Concepts and Techniques*. Igan, MN: West Publishing.

Whittaker, James A. i Steven Atkin (2002), „Software engineering is not enough", *IEEE Software* 19(4) (jul/avgust): 108-115.

Wilde, Norman, Paul Matthews i Ross Huitt (1993), „Maintaining object-oriented software", *IEEE Software*, 10(1) (januar): 75-80.

Williams, L., R. Kessler, W. Cunningham i R. Jeffries (2000), „Strengthening the case for pair programming", *IEEE Software* 17(4) (jul/avgust): 19-25.

Wilson, Peter B. (1995), „Testable requirements: An alternative software sizing measure", *Journal of the Quality Assurance Institute* (oktobar): 3-11.

Wing, Jeannette M. (1990), „A specifier's introduction to formal methods", *IEEE Computer*, 23(9) (septembar): 8-24.

Dobar, jasan uvod u formalne metode sa ukazivanjem na mnoge druge resurse.

Winograd, T. i F. Flores (1986), *Understanding Computers and Cognition*. Norwood, NJ: Ablex.

Withrow, Carol (1990), „Error density and size in Ada software", *IEEE Software*, 7(1) (januar): 26-30.

Pronalazi krivu gustine grešaka oblika U pri poređenju veličina i grešaka u Unisysovom softveru Ada.

Wittig, G. E. i G. R. Finnie (1994), „Using artificial neural networks and function points to estimate 4GL software development effort", *Australian Journal of Information Systems* 1(2): 87-94.

Wolverton, R. W. (1974), „The cost of developing large-scale software", *IEEE Transactions on Computers*, C23(6): 615-636.

Opisuje jedan od prvih modela troškova gde se koristila matrica troškova u kombinaciji sa težinom projekta.

Wood, Jane i Denise Silver (1995), *Joint Application Development*, 2. izdanje, New York: John Wiley.

Yourdon, Edward (1982), *Managing the System Life Cycle*. New York: Yourdon Press.

--- (1990), *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice Hall.

--- (1994), „Developing software overseas", *Byte*, 19(6) (jun): 113-120.

Yourdon, Edward i Larry Constantine (1978), *Structured Design*. Englewood Cliff, NJ: Prentice Hall.

Uvođenje pojmova sprege i kohezije za procenjivanje kvaliteta dizajna.

Zave, Pamela (1984), „The operational versus the conventional approach to software development", Communications of the ACM, 27(2) (februar): 104-118.

Zave, Pamela i Michael Jackson (1997), „Four dark corners of requirements engineering", *ACM Transactions on Software Engineering and Methodology*, 6(1) (januar): 1-30.

Operativni opis prema konvencionalnom pristupu razvoju softvera.

Zelkowitz, Marvin V., Dolores R. Wallace i David Binkley (1998), „Understanding the culture clash in software engineering technology transfer," tehnički izveštaj sa Univerziteta u Merilendu (2. jun).

Indeks

A

adaptivni razvoj softvera (ASD), 60
adaptivno održavanje, 508
agilne metode, 59, 144
agilnost
  četiri karakteristike, 60
agregacija, 161
agregacija, 304-305
aktivnost, 16, 64, 83
alati za održavanje (*videti takođe* podmlađivanje), 524-532

alati za testiranje, 403-405, 450-453, 474
alati, 4, 35, 40, 631
  automatizovano kodiranje, 340, 363
  automatizovano testiranje sistema, 357
  definicija, 4
  kriterijumi za izbor, 474
  modeliranje procesa, 63, 79
  ocenjivanje, 138
  za testiranje, 403-405
alfa test, 448
algebarska specifikacija, 184-188
algoritam prostiranja unazad, 115
algoritmi, 342, 355, 363, 367, 375, 382
analitičar sistema, 143
analitičar zahteva, 25, 143, 147
analitičari, 429-430
analiza domena, 566
analiza osnovnog uzroka (*videti takođe* grafovi uzrok posledica), 457
analiza post mortem, 575-580, 596-597
analiza sistema, *videti* analiza
analiza stabla grešaka, 264
analiza svojstava, 545, 640
analiza uticaja, 526-630
analiza, 2, 39, 44
analizator koda, 340
analizator podataka, 402
aplikativne tačke, 111, 112, 113
apstrakcija, 30, 40, 64, 157, 282, 630
  stepen, 172, 207
apstraktna klasa, 161, 290
apstraktni podaci, 270
arhitektonski stilovi, 229
arhitektura baze podataka, 233
arhitektura klijent/server, 236, 237
arhitektura specifična za domen, 236
arhitektura, 32, 282, 631
  definicija, 229
Ariane-5, 37, 42, 44, 75, 135, 211, 281, 334, 361, 411, 475, 496, 539, 596, 626
  odgovornost medija, 44
  poslovni uticaj, 38
  problemi sa zahtevima, 411
  simulacija, 540
  upravljanje rizikom, 135, 539-540
asocijacija, 161, 304-305
asocijativna klasa, 162
atribut u opsegu klase, 160
automatizovana okruženja za testiranje, 404, 411, 450-452, 474, 530-531
automatizovano dokazivanje teorema, 384

B

Bailey-Basilijev model procene rada, 110, 118
bela kutija, 374, 376, 469, 566

Belady-Lehmanov model održavanja, 516
beta test, 448, 449
bezbednosni slučajevi, 466
bezbednosno kritični sistemi, 7, 175, 462-265, 553
bezbednost, 43, 147, 150, 175, 256, 257, 466, 467
bezlični razvoj projekta, 102, 373
bibliotekar, 27
blackboard arhitektura, 233
Boehmov model kvaliteta, 559-561
brza izrada prototipa, 193
brzi referentni vodič, 494
bubice, 6, 366
buka, 601, 602, 604-606, 621

C

CASE
  NIST/ECMA referentni model, 228
CASE alati (*videti takođe* alati), 35, 104, 201, 207, 228, 113-114
cevi i filtri, dizajn, 230, 272
ciklomatski broj, 400, 557, 520-522
cilj, 563
CMM, 12, 581-587, 591, 613-616, 654
COCOMO II model za procenu, 111-115, 133-134, 138, 517-518

COCOMO model za procenu, 111, 116, 118, 591, 622
COTS (kataloški proizvod), 15, 44
crna kutija, 374, 376, 469, 565
Crystal, 59

Č

činjenje
  greška činjenja, 369
čista soba, 469-473, 618
  definicija, 469
  procenjivanje, 470-471, 618-621
čovek, meseci rada po, 133

D

datoteke razlika i kontrola konfiguracije, 425-427
dejstva, 317
delotvornost testa, 389, 456
deterministički sistem za predviđanje, 554
difuzija inovacije, *pogledajte* prenos tehnologije
dijagnostičke transakcije, 258
dijagram aktivnosti, 312
dijagram entiteti-odnosi ili ER dijagram, 158
dijagram interakcije, 310
dijagram klasa, 188, 159-162
dijagram komponenti, 300
dijagram prelaza stanja, 165-169, 188
  akcije i aktivnosti, 168
  stanje, 169
dijagram raspoređivanja, 300
dijagram saradnje, 300, 310
dijagram sekvenci, 188, 300, 310
dijagram slučaja upotrebe, 188, 209, 173-174
dijagram stanja, 311
dijagram toka podataka, 172-173
  proces, 172
  skladište podataka, 172
  tok podataka, 172
  učesnik, 172
dijagrami prelaza stanja, 66
dijagrami toka podataka, 355, 533
dinamička analiza koda, 402, 403
dinamički model, 64, 67
dinamički modeli sistema, 67, 623
dinamički prikaz klasa, 292
dinamičko vezivanje, 316
direktno dejstvo i povratna sprega (u kontroli procesa), 235
dizajn (*videti takođe* dizajn korisničkog interfejsa, dizajn rukovanja podacima, prototipovi, dizajn rukovanja poslovima)
  definicija, 223
  dizajn programa, 315
  karakteristike dobrog dizajna, 248, 340
  komplet alatki, 317
  konceptualni i tehnički, 224
  kulturološka pitanja, 242
  mere za OO dizajn, 326
  merenje kvaliteta, 268
  pitanja u analizi karakteristika, 245-246, 264
  saradnja tokom dizajna, 240
  sistema, 223
  usredsređen na zadatak ili na korisnika, 283
dizajn i ponovni dizajn, 259, 349
dizajn koda
  definicija, 229
dizajn po ugovoru, 261, 282, 361
dizajn programa, 315
dizajn sistema, 224, 300
dizajn spolja ka unutra, 32, 227
dizajn upravljanja podacima, 319
dizajn upravljanja zadacima, 320-323
dizajn usredsređen na korisnika, 283
dizajn usredsređen na zadatak, 283
dizajn zajedničkih podataka, 270
dizajn, kriterijum, 149
dizajn, metodi i notacije, 31
dizajn, prostor (*videti takođe* prostor rešenja), 149
dizajni, poređenje, 245, 270
dobro definisan, 401
dodeljivanje prioriteta zahtevima, 153
dokaz
  izvori, 636
  kredibilnost, 636
  publika za, 638
  u donošenju odluke o prenosu tehnologije, 634
dokazivanje ispravnosti koda, 380-384, 470
dokazivanje teorema, 203
dokumentacija, 150, 193-198, 276, 338, 351-355, 367, 452-462, 488-495, 533
domensko znanje, 196
domet kontrole i efekta, 254
donošenje odluka, 632, 641
  grupe, 644, 647
  kontekst problema, 642
  legitimacija, 642
  model odluke na osnovu prepoznavanja, 645
  polarizacija konteksta, 646
  polarizacija status quo, 647
  pronalaženje problema, 642
  rešavanje problema, 642
  stereotipna pretnja, 647
  strategije za, 644
  vrste odluka, 641
doslednost, 560
dostupnost, 441, 438-439

Dromeyev model kvaliteta, 562-563
Duaneov model pouzdanosti softvera, 601
dubina nasleđivanja, 326
dvosmisleni, 155
dvostruko stohastički model, 446

E

editori teksta, 530
efekti, 432
efikasnost testa, 456
eksperiment, *videti* formalni eksperiment
eksperimenti u čistoj sobi, 471, 619
eksperti iz oblasti primene, 147
ekspertski sud, 106
ekstremno programiranje (XP), 59
  karakteristike, 60-62
ekstrovertni stil rada, 99
empirijska procena, *pogledajte* procena
enkapsuliranje, 286, 290
entitet, 17
E-sistem, 516, 539, 502-503
etički kôd, 667
evolucija softvera, 504, 505, 506
evoluciona faza razvoja, 504
evolutivni prototip, 192

F

facetovana klasifikacija, 568
faktori ponašanja, 63
fazni razvoj, 55
filozofija testa, 375
FMEA (analiza režima otkaza i posledica), 466
fog indeks, 523
formalna inspekcija, 200
formalni eksperimenti, 546, 547, 640
formalni metodi, 213, 219, 175-187
formalni specifikacioni jezici, 175, 181, 184
funkcija gustine verovatnoće, 442-443
funkcija pouzdanosti, 443
funkcija raspodele, 443
funkcionalne notacije, 175
funkcionalne tačke, 111, 138
funkcionalni test, 371, 420, 430-436
funkcionalni zahtev, 148
funkcionalnost, 111

G

gantogram, 93, 124
Garvin
  gledišta na kvalitet, 9
generalizacija, 161, 304-305
generativna višekratna upotrebljivost, 566
generatori slučajeva za testiranje, 405
generatori ukrštenih referenci, 531
glavno tržište, 635
Goel-Okumotov model ponovne upotrebe softvera, 601, 607
graf aktivnosti, 84, 86, 139
grafovi uzroka i posledica, 432-435
granice (*videti takođe* granice sistema), 16
granice sistema, 16, 17, 40, 44, 124, 154, 173, 225, 297
  sistema Piccadilly, 37
greškau koordinaciji, 368
greška u oporavku, 368
greška u performansama, 368
greška u preciznosti, 367
greška u propusnosti, 368
greška u sistemskom softveru, 368
greška u rokovima, 368
greška, 44
  definicija, 6
  sejanje, 406
greške u kapacitetu, 368
greške u ograničenjima, 368
greške u preopterećenju, 368
greške u standardima i proceduri, 368
greške u zahtevima, 201, 411
greške, 6, 44, 256, 365, 369
  izvori, 240, 365-366, 417-419
  razlike između hardverskih i softverskih, 366, 499
  u zahtevima, 201, 411
  uspešno pronalaženje, 365, 389-390
  vrste, 367, 389
grupno donošenje odluke, 644
Gunningov fog indeks, 523
gustina grešaka, 474, 522, 620

H

haker, 5
hardverski otkazi, 368, 499
HAZOP (studije rizika i upotrebljivosti), 467
HAZOP karakteristične reči, 468
hendl, 288
hijerarhija klasa, 325, 355
hipoteza, 546, 548
horizontalna pogodnost za praćenje, 526
horizontalna ponovna upotreba, 567
hvatanje i ponavljanje ili reprodukcija, 404

I

identitet, 286, 288
IEEE standard, 196
ime, 288
implicitno pozivanje, 231, 341, 270-271

indeks specijalizacije, 324
inkrementalni razvoj, 21, 56, 57, 76, 421
inovatori, 634
inspekcija koda, 377, 378
inspekcije, 8, 124, 377, 609-612
  greške pronađene tokom, 389, 390, 378-379, 609, 611-612
  veličina timova, 379
  vreme za sastanke i pripreme, 339, 354, 351-352
instalacioni test, 372, 421, 450
integracija, 35
integraciono testiranje, 371, 411, 473, 390-396
  poređenje strategija, 394-396
integrisana okruženja, 631
integrisani razvojni tim, 128
integrisanje po principu „velikog praska", 393
integrisanje po principu odozdo nagore, 390
integrisanje po principu sendviča, 394
integritet, 560
interfejsi (*videti takođe* korisnički interfejs), 315, 340
  unutrašnja dokumentacija, 351
interna validnost, 557
interpreteri, 234
introvertni stil rada, 99
intuitivni stil rada, 99
inženjeri za testiranje, 25
inženjerski pristup, 21
isečak stabla, 266
ISO 9000, 12, 616, 589-590
ispravljanje i uklanjanje grešaka, 258, 366
istraživač tržišta, 147
iterativni razvoj, 56, 57, 76
izbor specifikacione tehnike, 206
izdanja, 56, 57, 423
izgradnja tima, 138, 622
izrada prototipa korisničkog interfejsa, 31, 630
izveštaj o analizi testa, 452, 459
izvođenje zahteva, 145, 147
izvršne specifikacije, 207
izvršni dizajn
  definicija, 229

J

javni i privatni, 309
javni interfejs, 309
jedinično testiranje, 371, 376, 389-473
Jelinski-Moranda model pouzdanosti softvera, 445, 601, 602-605
jezici
  uticaj na kvalitet, 344-346, 363

K

karakteristične reči, 468
kasna većina, 634-635
kataloški proizvod, *videti* COTS
klasa, 159, 288
klasifikacija (OO sistema), 286, 288
klasifikacija grešaka, 369
klasifikacija komponenata, 567
klasifikaciona stabla, 410
klijent, 146, 236, 237, 262
klizanje aktivnosti, 88
ključne oblasti procesa CMM, 582
ključni postupci CMM, 586
ključne reči, 265
kôd sklon greškama
  prepoznavanje, 409, 408-410
kohezija, 248, 251, 328, 340, 342
Kolmogorovljevo rastojanje, 603, 608
komentari u programima, 352
komparatori datoteka, 530
komplet alatki, 317
komponente, 228
kompozicija (kod OO sistema), 304-305
kompozicija, 161
kompoziciona višekratna upotrebljivost, 566
komunikacije, međuljudske, 63, 240, 366, 622
komunikacioni dijagram, 188
komunikacioni kanali, 68, 97, 101, 136
konačni automat, 164
  prelaz stanja, 164
  stanje, 164
konceptualna bliskost, 568
konceptualni dizajn, 223
konceptualni model, 157
konfiguracija sistema, 423
konkretan subjekt i posmatrač, 322
konkurentnost, 244
kontekst dijagram, 37
kontekst, 317
kontrola procesa, 235
kontrola promena, 428, 524-526
kontrola verzije i izdanja, 423
kontrola, 64, 235, 254, 343, 469
kontrolne strukture, 341
korektivno održavanje, 507
korigovanje podataka, 555
korisnici, 146, 150, 195, 430
korisnički interfejsi, 29, 32, 282
  dizajn, 242, 318-319
  kulturološka pitanja, 242
  preferencije korisnika, 243
korisničko uputstvo, 489-491

korisnik
definicija, 15, 484
korišćenje, 294
krajnja tačka, 84
krajnji rok, 84
kriterijum usklađenosti, 151, 153
kritička revizija dizajna, 274, 275
kritična putanja, 90
kupac, 26, 146
kupac, definicija, 15
kutija stanja, 376, 469
kvalitet
Garvinova gledišta, 9
kontekst poslovnog okruženja, 12, 40
proces, 11
proizvod, 10
tehnički, 40
kvalitet dizajna, 268, 516
kvalitet softvera, *videti* kvalitet
kvalitet, zahtevi, 149-150, 152

L

Laieva notacija, 64, 74, 80
latentno vreme, 88
lažne rutine, 392, 404
Lehmanovi zakoni evolucije softvera, 505
letimični pregled koda, 377
Li-Henry mere, 329, 336
linija života, 310
lista za proveru, 199
Littlewoodovi modeli pouzdanosti softvera, 446, 601, 602-605, 607
ljudski faktor (*videti takođe* korisnički interfejs), 621-622
„Aranžer pozajmica" (seminarski rad), 41, 77, 137, 215, 283, 335, 363, 413, 497, 541, 627, 671, 597-598
logika, 178-180
lokalizovanje ulaza i izlaza, 346
Lorenz-Kidd mere, 324-326, 331, 336

M

Marvel, 70
mašinsko učenje, 116-117
matrična organizacija, 128
McCallovi modeli kvaliteta, 11, 559
međusobne sumnje, 256
međusobno isključivanje, 246
mera
definicija, 553
vrednovanje, 554, 557
mere Chidambera i Kemerera, 329, 326-329, 336
merenje softvera, *videti* merenje
merenje veličine, 111, 324
merenje zahteva, 204
merenje, 34, 40, 553, 631
Chidamber-Kemerer, 326-329, 336,
efikasnost i delotvornost testa, 456
kvalitet dizajna, 268
Li-Henry, 329, 336
Lorenz-Kidd, 324-326, 331, 336
mere za OO dizajn, 326
OO mere veličine, 323
poboljšanja procesa, 574, 581-590
pouzdanost, dostupnost i održljivost, 442
za održavanje, 505, 518-522, 541, 617-618
za višekratnu upotrebljivost, 569
zahteva, 204
metod kritične putanje (CPM), 87, 91, 92
metod, 159
metode i notacije analize i dizajna, 31, 630
metodi, 4, 290
miljokazi, 83, 84, 88, 131
MMRE, 117-118
model kvaliteta
Boehm, 559-561
Dromey, 462-563
ISO 9126, 561-562
McCall, 11, 559-561
model kvaliteta ISO 9126, 561-562
model referisanja, 128
model vodopada, 28, 48, 73
nedostaci, 50
sa izradom prototipa, 52
model odlučivanja zasnovan na prepoznavanju, 645
model zrelosti ljudi, 591-593
model zrelosti sposobnosti, *pogledajte* CMM
model, proveravanje, 203
modeli pouzdanosti
poređenje, 555-556
modeli zrelosti (*videti takođe* CMM), 581-590
modeli zrelosti procesa (*videti takođe* CMM, ISO 9000, SPICE), 581, 613-616
merenje i procena procesa, 77, 574
modeli, (*videti takođe* procena, održavanje, kvalitet, pouzdanost, simulacija)
napora za održavanje, 516, 516-518
ponašanja grešaka, 474, 522, 555-556
modularni
definicija, 228
modularno raščlanjivanje, 32, 227
modularnost, 157, 207, 228, 238, 341
mogućnost ponovnog korišćenja softvera, *videti* višekratna upotrebljivost

mogućnost praćenja, 155, 197, 199, 202, 340, 526, (*videti takođe* analiza posledica)
monitor, 246, 249, 403
Musa model pouzdanosti softvera, 443, 446
Musa podaci, 443, 602, 606
Musa-Okumotov model pouzdanosti softvera, 607

N

nasleđeni sistemi, 504
nasleđivanje, 159, 286, 290, 326
nasumično testiranje, 388
nauka o odlučivanju, 642
nečinjenje
  greška nečinjenja, 369
nefunkcionalni zahtevi, 149-150
neizvesnost tipa 1 i tipa 2, 440, 444-445
neizvesnost u predviđanju otkaza, 440-444
neto sadašnja vrednost, 594-595
neuronske mreže, 116
nezavisna provera valjanosti, 482
nezavisnost komponenti, 248, 346, 355, 391
NIST/ECMA referentni model, 228
nivo specifikacije, 194
niz (aktivnosti), 64

O

objekat, 17
objektne tačke, 112
objektni model, 300
objektno orijentisana tehnologija, 29
objektno orijentisani sistemi
  definicija, 286
  dizajn, 32, 227, 231, 293, 300, 315, 341, 360
  klasifikacija, 288
  mere veličine, 323
  mere, 323-332
  načini za predavanje argumenata, 360
  notacije zahteva, 158, 213
  održavanje, 512
  pravljenje i testiranje, 293, 397
  specifikacija, 70, 293
  testiranje, 397-398
obrada izuzetaka, 253, 254, 281, 361, 539-540
obrasci za izveštavanje o greškama, 460
obrasci za izveštavanje o odstupanju, 460
obrasci za izveštavanje o problemima, 460-462
obrazloženje dizajna, 276, 355
obučavanje korisnika, 484
obuka operatera, 484, 485
obuka, 57, 125, 483-488, 495
OCL (Object Constraint Language), 188, 180-181
odbor za upravljanje konfiguracijom, 524
održavanje softvera, *videti* održavanje
održavanje, 340, 499
  adaptivno, 508
  korektivno, 507
  merenje karakteristika, 505, 518-522, 541, 617-618
  objektno orijentisani sistemi i, 512
  usavršavajuće, 508
održljivost, 150, 151, 207, 441, 541, 438-439, 518-522
ograničenja, 149-150, 153, 178-182, 195
oklevala, 634-635
okruženja, 35, 631
okruženje, 150, 154-155, 196, 202-203
okvirno rešenje, 317
onlajn pomoć, 486, 494
operacija u opsegu klase, 161
operativna notacija 178
operativni sistem, 56
operativni profil za modeliranje pouzdanosti, 447
opis testa, 452, 456
opisna notacija, 178
oportuna cena, 594
opseg prihvatljivosti, 555
opšti vodič kroz sistem, 492
organizacija, 64
ortogonalna klasifikacija grešaka, 369
osnova, 530, 546, 563
osnovni skup znanja za softversko inženjerstvo, 657
osobina bezbednosti, 220
osoblje, 95
otkaz, 6, 44, 256, 264
otkrivanje grešaka, 256, 370, 389
otpornost na greške, 256, 258, 539
ozbiljnost greške, 74, 439

P

paketi (za OO)
  dijagrami, 300, 309
  zavisnost, 309
paradigma
  definicija, 4
paralelno testiranje, 448
Parnasove tabele, 177
PERT, tehnika evaluacije i pregleda programa, 92
perzistencija, 286, 291
Petrijeve mreže, 169-172
Piccadilly, primer televizijskog sistema, 36, 43, 73-75, 133, 209, 279, 332, 359, 411, 474, 495, 538, 595, 625
pilot test, 448
plan bezbednosti, 123

plan integracije sistema, 401
plan integracije, 401, 421
plan izgradnje, 421, 429, 479
plan održavanja, 123
plan osiguranja kvaliteta, 123
plan testiranja, 124, 400-402, 452-454
planiranje testa, 125, 400-402
planiranje, (*videti* rukovanje projektom, konfiguracija, plan testiranja)
poboljšanje procesa, 72, 613-621, 654
  smernice, 624-625
podaci o greškama, 439
podizvođači, 16
podmlađivanje (*videti takođe* reinženjersvo, restrukturiranje, redokumentaciju, reverzni inženjering), 532
  relacione notacije, 175
pokrivenost testom, 412
polarizacija konteksta, 646
polarizacija status quo, 647
polarizacija, 646, 647, 602-604
polimorfizam, 286, 290
politika, 64
ponašanje, 290
ponovna upotreba, 34, 341, 349-350, 408, 564-574, 612-613
  analiza domena za, 566
  definicija, 565
  generativna prema kompozicionalnoj, 566
  iskustva, 350, 568-570, 612-613
  mere, 362, 569
  model procesa za softver rakete, 75
  ponovna upotreba crne kutije prema ponovnoj upotrebi bele kutije, 565
  poređenje potrošača i proizvođača, 349, 565
  troškovi i koristi, 570-574, 612-613
  uticaj na kvalitet, 408, 612
  uzorci dizajna, 247
  vertikalna prema horizontalnoj, 567
  zahteva, 149
poruke o otkazima, 493
poslovna vrednost, 12, 129, 615, 622
posmatrač, 322
postupak, 4, 46
postupna rafinacija, 376
potpunost testiranja, 387
pouzdanost, 438-447, 559
  razlike između hardverske i softverske, 442
poverenje u testiranje, 407, 619
povraćaj ulaganja, 13, 594-596
poželjne karakteristike, 155
pravljenje prototipa, 31, 51, 73, 191-193, 389
  brzo, 193
  dizajn, 263
  evolutivno, 192
  korisnički interfejs, 31, 630
  model, 53
  za bacanje, 192
PRED(x/100), 117-118
predavač, 26
predavanje argumenata, 360
predmet testiranja, 374
predskazivač testa, 482
predviđanje
  šum, 601, 602, 604-606
  modeli i procedure, 443-444, 601-609
  poboljšanje, 578, 601-609
  polarizacija, 602-604
  pouzdanost (*videti takođe* predviđanje pouzdanosti), 443
  rekalibracija, 609
  vrednovanje, 554
predviđanje pouzdanosti, 443, 555
pregledanje koda, 376, 377, 389
pregledi dizajna, 274, 389, 390
pregledi zahteva, 200, 411
pregledi, 8, 546, 577
  dizajna, 274, 276, 389, 390
  koda, 376, 377, 389
  zahteva, 200, 389, 411
pregovaranje, 153
prekvencijalna verovatnoća, 604
preliminarna revizija dizajna, 274
prenos tehnologije, 632-641
  brzina usvajanja, 633, 637
  dokaz za podršku, 634
  modeli, 639
  usvajanje stilova, 635
prenosivost, 559-562
prepoznavanje grešaka, 366
prethodnik, aktivnost, 84
pretpostavke, 196, 202
preuzimanje softvera od Ariane-4, 75
preventivno održavanje, 508
prevodioci i linkeri, 343, 361, 530
primerak, 290
priručnik za operatere, 491-493
privatni interfejs, 309
problemi održavanja
  uzroci, 510-518
procena
  gotovosti projekta, 87
  metod kritične putanje (*videti takođe* CPM), 87
  pomoću analogije, 107
  pomoću ekspertskog suda, 106

rada na održavanju, 514–516
rada na projektu, 104
tačnosti, 104, 106
uzroci netačnosti, 105
procena rada, 104
algoritamske metode, 108
Bailey i Basili, tehnika modeliranja, 110
COCOMO model (*videti takode* COCOMO, COCOMO II), 111
procena tačnosti modela, 117
upotreba mašinskog učenja, 115
Walston i Felix, model, 109
Wolvertonov model, 107
procena prema predvidanju, 553
procena veličine
objektne tačke, 111
procena, 544–549
izbor tehnika za, 549
klopke u, 550–551
pripremanje za, 548
procesa, 574
proizvoda, 559
resursa, 590
vrste, 545–547
procenat popusta, 594
proces
definicija, 45
razlozi za modeliranje, 48, 631
proces razvoja softvera, 25
proces razvoja, 24, 504
faze, 47
proces razvoja, 54, 424
proces, integracija, 35
proces, izvršavanje, 73
proces, modeli, 11, 48, 64, 77
dinamički, 64, 67
fazni razvoj, 55
izrada prototipova, 53, 73
model specifikacije rada, 54
poželjne osobine alata i tehnika, 72
preuzimanje, 75
skupovi, 63
zajednički problem u cilju poređenja, 77
proces, notacije, 77
proces, programiranje, 69
proces, upravljanje, 73
procesno ograničenje, 149–150
produkcioni sistem, 56, 424
produktivnost programera, 108, 140, 362
produktivnost, 67, 109, 114, 140, 362
profesionalna licenca, 659
profesionalni razvoj, 668
programer, 25
programiranje sa N verzija, 257
programska revizija dizajna, 274, 276
proizvod rada
definicija, 526
projekat
faze, 83
projekat, alati za upravljanje, 91
projekat, hijerarhija, 102
projekat, klizanje, 88
projekat, napredovanje, 93, 94
projekat, organizacija, 128, 101–104
projekat, osoblje, 95
projekat, planiranje, 82, 123
projekat, upravljanje, 83
projektanti sistema, 430
projektanti, 25
projektni rokovi, 83
projektno ograničenje, 149–150, 153, 178, 180, 182, 195
prolazak kroz dokumente, 200
prosečno vreme
do otkaza, 441
do popravke, 441
između otkaza, 441
prosečno vreme između otkaza, 150
prostor dizajna, 244
prostor rešenja, 149
proširenje, 295
prototip za bacanje, 192, 264
provera kvaliteta, 464
provera na stolu, 367
provera redosleda, 402
provera strukture, 402
prvoklasni softver, 475
pseudokôd, 234, 346–349, 352, 534
P-sistem, 501–503, 515, 539

R

racionalni stil rada, 99
računarska nauka, poređena sa softverskim inženjerstvom, 4
računarski podržani razvoj softvera, *videti* CASE
računska greška, 367
rad za održavanje, 514, 516
radni prostor, 104, 621
radni stilovi, 99, 138, 541
radno okruženje, 621
rana većina, 634–635
rani prihvatioci, 634–635
rano tržište, 635
raspoloživo vreme, 88
raspored projekta, 83
rast pouzdanosti, 442

rasuđivanje na bazi slučaja, 116
raščlanjivanje na osnovu događaja, 32, 227
raščlanjivanje na osnovu podataka, 32, 227
raščlanjivanje, 32, 157, 227
razdvajanje nadležnosti, 157
razvoj, definicija, 15
razvojni tim
  uloge, 25
rečnik podataka, 198, 279, 354, 356, 411, 533
redokumentacija, 532, 533
refaktorisanje, 61
referenca, 288
referentni test, 448
referentni vodič poruka o otkazima, 493
regresioni testovi, 425
regresiono testiranje, 121, 390, 425, 426-437
reinženjerstvo, 532, 536
rekalibrisanje predviđanja, 606-609
rekurzija, 345
relacije
  agregacija, 304-305
  asocijacija, 304-305
  generalizacija, 304
  kompozicija, 304-305
restrikcije u prikazu klasa, 292
restrukturiranje, 532, 534
resurs, 64
resurs, opterećenje, 94
reverzni inženjering, 532, 535
režim otkaza, 466
Richardsova formula za poverenje u softver, 408
rizici
  opšti, 120
  specifični za projekat, 120
rizik, 58, 73, 466
  definicija, 120
  ozbiljnost, 74
  verovatnoća, 74
rizik, analiza, 58, 120
rizik, izloženost, 121, 122
rizik, kontrola, 120
rizik, plan za upravljanje, 123
rizik, planiranje, 122
rizik, razrešavanje, 122
rizik, sprega, 123
rizik, umanjenje, 122
rizik, upravljanje
  aktivnosti, 120-123
  definicija, 120
rizik, uticaj, 120
rizik, verovatnoća, 120
rotacije, 422-423
rukovaoci, 391, 392, 404

S

sadašnja vrednost, 594
sakrivanje informacija, 238, 290, 342, 343
samodelegiranje, 310
saradnički dizajn, 240
sastanci (*videti takođe* tehnika Delphi, inspekcije), 98, 377, 378
  dijagram sekvence poruka (MSC), 163-164
  UML stil, 210
Scrum, 60
SDL, jezik za specifikaciju i opisivanje, 189, 212-213
sejanje grešaka, 406, 619
sestrinski projekti, 546
SHARD karakteristične reči, 468
signatura operacije, 315
simboličko izvršavanje, 383
simulacija, 199, 207, 212, 451, 475-476, 481
sinhronizacija, 246, 396, 525
sintaksna greška, 367
sinteza, 2, 39
sistem
  definicija, 17
  koraci u građenju, 23-25
  primeri, 18
  značaj, 16
sistemu „ključ u ruke", 16
skladišta za ponovno korišćenje, 363, 566
skladišta, 233, 363, 566, 568
  za upravljanje konfiguracijom, 531
skript scenarija, 294, 324
skript testa, 410, 458
SLIM, 118, 138, 622
slojevitost, 20, 232
složenost dizajna, 259, 269
složenost, 505, 520, 529-530
slučaj, 294, 363
slučajan izbor, 547
slučajevi korišćenja, 294, 297
slučajna ispravnost, 411
smanjenje
a softvera (SCR), 190
smernice za obučavanje, 487
snaga veza, 248, 340, 342
softver
  upotrebe, 1
softverska arhitektura, *videti* arhitektura
softverska okruženja, 35
softverski alati, *videti* alati
softverski proces, 32, 631
softverski životni ciklus, 46
softversko inženjerstvo
  budućnost, 629

izmene u, 27
kao rešavanje problema, 2
u poređenju sa računarskom naukom, 4
uloge u, 14
uspesi u, 5
spasilački kôd, 361
specifikacija
nivo, 194
specifikacija i procena testova, 452, 452–453
specifikacija ponašanja procesa zasnovana na pravilima, 70
specifikacija rada, 54
specifikacioni jezici, 187–191
SPICE, 12, 587–588
spiralni model, 58, 73
Win-Win model, 132
spoljašnja dokumentacija, 351, 354–355
sposobnost analitičara, 114
sprečavanje grešaka, 256
srednja veličina relativne greške, 117–118
S-sistem, 500, 503, 515, 538
stabilnost pouzdanosti, 442
stablo sa korenom, 345
standardi programiranja, 337–340
standardi, 49
programiranja, 337–340
standardi, Defense Department 2167A, 49
statička analiza koda, 402, 531
statički model, 64
statički prikaz klasa, 291
statističko testiranje, 470, 472, 474, 620
stečena vrednost, 129
stepen softevrskog inženjerstva, 655
stepeni apstrakcije, 40, 64, 238, 239
stereotip, 174
stereotipna pretnja, 647
stilovi dizajna, 236, 270
stohastički sistem za predviđanje, 554
strategije testiranja, 124, 400–402, 452–454
stresne greške, 368
strukture kutija, 376, 469
strukture podataka, 343, 354
strukture posla (*work breakdown structure*, WBS), 84
studija slučaja kao tehnika za procenu, 546, 638
studije slučajeva za čistu prostoriju, 619–620
stvarno potrebno vreme (procena), 88
subjekt, 322

Š

šabloni dizajna, 247, 317, 318, 321

T

tabele odlučivanja, 260
tačka podešenja, 235
tačke gledišta, 146
tehnički dizajn, 224
tehnika
definicija, 4
tehnika Delphi, 107, 647–653
tehnike
modeliranje procesa, 63, 79
temporalna logika, 179
test
definicija, 385
test integracije odozgo nadole, 392
test performanse, 372, 420, 436–438
test podaci (za modele pouzdanosti), 555
test prihvatanja, 372, 420, 447–450
test slučajevi, 385, 400
test tačka, 385
testiranje dokumentacije, 437
testiranje grana, 387, 388, 408–409
testiranje i dokazivanje, 385, 389
testiranje iskaza, 387, 408
testiranje komponenti, 371
testiranje modula, 371
testiranje na nula otkaza, 445
testiranje niti, 430
testiranje objektno orijentisanih sistema, 397
testiranje od definicije do upotrebe, 387
testiranje okruženja, 437
testiranje otvorene kutije, 374
testiranje putanje, 387, 408, 409, 411
testiranje sistema (*videti takođe* testiranje), 372, 390, 417–430, 450–452
testiranje svih upotreba izračunavanja /nekih upotreba predikata, 387
testiranje svih upotreba predikata /nekih upotreba izračunavanja, 387
testiranje svih upotreba, 387, 388
testiranje toka transakcije, 389
testiranje upotrebljivosti, 437
testiranje zahteva, 151, 156
testiranje zatvorene kutije, 374
testiranje, (*videti takođe* regresiono testiranje, jedinično testiranje, integraciono testiranje, testiranje sistema, test prihvatanja, instalaciono testiranje, testiranje performanse, testiranje niti, funkcionalno testiranje)
testiranje, 125, 365, 417–430
crna i bela kutija, 374
kriterijumi završetka, 405–410

odnosi snaga, 387
poverenje, 408
proces, 372
sistem, testiranje (*videti takođe* testiranje sistema), 372, 390
zatvorena i otvorena kutija, 374, 385
testovi bezbednosti, 437
testovi kapaciteta, 436
testovi kompatibilnosti, 437
testovi konfiguracije, 436
testovi kvaliteta, 385, 437, 516
testovi ljudskih faktora, 437
testovi održavanja, 437, 512
testovi oporavka, 437
testovi opterećenja, 436
Therac-25, 42, 467
tim sa glavnim programerom, 101
tim za održavanje, 26, 515, 509-510
tim za testiranje, 374, 429-430
tim za upravljanje konfiguracijom, 27, 525
tok kontrole, 404, 533
tok rada, dijagrami, 299
topologija prstena, 236
topologija zvezde, 236, 422
tragovi događaja, 162
trajanje aktivnosti, 84
trajanje ciklusa, 56
transakcija, 28
transformacija, 28, 30, 54
transformacioni model, 54
trošak (*videti takođe* rad), 342
trošak, 28
troškovi nasuprot rokova, 342, 622-624
troškovi održavanja, 505, 514-516
trud da bi se olakšalo održavanje, 342, 508, 514-516

U

učesnik, 172, 294
ugrađeni sistemi, 32
ukrštanja (*videti takođe* analiza uticaja), 279, 531
ulazne i izlazne grane, 254, 331, 403
uloge održavanja, 508-510
uloge projektnog osoblja, 95
ultra visoka pouzdanost, 463
UML (*Unified Modeling Language*), 159-162, 188-189, 209-211
UML, 181
UML dijagram klasa, 211, 159-162
UML dijagram prelaza stanja, 165-169
UML stil dijagrama sekvence poruka (MSC), 210
UML, 298-300, 303, 307, 309-315, 335
Unified Modeling Language, *videti* UML
uniformna funkcija gustine, 442-443
unošenje neispravnosti, 610
u-plotovi, 602-604
upravljanje konfiguracijom, 123, 429, 531, 532, 423-428, 524-526
upravljanje resursima, 68, 590, 621-624
upravljanje u paketima, 125
uravnotežen rezultat, 593
usavršavajuće održavanje, 508
usidrenje i korekcije, 647, 653
uslov predstavljanja, 557
uslovno prevođenje i kontrola konfiguracije, 425-427

V

V model, 52
validacija (provera valjanosti), 198-201, 267, 372, 411, 420, 554, 557
definicija, 51
validan u užem i širem smislu, 557
veličina tima, 40, 379
verifikacija (procena), 198, 202-203, 267, 420
definicija, 51
verovatnoća problema, 74, 120
vertikalna pogodnost za praćenje, 526
vertikalna višekratna upotrebljivost, 567
verzije, 423
više projekata, 257, 465
višekratna upotrebljivost, proizvođač, 349, 565
višekratna upotrebljivost, potrošač, 349, 565
višekratna upotrebljivost softvera, *videti* višekratna upotrebljivost
vodič za programere, 493
Volere model, proces definisanja zahteva, 148, 218
vreme za tržište, 28, 511, 595, 623
vremenski testovi, 437

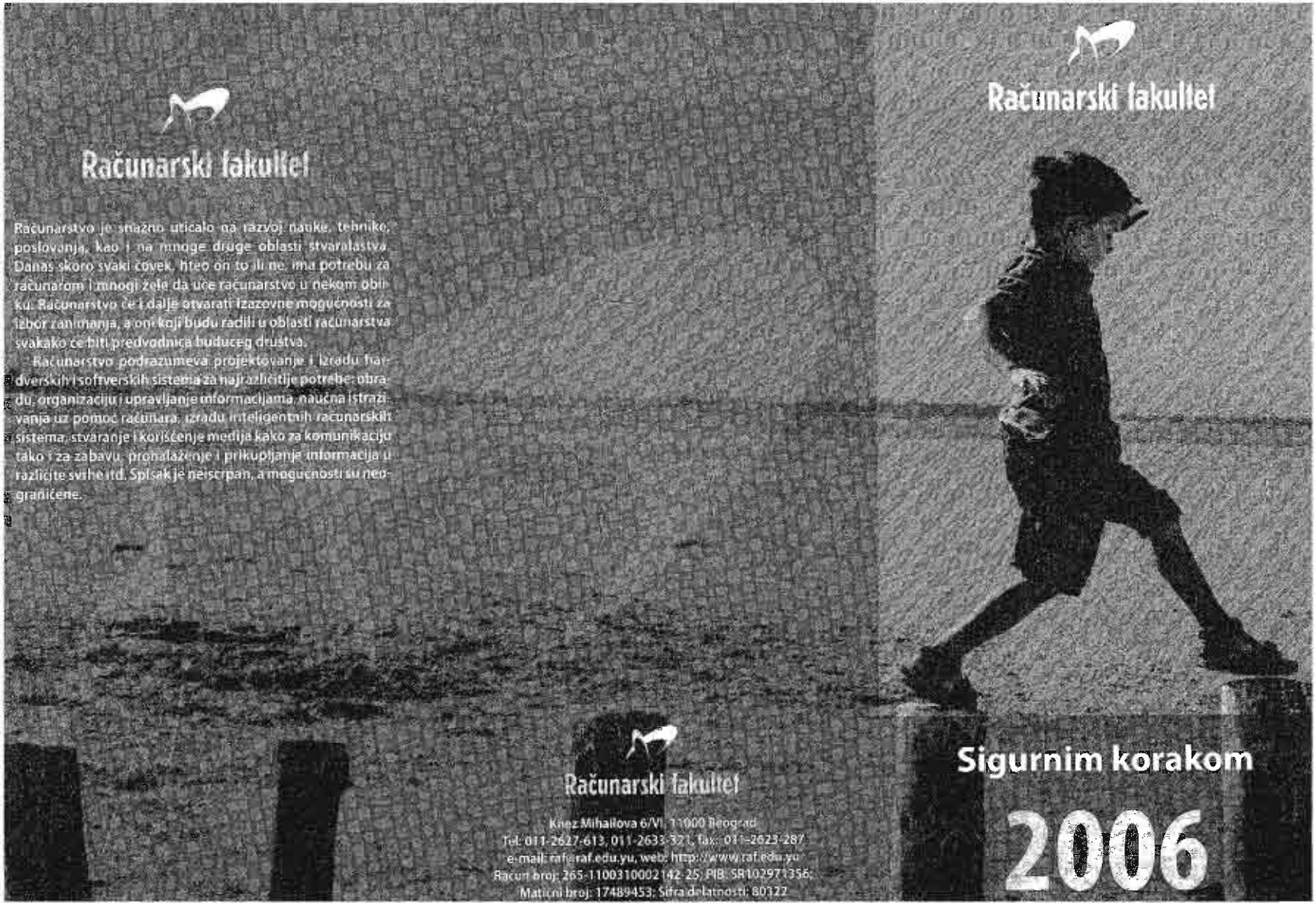
W

Walston i Felix, model, 118, 139
Wasserman, osam pojmova za disciplinu softverskog inženjerstva, 30, 630-632
Wasserman, sedam tehnoloških promena, 28
Win-Win spiralni model, 132
Wolvertonov model, 107
W-teorija, 132

Z – Ž

Z, 181-183
zadatak, 320
koga pokreće događaj, 320
koji pokreće vreme, 320
zaglavlje bloka komentara, 339, 354, 351-352

zahtev
  funkcionalni, 148
  nefunkcionalni, 149–150, 152
  u pogledu kvaliteta, 149–150, 152
zahtevi
  definicija, 143
  definicije u, 182
  dodela prioriteta, 152, 153
  domensko znanje, 196
  dosledni, 146, 155
  ispravni, 155
  izvodljivi, 156
  karakteristike, 155
  kompletni, 155
  kriterijum usklađivanja, 153
  menjanje, 39
  modelovanje, 156
  mogućnost praćenja, 156
  mogućnost testiranja, 151, 156
  objektno orijentisani, 293
  poželjne karakteristike, 206
  pretpostavke, 196, 202
  tipovi, 148–150
  tvrdnje u, 182
zahtevi servisne stanice Royal, 287
zahtevi, dokument za definisanje, 153–154
zahtevi, dokumentacija, 193–198
zahtevi, nestabilnost, 140
zahtevi, praćenje, 197
zahtevi, specifikacija, 54, 153–154, 195–197
zahtevi, specifikacioni jezici, 187–191
zahtevi, validacija, 198–199
zahtevi, verifikacija, 202–204
zainteresovani subjekt, 146–147
zasebne datoteke za kontrolu konfiguracije, 425-427
zaštitnik, 247, 249
zrelost sposobnosti, *videti* CMM
životni ciklus, 46, 503, 505
  promene u toku, 504


Računarski fakultet
Računarstvo je snažno uticalo na razvoj nauke, tehnike, poslovanja, kao i na mnoge druge oblasti stvaralaštva. Danas skoro svaki čovek, hteo on to ili ne, ima potrebu za računarom i mnogi žele da uče računarstvo u nekom obimu. Računarstvo će i dalje otvarati izazovne mogućnosti za izbor zanimanja, a oni koji budu radili u oblasti računarstva svakako će biti predvodnici budućeg društva.
Računarstvo podrazumeva projektovanje i izradu hardverskih i softverskih sistema za najrazličitije potrebe: obradu, organizaciju i upravljanje informacijama, naučna istraživanja uz pomoć računara, izradu inteligentnih računarskih sistema, stvaranje i korišćenje medija kako za komunikaciju tako i za zabavu, pronalaženje i prikupljanje informacija u različite svrhe itd. Spisak je neiscrpan, a mogućnosti su neograničene.
Računarski fakultet
Knez Mihailova 6/VI, 11000 Beograd
Tel: 011-2627-613, 011-2633-321, fax: 011-2623-287
e-mail: rafe@raf.edu.yu, web: http://www.raf.edu.yu
Račun broj: 265-1100310002142-25; PIB: SR102971356;
Matični broj: 17489453; Šifra delatnosti: 80122
Računarski fakultet
Sigurnim korakom
2006



Želite li da studirate računarstvo? Računarski fakultet je pravo mesto za vas.

Računarstvo nije jedna disciplina, nego čitava familija disciplina. Tokom poslednje dekade dvadesetog veka odigrale su se važne promene u računarstvu i komunikacijama, a uticaj koje su ove tehnologije imale na društvo, doveo je do važnih promena u disciplinama računarstva.

Studijski programi na Računarskom fakultetu

Računarske nauke

Računarske nauke obuhvataju široki spektar naučnih oblasti, od teorijskih i algoritamskih osnova do najznačajnijih pronalazaka u robotici, inteligentnim sistemima, bioinformatici i drugim uzbudljivim poljima nauke. Disciplina računarskih nauka grana se u tri kategorije:

- Projektovanje i implementacija softvera
- Istraživanje novih oblasti primene računara
- Rešavanje problema u računarstvu

Ukoliko želite da se bavite razvojem računarskih programa, programskim jezicima, algoritmima, arhitekturom i organizacijom računara, računarskim mrežama i operativnim sistemima, računarskom grafikom i vizualizacijom, upravljanjem informacijama, multimedijalnim sistemima, kao i da se bavite naučno-istraživačkim radom, ili da istražujete nove oblasti primene računara, ovo je studijski program za vas.

Računarsko inženjerstvo

Računarsko inženjerstvo je disciplina koja se bavi projektovanjem računara i sistema koji su zasnovani na računaru. To uključuje izučavanje hardvera, softvera, računarskih komunikacija, kao i njihovu međusobnu interakciju. Trenutno, dominantna oblast računarskog inženjerstva su ugrađeni sistemi, tj. uređaji koji imaju u sebi ugrađen softver i hardver. Na primer, uređaji kao što je mobilni telefon, CD plejer, digitalni video rekorder, alarmni sistemi, rendgen, laserski medicinski instrumenti, zahtevaju integraciju hardvera i ugrađenog softvera, i svi su oni proizvod računarskog inženjerstva.

Ukoliko vas interesuju mobilni uređaji, bežične komunikacije, obrada multimedijalnih signala, upravljanje pomoću računara i ostale oblasti na granici elektronike i računarstva, a sve to sa aspekta inženjera, tj. nekoga ko stvara novi proizvod na osnovu dobro utemeljenih znanja, ovo je studijski program za vas.

Informacioni sistemi

Integrišu dostignuća informacionih tehnologija i poslovne procese da bi odgovorili na izazove savremenog poslovanja. Sa tehnološke strane informacioni sistemi povezuju računarske aplikacije sa tradicionalnim oblastima u preduzetništvu.

Ukoliko želite da izučavate koja vrsta informacije je potrebna preduzeću, kako se stvara informacija, da li se ona isporučuje ljudima kojima je potrebna, da li im je predstavljena na odgovarajući način, da li je organizacija strukturirana tako da efikasno koristi tehnologiju, da li su poslovni procesi dobro osmišljeni, da li se u potpunosti koriste pogodnosti koje je stvorila informaciona tehnologija, da li je organizacija sposobna da se dovoljno brzo prilagođava promenljivim spoljnim uslovima, kao i da rešavate slična pitanja i probleme, studijski program *Informacioni sistemi* je upravo za vas.

Informacione tehnologije

Naziv *Informacione tehnologije* ima dvostruki značaj. U najširem smislu ovaj termin se koristi za sve što se tiče računarstva. U akademskom smislu, ovaj termin pokriva studijske programe koji pripremaju studente da odgovore na potrebe za računarskom tehnologijom u savremenim poslovnim sistemima, vladi, zdravstvu, školama i drugim organizacijama. U prethodnom delu rekli smo da se *Informacioni sistemi* fokusiraju na pojam „informacije" vezane za „informacione tehnologije". Informacione tehnologije su u tom smislu komplementarne: njihovo težište je više na samoj tehnologiji, a manje na informaciji. To je nova disciplina koja se razvija vrtoglavom brzinom i koja je nastala iz potrebe da se odgovori na praktične, svakodnevne zahteve poslovanja unutar različitih organizacija. Danas, sve organizacije zavise od informacione tenologije.

Ukoliko želite da se bavite planiranjem i upravljanjem informaciono-tehnološkom infrastrukturom organizacije, želite da ovladate posebnim veštinama i znanjima, kao što je razumevanje sastava i strukture umreženih sistema, da budete odgovorni za pouzdanost, bezbednost kao i za efikasnu upotrebu računarskih resursa, ovaj studijski program je za vas.

Softversko inženjertsvo

Softversko inžinjerstvo je disciplina koja se bavi razvojem i održavanjem softverskih sistema, sa ciljem da budu pouzdani, efikasni i isplativi, kao i da ispunjavaju sve zahteve korisnika. Softversko inženjerstvo je po svom karakteru različito od drugih inženjerskih disciplina, kako zbog neopipljive prirode softvera, tako i zbog diskretne prirode softverskih operacija. Softversko inženjerstvo integriše principe matematike i računarskih nauka sa ustaljenom inženjerskom praksom razvijenom za opipljive, fizičke proizvode. Studijski programi Računarske nauke i *Softversko inženjerstvo* imaju mnogo dodirnih tačaka, ali se na studijama softverskog inženjerstva težište stavlja na razvoj i održavanje zaista korisnog i upotrebljivog softvera.

Ukoliko vas interesuje profesionalni inženjerski razvoj i održavanje softvera tokom celog njegovog životnog ciklusa, upravljenje softverskim projektima, kontrola kvaliteta i unapređenje softverskih procesa, metrika i ekonomika softvera i slični izazovi, ovo je studijski program za vas.

dijske programe, izučavaju se sledeće discipline:

- Računarske nauke
- Računarsko inženjerstvo
- Informacioni sistemi
- Informacione tehnologije
- Softversko inženjerstvo

Imajući u vidu da računarstvo nudi široku lepezu zanimanja, nemoguće je da jedna osoba istovremeno postane stručna za sve. Stoga, neko ko želi da postane profesionalac u računarstvu mora da se opredeli za jednu od disciplina, a samim tim i za jedan od studijskih programa.

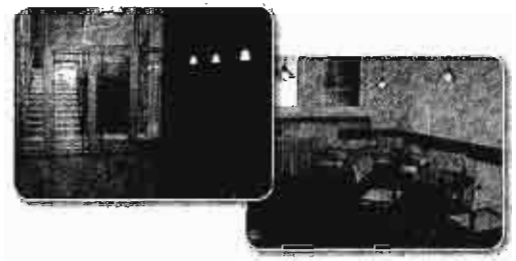
Način studiranja

Svi predmeti u okviru odgovarajućeg studijskog programa su jednosemestralni. Ocena iz predmeta dobija se na osnovu pokazanih rezultata u toku nastave, domaćih zadataka, urađenih obaveznih vežbi, kolokvijuma i završnog ispita. Na početku nastave iz svakog predmeta, studenti se upoznaju sa načinom ocenjivanja iz tog predmeta. Tokom studija obavlja se redovna kontrola kvaliteta planova i programa. Studenti periodično ocenjuju nastavnike, program i proces izvođenja nastave.

Osnovne studije traju osam semestara. Student koji ispuni sve zahteve na odgovarajućem studijskom programu, dobija zvanje **Diplomirani inženjer računarstva**.

Cena studiranja

Cena studija zavisi od broja upisanih predmeta. Po svakom ECTS kreditu upisanog predmeta, plaća se 50€. Prilikom upisa nove školske godine potrebno je upisati minimalno onoliko predmeta koliko je potrebno za 60 ECTS kredita, što iznosi 3000€. Ova cena uključuje predavanja, vežbe i prateću literaturu, bilo u obliku knjiga bilo u elektronskom obliku. Za avansno plaćanje cele sume, važi popust od 10%. U slučaju plaćanja na rate uplaćuje se 40% sume pri upisu (1200€), sledeća rata od 30 % (900€) pri overi zimskog semestra, a treća (900€) do kraja aprila sledeće godine.



РАЧУНАРСКА ГИМНАЗИЈА
У БЕОГРАДУ

Рачунарска гимназија је почела са радом школске 2005/2006. године уписом прве генерације ученика у први разред по решењу Министарства просвете бр. 022-05-121/2004-03 од 11.04.2005. године. Прилагођује се заставни план и програм за стицање образовања одарених ученика у Рачунарској гимназији објављен у Просветном гласнику број 26/2004.

Школовање у Рачунарској гимназији траје **четири школске године** и наставе се одржава на **српском језику**. Кандидати за упис у Гимназију морају имати стечено основно образовање. У Гимназију се могу уписати кандидати који **успешно положе пријемни испит из математике**.

ПРИЈЕМНИ ИСПИТ

Полагање пријемног испита је из математике и обавиће се средином јуна у Рачунарској гимназији у Кнез Михаиловој 6/VI. Задаци за пријемни испит ће бити из Збирке задатака из математике за квалификациони испит за упис у средње школе објављене у издању Просветног прегледа.

Ученици који желе да се упишу у први разред Рачунарске гимназије полажу **пријемни испит** на коме могу да освоје **максимално 70 бодова**.

Поред броја освојених на пријемном испиту из математике, вреднују се и:

- **општи успех** постигнут у шестом, седмом и осмом разреду тако што се сабере средње оцене из свих VI, VII и VIII разреда основне школе и помножи са бројем 2
- **укупан број бодова** заокружује се на две децимале. На основу општег успеха, кандидат може да оствари највише 30 бодова.

За десет најбољих ученика, обезбеђене су стипендије. На конкурс могу да се пријаве ученици који су освајали награде на такмичењима из информатике, математике и физике у организацији Министарства просвете и спорта, Друштва математичара и Друштва физичара. Заинтересовани кандидати би требало да поднесу пошта бна документа, најкасније до почетка јуна. Десет ученика са највећим бројем бодова, изабе могућност да се бесплатно школује у Рачунарској гимназији. Начин бодовања је следећи:

	Окружно	Републичко	Савезно	Балканијада
Прво место	7	6	8	12
Друго место	2	5	6	11
Треће место	1	4	7	10

Редослед кандидата се утврђује према укупном броју бодова које је кандидат стекао на основу пријемног испита, општег успеха у шестом, седмом и осмом разреду основне школе.

Цена школовања за једну годину је 1400 евра у динарској противвредности по средњем курсу НБС на дан уплате. Ова сума је гарантована за све четири године школовања.

У цену су укључена предавања, вежбе, менторски рад из рачунарства, математике и физике, два комплета литературе, који може бити у штампаном или електронском облику (један комплет литературе остаје у власништву ученика, а други, школски, остаје у ранијој години, на располагању у ученицима текућом години), екскурзија, осигурање ученика. За авансно плаћање пуне суме, одобрава се попуст од 10%. У случају плаћања на рате уплаћује се 40% суме при упису, друга рата од 30% до 31. децембра, а трећа до 10. марта.

Ученици који први разред заврше са одличним успехом остварују попуст од 5% при упису у наредну школску годину. Ученици који освоје награду на информатике, математике или физике у организацији Друштва математичара Србије или Друштва физичара бар на републичком нивоу остварују попуст од 25% при упису у наредну школску годину.

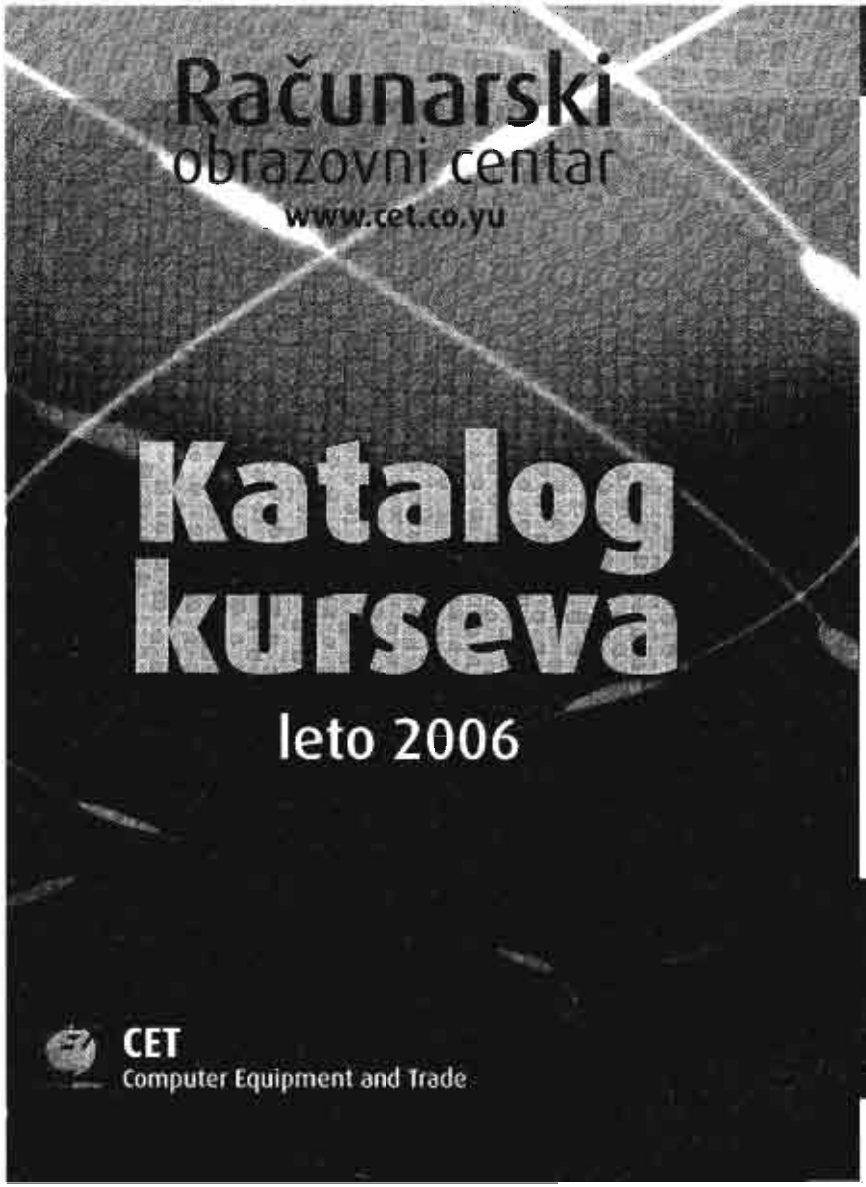
ДОКУМЕНТА ПОТРЕБНА ЗА УПИС

Потребно је поднети следећа документа:

1. **пријава за упис** (стандардан образац) коју ученик или његов родитељ попуњава и потписује;
2. **сведочанство о завршеној основној школи**;
3. **извод из матичне књиге рођених**;
4. **доказ о освојеним наградама или похвалама на такмичењима из математике, физике и информатике; Вукову диплому;**
5. **дипломе или доказе о другим постигнутим резултатима** (за случај да по броју бодова не може да се утврди редослед кандидата).

Све остале информације, можете добити у просторијама школе у Кнез Михаиловој улици 6/VI од 8 до 16 часова, или на телефоне 011/639-104, 636-090.

sekretarijat@rg.edu.yu
http://www.rg.edu.yu



bavi se obukom za rad na računarima, prodajom licencnog softvera, izdavaštvom i prodajom domaće i strane kompjuterske literature i konsaltingom.

CET je ... ▼

Školski centar

Microsoft Gold Certified Partner: Program Microsoft Gold Certified Partner ukazuje na Microsoft CTEC (Certified Technical Education Center) partnera sa iskustvom i znanjem potrebnim za osmišljavanje, pripremanje, prilagođavanje i obavljanje obuke za velike korisnike i organizacije. Zadovoljivši stroge Microsoftove kriterijume, CET je dobio status Microsoft Gold Certified Partner for Learning Solutions.

Microsoft Certified Technical Education Center: CET je Microsoftov sertifikovani tehnički obrazovni centar. Microsoft **CTEC** je partnerska obrazovna ustanova sertifikovana da pruža specijalizovana znanja najzahtevnijem krugu korisnika Microsoftovih proizvoda. Školovanje je isto u svakom centru širom sveta, bazirano na definisanim **Microsoft Official Cirriculum** pravilima i izvodi ga trener posebno sertifikovan za taj konkretni seminar.

Adobe Certified Training Provider: U organizacijama sa statusom **Adobe Certified Training Provider** obuka za rad sa Adobeovim programima sprovodi se po standardima propisanim od strane poznate softverske firme za dizajn Adobe. ACTP promoviše polaznike kurseva kao profesionalce u oblasti kompjuterske grafike, foto i video post produkcije i Web dizajna.

Autodesk Training Center: Autodesk Training Center organizacije autorizovane su od strane Autodeska da pruže praktičnu obuku vođenu iskusnim instruktorima za korišćenje Autodeskovog softvera. ATC centri su jedine ustanove koje nude autorizovanu obuku i koje prate Autodeskove stroge standarde za dobijanje originalne diplome. Na intenzivnim kursevima dolazi se do rešenja koja odmah mogu biti primenjena u praksi.

Corel Training Partner: Corel Training Partner (CTP) pruža tehničku obuku za profesionalni rad na proizvodima firme Corel i kao takav je prepoznat i priznat kao partner kompanije Corel. Pridruživši se CTP programu, nezavisni školski centri postaju bliski partneri sa Corelom. Ovaj sveobuhvatan program je posebno namenjen specijalizovanim organizacijama koje pružaju profesionalnu obuku korisnicima koji koriste Corelove grafičke aplikacije.

Član CompTIA udruženja: CompTIA okuplja kompanije koje se bave tehnologijama računarske industrije. Udruženje razvija standarde koji ne zavise od proizvođača u oblasti e-trgovine, usluga, i obukom. U okviru obuke osmišljeni su kursevi i ispiti koji označavaju poznavanje hardvera. Internet tehnologija i projektovanja mreža.

Pearson VUE centar za testiranje: Virtual University Enterprises (VUE) je organizacija koja pruža informatičke usluge obezbeđujući softver, aplikacije i sistem za prikupljanje rezultata znanja pojedinaca. Preko Pearson VUE-a mogu se polagati ispiti za dobijanje sertifikata mnogih poznatih softverskih i hardverskih kuća kao što su: Microsoft, Novell, IBM, CompTIA, Linux Professional Institute, Informix Software, itd.

CET
Computer Equipment and Trade
11000 Beograd
Računarski obrazovni centar: Kneza Mihaila 6/VI,
tel/fax: (011) 3281-614; 3281-720; 3282-995; 631-626
Knjižara: Skadarska 40b, tel/fax: (011) 3243-043; 3237-246; 3235-139, 3243-313
www.cet.co.yu skola@cet.co.yu

Škola računarstva i informatike za predškolsku i školsku decu od 5 do 10 godina



Kreativnost postaje princip humanog življenja. Želimo takvo obrazovanje koje će biti usmereno na razvoj snažnih stvaralačkih ličnosti koje će stručnom, naučnom i stvaralačkom sposobnošću dostići nivoe u razvijenom svetu.

Stvaranje takvog edukativnog programa, koji će stvoriti fleksibilnost u kreiranju nastavnih aktivnosti, kvalitet, interaktivan odnos **edukator - učenik** i prilagođavanje potrebno razvoju sredine, **osnovni su kvaliteti obrazovnog sistema**.

- kompjuterski bukvar
- kompjuterski crteži
- Internet kutak
- kompjuterska igraonica - igrice za decu odabrane prema uzrastu
- razvoj veština i znanja, samostalno korišćenje PC računara, kao velike igračke koju volimo.

Kompjuterska igraonica radionica

Igranje i upoznavanje sa kompjuterom.

Obrazovni program predviđen je za predškolsku i školsku decu od 5 do 10 godina.

Cena

Cena nastave je **80** bodova bez poreza. Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti. Nastava traje 30 časova i organizuje se subotom po dva školska časa.

Škola računarstva i informatike za učenike

Škola računarstva CET organizuje nastavu za učenike osnovnih i srednjih škola iz oblasti računarstva i informatike. Iskusni profesori uvešće vas u osnovne tehnike rada sa dokumentima, objasniti vam tehnike progamiranja i osposobiti da napravite vašu igricu, vaš sajt...

Osnovni nivo (60 časova)

1. Upoznavanje sa grafičkim operativnim sistemom zasnovanim na prozorima
2. Pokretanje programa i korišćenje jednostavnih programa (kalkulator, paint, notepad...)
3. Rad sa datotekama i periferiskim uređajima (kopiranje, premeštanje...)
4. Podešavanje karakteristika grafičkog radnog okruženja (izgled prozora, način prikaza ikona, rezolucije ekrana...)
5. Ubacivanje različitih objekata i njihovo uklapanje u tekst
6. Rad sa tabelama, generisanje obrazaca, cirkularnih pisama...
7. Rad sa tabelama, kreiranje tabele, formatiranje, tabelarne kalkulacije, povezivanje sa bazama podataka, filtriranje podataka, sortiranje tabela, grafički prikazi tabela, kreiranje različitih izveštaja...

Web radionica (60 časova)

1. Kreiranje, čuvanje, otvaranje dokumenata
2. Photoshop
3. Obrada crteža i fotografija na računaru
4. Dreamweaver

Progamiranje (60 časova)

1. Pojam algoritma, korišćenje algoritama u rešavanju problema
2. Opis nekih jednostavnih algoritama
3. Programski jezici, proceduralni programski jezici, zapis podataka u izabranom programskom jeziku
4. Napišimo svoj program
5. Testiranje programa i otklanjanje grešaka u programima

Cena

Cena nastave je **150** bodova bez poreza. Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti. Uplatu je moguće izvršiti u tri mesečne rate. Prva rata se plaća prilikom upisa, a ostale dve posle mesec dana tj. posle dva meseca. Ukoliko iz iste porodice školu pohađaju bar dva učenika ostvaruju popust od 10%. Ukoliko uplatu izvršite odjednom cena je **120** bodova.

Microsoft kursevi za korisnike

Ovi kursevi pokrivaju osnovni i srednji nivo znanja i namenjeni su krajnjim korisnicima. Kursevi se održavaju svakog radnog dana, po 3 ili 4 časa dnevno. Nakon odslušanog kursa moguće je polagati ispit za dobijanje **Microsoft Office Specialist Core Certification** za Microsoft Word 2003, Excel 2003, Power Point 2003, Access 2003 i Outlook 2003.

Microsoft Office Specialist

Šifra	Naziv kursa	Trajanje č/d	Cena
M13	Windows XP	20/7	70
M20	Visual Basic .NET	30/10	180
M21	Word 2003	24/8	85
M22	Excel 2003	30/10	120
M23	Access 2003	30/10	120
M24	PowerPoint 2003	24/8	90
M25	Outlook 2003	20/7	80
M26	FrontPage 2003	30/10	120
M27	Microsoft Project 2003	40/10	200

Napomene:

- Cene su u bodovima, bez poreza.
- Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti.
- U cenu je uračunata prateća literatura.

Microsoft CERTIFIED Technical Education Center

CISCO kursevi

CCNA & CCNP Engineer (Cisco Certified Network Associate & Professional)

Potreba za visoko obučenim mrežnim profesionalcima je i dalje veoma uočljiva. **CCNA** je važan početni korak profesionalaca za mreže na njihovom putu ka uspešnoj karijeri. Na ovom Cisco kursu, razvićete praktično znanje o radu modernih TCP/IP mreža. Naučićete komande i tehnike koje se koriste da bi se rešavali problemi konekcija ka hostovima, raditi na Cisco switchevima i routerima, praviti backup i restore konfiguracione datoteke i upravljati mrežnom opremom. **CCNP** kurs je logican nastavak puta jednog CCNA inzinjera. On je mnogo teži i opširniji od standardnih CCNA kurseva (predavač će vam postavljati probleme i prenositi bogato iskustvo iz prakse), nakon ovog kursa možete rešavati **80%** praktičnih problema koji se dešavaju u praksi, u našoj zemlji.

Kursevi su namenjeni menadžerima, mrežnim profesionalcima, administratorima, specijalistima i tehničarima koji konfigurišu, održavaju i rešavaju probleme velikih mreža u kojima se nalaze routeri, switchevi i access serveri i individualcima koji žele da dobiju CCNA & CCNP sertifikat.

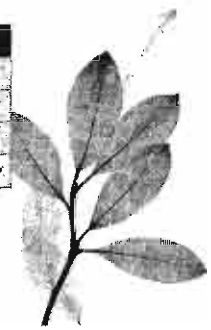
Naziv kursa	Trajanje č/d	Cena
CCNA Engineer (Cisco Certified Network Associate)	80/20	1400
CCNP Engineer (Cisco Certified Network Professional)	160/40	2800

Adobe Professional kursevi

Adobe zvanični kursevi su oni kursevi koji se predaju isključivo u zvaničnim Adobeovim trening centrima, ili ACTP (Adobe Certified Training Provider).

Kursevi su u potpunosti priprema za odgovarajuće ispite, za sticanje diplome ACE. Ovo zvanje stiče se za svaki Adobeov program, posebno. Kursevi se dele po programima (npr. Photoshop, Illustrator itd.) i nema posebnih nivoa kurseva. Na kraju se dobija sertifikat o odslušanom kursu, ali ne kompanije Adobe, već škole koja je autorizovana da ih izdaje, odnosno da izvodi autorizovanu nastavu.

Šifra	Naziv kursa	Trajanje č/d	Cena
AP5	Adobe Premiere Pro	80/10	300
AP6	Adobe Indesign CS2	20/5	250
AP7	Adobe Photoshop CS2	40/5	200
AP8	Adobe Illustrator CS2	40/5	330



Napomene:

- Cene su u bodovima, bez poreza.
- Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti.
- U cenu je uračunata prateća literatura.

Adobe Certified Training Provider

Adobe

Grafički kursevi

Grafičke radionice su skup kurseva o grafičkim aplikacijama, znanjima i „veštinama" potrebnim da bi se poznavanje rada u ovim aplikacijama primenilo u praksi.

Šifra	Naziv kursa	Trajanje č/d	Cena
GD1	**Grafička radionica:** CorelDraw13, Adobe Photoshop CS2 i QuarkXpress 6	60/15	260
GD2	**Adobe grafička radionica:** Adobe Illustrator CS2, Adobe Photoshop CS2 i Adobe InDesign CS2	60/15	350
WR1	**Web radionica:** Adobe Photoshop CS2, Macromedia Dreamweaver 8 i Macromedia Flash 8	60/15	300
DF1	**Digitalna fotografija**	40/10	250
VKM	**Program vizuelnih kompjuterskih medija**	500/125	4000

Napomene:

- Cene su u bodovima, bez poreza.
- Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti.
- U cenu je uračunata prateća literatura.

Autodesk Professional kursevi

AOTC (Autodesk Official Training Courseware) jesu zvanični kursevi kompanije Autodesk. Predaju se isključivo u zvaničnim školama ili **ATC** (Autodesk Training Center), a predavači moraju biti sertifikovani, za svaki kurs posebno. Uglavnom su podeljeni po nivoima, a prate ih zvanične AOTC skripte. Autodesk ne izdaje IT sertifikate, tako da se jedino može dobiti Autodeskov zvanični sertifikat o uspešno savladanoj nastavi.

Šifra	Naziv kursa	Trajanje č/d	Cena
DP2	3D Studio Max 6, Fundamentals	40/5	200
DP3	Mechanical Desktop 6	60/15	200
DP5	Mechanical Desktop Surface Modeling	40/5	250
DP7	AutoCad 2004 level I	60/15	220
DP8	Mechanical Desktop 2004	60/15	220
DP9	AutoCad 2004 level II (3D)	40/5	220
DP10	AutoCad 2005 level I	60/15	220

Napomene:

- Cene su u bodovima, bez poreza.
- Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti.
- U cenu je uračunata prateća Autodesk literatura.

autodesk
authorized training center

CompTIA Professional kursevi

CompTIA (Computing Technology Industry Association) je međunarodno udruženje IT kompanija, čije su članice sve poznate svetske firme. CET je prvi i za sada, jedini član CompTIA-e, ali ujedno i A+ Certification Program Training Provider, odnosno trening centar za program obuke servisera računara (A+). Ovaj jedinstveni kurs u svetu u potpunosti zadovoljava potrebe osoba zainteresovanih da održavaju, popravljaju ili sklapaju hardver, bez obzira na to ko je njihov proizvođač. Po slušanju kursa dobija se sertifikat autorizovane škole o odslušanoj nastavi, a za sticanje sertifikata, odnosno zvanja A+ Certified Service Technician treba položiti ukupno dva, odgovarajuća ispita.

Šifra	Naziv kursa	Trajanje č/d	Cena
CP1	A+ program (za servisere)	45/5	280
CP2	Network +	54/6	400

Napomene:

- Cene su u bodovima, bez poreza.
- Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti.
- U cenu je uračunata prateća literatura i ručak.

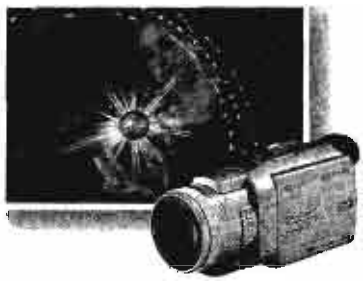
CompTIA.

Program vizuelnih kompjuterskih medija

Ovaj program je kreiran za slušaoce koji žele da se bave novim trendovima u kompjuterskom dizajnu i svoju karijeru nastave u reklamnim agencijama kao i raznim medijsko-produkcijskim kućama koje se bave Internetom, digitalnim medijima, časopisima, televiziji i filmu u ulozi 2D grafičkog i web dizajnera, audio-video nelinearnih montažera i 3D animatora sa vizuelnim efektima u post produkciji.

Program je podeljen u **četiri semestra** (500 časova) i koncipiran na bazi predavanja i praktičnih vežbi. Svaki radni dan se održavaju po četiri školska časa. Vaš računar Vam je na usluzi svaki dan još četiri časa. Svaki polaznik dobija odgovarajuću literaturu.

Iskusni profesori, profesionalci iz struke, upoznaće vas sa praktičnim znanjima profesije, tajnama reklamne industrije vodeći vas dublje u prirodu dizajna multimedija, interaktivnih prezentacija i produkcionih procesa. Radeći zajedno na projektima u umetničke i komercijalne svrhe za prave klijente, imaćete prave reference kao dokaz budućim poslodavcima i poslovnim partnerima o vašem iskustvu u profesiji.



CENA

Cena obuke je **4000 €**.

SPECIJALNI POPUST:

Za uplate do 23.6.2006. cena je **2800 €**

Program počinje **4.9.2006.**
Broj mesta je ograničen.

Programski jezici

Šifra	Naziv kursa	Trajanje č/d	Cena
PR00	Dinamičke strukture podataka u proceduralnim programskim jezicima	15/5	65
PR01	Objektno programiranje C++	40/10	155
PR02	Java 2.0	40/10	155
PR03	PASCAL	40/10	120
PR04	XML	40/10	245
PR05	Delphi	40/10	300
PR06	Java game design	60/15	250
PR07	PHP	40/10	250

Napomene:

- Cene su u bodovima, bez poreza.
- Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti.
- U cenu je uračunata prateća literatura.

Digitalna fortografija

U okviru ovog kursa polaznici će se upoznati sa fotografisanjem, skeniranjem i digitalnom obradom fotografija. Kurs je podeljen u dva dela. Prvi deo je posvećen fotografiji i digitalnim fotoaparatima. Drugi deo se odnosi na obradu fotografija pomoću računara. To je kompletan kurs za Adobe Photoshop CS2, najpopularniji i najkvalitetniji profesionalni programu za obradu slika.



Trajanje kursa: 40 časova (dve nedelje)

Cena: 250 € u dinarskoj protivvrednosti (u cenu je uračunata prateća literatura.)

Ispiti za međunarodne IT sertifikate

U CET-u kao **Pearson VUE** autorizovanom centru za testiranje možete polagati ispite za međunarodne IT sertifikate sledećih kompanija:

Šifra	Kompanija	Naziv programa	Cena
MP1	Microsoft	Microsoft Certified Professional	80
MP2	Microsoft	Microsoft Office Specialist	50
CT1	CompTIA	A+	180
CI1	Cisco	Cisco	205
LN	Linux	Linux	150
ACE	Adobe	Adobe Certified Expert	180

PEARSON VUE
AUTHORIZED CENTER

Ispit možete prijaviti lično kod nas, ili telefonom. Termini za polaganje ispita su svaki radni dan, od 9:00 do 15:00 časova. Ispiti se moraju prijaviti i/ili odložiti najmanje dva radna dana pre datuma polaganja ispita za Microsoft ispite, odnosno najmanje 15 dana za A+ i Cisco ispite.

Napomene:

- Za polaznike naših kurseva važi popust od 20% za Microsoft i Adobe ispite i popust od 10% za CompTIA ispite. Za Cisco i Linux ispite nema popusta.
- Ne postoje dodatni popusti za ponovni izlazak na isti ispit. Svaki izlazak na ispit se posebno naplaćuje.
- Cene su u bodovima. Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti.

Authorized Instructor

Sertifikat IC³
(Internet and Computing Core Certification)

IC³ je međunarodni sertifikat za svakoga ko želi da savlada osnovne kompjuterske i internet veštine koje današnje akademsko i profesionalno okruženje zahteva. Za pohađanje ovog kursa nije potrebno kompjutersko predznanje. IC³ predstavlja idealnu osnovu za MOS i CompTIA A+.

Za IC³ sertifikat je potrebno položiti sledeća 3 ispita:

- Kompjuterske osnove (hardver, softver, korišćenje osnovnih sistema).
- Ključne aplikacije (uobičajene programske funkcije, funkcije za obradu teksta, spreadsheet funkcije, funkcije za prezentaciju softvera).
- Living Online (mreža i internet, elektronska pošta, korišćenje interneta, uticaj kompjutera i Interneta na društvo).

Microsoft Professional kursevi

MOC (Microsoft Official Curriculum) su zvanični kursevi kompanije Microsoft koji se organizuju kroz lanac autorizovanih škola, odnosno **CTEC** (Certified Technical Education Center). Kursevi traju do pet dana, prate ih zvanične skripte. Svi kursevi su ujedno i priprema za polaganje ispita za sticanje Microsoftovih sertifikata. MOC kursevi mogu da se predaju isključivo u zvaničnim trening centrima, a predavači moraju biti sertifikovani. Kursevi su podeljeni po temama. Na kraju kurseva dobijaju se Microsoftovi univerzalni sertifikati o odslušanoj nastavi.

Spisak MOC kurseva, kao i njihov detaljni opis, možete pronaći na adresi:
http://www.microsoft.com/learning/training/find/findcourse.mspx

Kreirajte svoju profesiju

Ovde je početak Vašeg profesionalnog puta

2559 – Introduction to Visual Basic .NET Programming with Microsoft .NET

- Uvod u Visual Basic .NET
- Rad sa formama i kontrolama
- Korišćenje promenljivih i nizova
- Rad sa procedurama
- Strukture odlučivanja i petlje
- Obrada grešaka
- Kreiranje menija, statusnih linija i linija sa alatkama
- Koncepti objektno-orijentisanog programiranja
- Korišćenje ADO.NET
- Isporučivanje aplikacije

Trajanje kursa: 40 časova
Cena: 300 bodova

2609 – Introduction to C# Programming with Microsoft .NET

- Uvod u C#
- Rad sa formama i kontrolama
- Korišćenje promenljivih i nizova
- Strukture odlučivanja i petlje
- Obrada grešaka
- Kreiranje menija, statusnih linija i linija sa alatkama
- Koncepti objektno-orijentisanog programiranja
- Korišćenje ADO.NET
- Isporučivanje aplikacije

Trajanje kursa: 40 časova
Cena: 300 bodova

KSP1 – Programiranje polovnih aplikacija u MS Access-u

Kurs za programiranje poslovnih aplikacija korišćenjem Access baze i Access aplikacije. Neophodna osnovna znanja o bazama podataka.

- Kreiranje baza podataka
- Tabele
- Relacije
- Indeksi
- Formulari
- Izveštaji
- Korisnički toolbarovi
- Programiranje (VBA)
- Kreiranje aplikacije
- Zaštita

Trajanje kursa: 30 časova
Cena: 300 bodova

KSP2 – Kreiranje mobilnih database aplikacija na Pocket PC uređajima

Prikaz pravljenja aplikacija za mobilne uređaje koji pristupaju bazama serverskim bazama podataka direktno ili preko XML Web servisa. Potrebna osnovna znanja o Web servisima, Windows programiranju i bazama podataka.

- Koncepti
- Pocket PC aplikacije
- Kontrole
- .NET Compact Framework & ADO.NET
- Bezbednost
- Prikaz i ažuriranje podataka
- Poziv uskladištenih procedura i prenos parametara
- Korišćenje srednjeg sloja za pristup podacima (Web servisi)
- Direktno povezivanje na SQL Server bazu
- Kreiranje instalacije

Trajanje kursa: 15 časova
Cena: 150 bodova

KSP3 – Kreiranje XML Web servisa kao srednji sloj ka bazi podataka

- Koncept
- Jednostavni WEB servis
- WEB servis i baze podataka
- Poziv uskladištenih procedura i prenos parametara
- Bezbednost
- Keširanje
- Distribucija

Trajanje kursa: 15 časova
Cena: 150 bodova

KSP4 – Rad sa SQL programskim jezikom

Namenjen početnicima u radu sa bazama podataka. Primeri upita na SQL Serveru 2000.

Trajanje kursa: 15 časova **Cena:** 150 bodova

KSP5 – SQL Server 2000 programiranje

Kurs za programiranje MS SQL Servera. Neophodno poznavanje SQL upita.

- Baze podataka
- Bezbednost
- TSQL
- Uskladištene procedure
- Korisnički definisane funkcije

Trajanje kursa: 20 časova
Cena: 200 bodova

KSP6 – SQL Server 2000 replikacija podataka

Rad sa različitim tipovima replikacija podataka na SQL Serveru 2000. Neophodno dobro poznavanje SQL Servera 2000.

- Koncept
- Tipovi replikacije
- Publikacije
- Distributer
- Pretplatnici (Push/Pull)
- Bezbednost
- Monitoring replikacija

Trajanje kursa: 15 časova
Cena: 150 bodova

Napomene:
- Cene su u bodovima, bez poreza.
- Vrednost jednog boda je jedan evro u dinarskoj protivvrednosti.

Kursevi se održavaju svake subote po 5 časova.

Specijalističko obrazovanje
pretplata za septembar 2006

Microsoft GOLD CERTIFIED Partner

Microsoft CERTIFIED Technical Education Center

Od **4. septembra 2006.** godine CET će započeti sa novim ciklusom obuke za određene profile zanimanja. U pitanju su višemesečni kursevi, vežbe i konsultacije za sledeće profile:

S01.3 - Microsoft Sistem Inženjer (MCSE) za Windows 2003 (360 časova)

- **2285** Installimg Configuration and Administering Microsoft Windows XP Professional
- **2273** Managing and Maintaining a Microsoft Windows Server 2003 Environment
- **2276** Implementing a Microsoft Windows Server 2003 Network Infrastructure Network Hosts
- **2277** Implementing Managing and Mainting a MS Windows Server 2003 Network Infrastructure Network Services
- **2824** Implementing Microsoft Inetrnet Security and Acceleration Server 2004
- **2278** Planning and Maintaing Microsoft Windows Server 2003 Network Infrastructure
- **2279** Planning Implementing and Maintaing a Microsoft Windows Server 2003 Active Directory Infrastructure
- **2282** Designig a Microsoft Windows Server 2003 Active Directory and Network Infrastructure

S01R - Inženjerska radionica - Win 2003 (240 časova)

Potrebno je poznavanje rada u Windowsu. Časovi su koncipirani kroz vežbe. Nastavne jedinice prate globalno gradivo kursa sa oznakom S01.3. Kroz realno okruženje rešavaju se problemi u Windowsu okruženju.

S02 - Microsoft Administrator baza (MCDBA) za SQL Server 2000 (320 časova)

(Potrebno predznanje: rad u Windowsu, iskustvo u programiranju i relacionim bazama podataka)

- **2273** Managing and Maintaining a Microsoft Windows Server 2003 Environment
- **2072** Administering a Microsoft SQL Server 2000 Database
- **2073** Programming a Microsoft SQL Server 2000 Database
- **2389** Progamming with Microsoft ADO.NET
- **2310** Developing Web Applications Using Microsoft Visual Studio .NET

S06.3 - Microsoft Office Specialist (MOS) (120 časova)

(Potrebno predznanje: Windows 9x i poznavanje osnovne tehnike rada sa Microsoft Office 2003)

- Microsoft Word 2003 Expert
- Microsoft Excel 2003 Expert
- Microsoft PowerPoint 2003
- Microsoft Access 2003
- Microsoft Outlook 2003

S07.1 - WEB Application Development (MCAD) (220 časova)

(Potrebno predznanje: osnove programiranja i iskustvo u izradi Web aplikacija)

- Introduction to Win 2000
- **2667** Introduction to Programming
- **2073** Programming a Microsoft SQL Server 2000 Database
- **2559** Introduction to Visual Basic .NET Programming with Microsoft .NET
- **2389** Programming with Microsoft ADO .NET
- **2310** Developing Microsoft ASP. NET Web Applications Using Visual Studio .NET
- **2524** Developing XML Web Services Using Microsoft ASP.NET

S08.2 - Microsoft Certified Systems Administrator (MCSA) za Windows 2003 (220 časova)

(Potrebno predznanje: rad u Windowsima i poznavanje osnovnih mrežnih i hardverskih pojmova)

- **2285** Installimg Configuration and Administering Microsoft Windows XP Professional
- **2273** Managing and Maintaining a Microsoft Windows Server 2003 Environment
- **2276** Implementing a Microsoft Windows Server 2003 Network Infrastructure Network Hosts
- **2277** Implementing Managing and Mainting a Windows Server 2003 Network Infrastructure Network Services
- **2824** Implementing Microsoft Inetrnet Security and Acceleration Server 2004

S09.4 - .NET VB Track (320 časova)

(Potrebno predznanje: rad u Windowsu, osnove programiranja i poznavanje osnove baze podataka)

- Introduction to Windows 2000
- **2667** Introduction to Programming
- **2073** Programming a MS SQL Server 2000 Datab.
- **2559** Introduction to Visual Basic .NET Programming with Microsoft .NET
- **2389** Programming with Microsoft ADO.NET
- **2310** Developing Microsoft ASP.NET Web Applications Using Visual Studio .NET
- **2524** Developing XML Web Services Using Microsoft ASP.NET
- **2565** Developing Microsoft .NET Applications for Windows (Visual Basic .NET)
- **2710** Analyzing Requirements and Defining Microsoft .NET Solution Architectures

S09.5 - .NET C# Track (320 časova)

(Potrebno predznanje: rad u Windowsu, osnove programiranja i poznavanje osnova baze podataka)

- Introduction to Windows 2000
- **2073** Programming a MS SQL Server 2000 Datab.
- **2609** Introduction to C# Programming with Microsoft® .NET
- **2389** Programming with Microsoft ADO.NET
- **2310** Developing Microsoft ASP.NET Web Applications Using Visual Studio .NET
- **2524** Developing XML Web Services Using Microsoft ASP.NET
- **2555** Developing Microsoft .NET Applications for Windows (Visual C# .NET)
- **2710** Analyzing Requirements and Defining Microsoft .NET Solution Architectures

S011.1 - Microsoft Certified Desktop Support Technician MCDST (40 časova)

- **2261** Supporting Users Running the Microsoft Windows XP Operating Systems
- **2262** Supporting Users Running Applications on a Microsoft Windows XP

S020.1 - Database Developer (MCIT) (80 časova)

- **2781** Designig Microsoft SQL Server 2005 Server - Side Solutions (3 dana)
- **2782** Designig Microsoft SQL Server 2005 Datebase (2 dana)
- **2783** Designig the Date Tier for Microsoft SQL Server 2005 (1 dan)
- **2784** Tuning and Optimizing Queries Using Microsoft SQL Server 2005 (3 dana)

S020.2 - SQL Server 2005 (MCTS) (40 časova)

- **2779** Implementing a Microsoft SQL Server 2005 Database (3 dana)
- **2780** Maintaing a Microsoft SQL Server 2005 Database (3 dana)

S020.3 - Database Administr. (MCITP) (80 čas.)

- **2786** Designing a Microsoft SQL Server 2005 Infrastructure (2 dana)
- **2787** Designing Security for Microsoft SQL Server 2005 (2 dana)
- **2788** Designig High Availability Database Solutions Using Microsoft SQL Server 2005 (3 dana)
- **2789** Administering and Automating Microsoft SQL Server 2005 Database and Services 2005 (1 dan)
- **2790** Troubleshooting and Optimizing Datebase Services Microsoft SQL Server 2005 (2 dana)

S020.4 - Business Intelegence Developer

KURS	Broj časova	Cena bez popusta	CENE U PRETPLATI do 23.6.2006.	
			Popodnevni i večernji termin	Jutarnji termin
S01.3	360	2500 €	1600 €	1500 €
S02	320	2400 €	1500 €	-
S06.3	120	500 €	400 €	-
S07.1	220	1450 €	1000 €	870 €
S08.2	228	1500 €	1050 €	900 €
S09.4	320	2200 €	1400 €	1320 €
S09.5	320	2200 €	1400 €	1320 €
S011.1	40	400 €	360 €	-
S020.1	80	750 €	685 €	-
S020.2	40	400 €	360 €	-
S020.3	80	750 €	685 €	-
S01R	240	1500 €	-	-

CENE

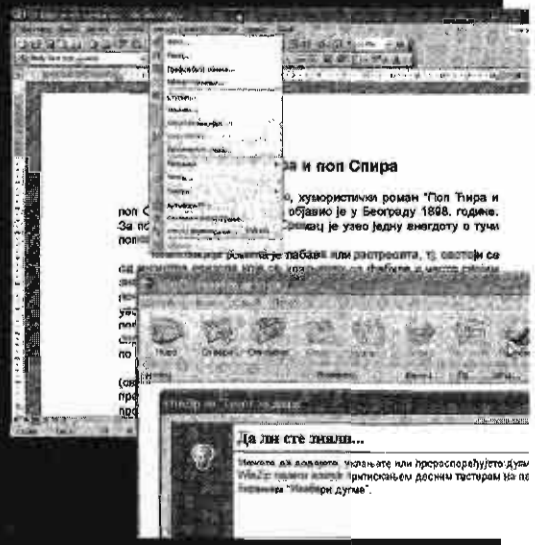
Plaćanje je omogućeno u pet jednakih mesečnih rata a za kurs **S06.3** u dve rate. Poseban popust od **20%** odobravamo za avansno plaćanje cele sume osim za kurs **S06.3**. Za firme koje na kurs pošalju više od dva polaznika, odobravamo specijalne popuste do **15%**. Za kurseve sa oznakom **S020** pohađanjem jednog kursa ostvrili ste popust za svaki sledeći od **10%**. Cene su date bez poreza.

TERMINI

Kursevi počinju **4.9.2006.** Svi polaznici dobijaju prepručenu literaturu koja je predviđena za uspešno praćenje kurseva. Za polaznike kurseva važi poseban popust od **20%** za polaganje Microsoftovih ispita za IT sertifikate. Ukoliko Microsoft predloži novu koncepciju za određena zanimanja, moguća je izmena nastave.

Piši kao što govoriš,
čitaj kako je napisano...

ЛОКАЛИЗОВАНО ЈЕ!



Najpoznatije aplikacije vodećih proizvođača softvera na srpskom jeziku, ćirilicom i latinicom.

LOKVER

WebSoft

Šta je SIAT?

SIAT je besplatan mini program koji se na vašem računaru izvršava u pozadini i koji na vaš materinji jezik automatski prevodi aplikacije koje ste izabrali. Naziv SIAT je akronim za Software Interface Automatic Translation, što znači: automatsko prevođenje softverskog interfejsa.

Po narudžbini francuske firme GlobalWare International, SIAT su zajednički razvili WebSoft i CET.

LOKVER baze - и ћирилица i latinica

LOKVER baze se sastoje od prevoda na srpski jezik, koje su realizovane u oba pisma. Kupovinom nekog prevoda dobijate obe njegove verzije. Prelazak sa jednog pisma na drugo je vrlo jednostavan i može se izvršiti bilo kada u toku rada. Osim toga, u svakom trenutku možete da se vratite i na izvorni jezik aplikacije.



Operativni sistemi

SIAT se može koristiti na praktično svim Windows platformama (počev od Windowsa 98 SE), mada najefikasnije radi sa najnovijim verzijama kao što su Windows 2000 i XP.

SIAT radi nezavisno od jezika operativnog sistema. To znači da ćete izabrane aplikacije moći da koristite na srpskom jeziku i oka imate, npr., Windows na engleskom.

Kako se nabavlja SIAT?

SIAT možete besplatno preuzeti, sa naše otpremne stranice www.cet.co.yu, sa veb lokacije GlobalWare International www.globalware.com, kao i sa mnogih popularnih veb lokacija koje omogućavaju preuzimanje besplatnog softvera. Zajedno sa SIAT-om dobijate početnu prevodilačku datoteku koja sadrži prevode više od 1000 termina koji se najčešće koriste u Windows programima.

Kako se nabavljaju prevodi?

Sve raspoložive LOKVER prevode možete naručiti na našoj stranici za naručbine www.cet.co.yu, gde ćete naći i instrukcije za plaćanje. Odmah po izvršenoj uplati, mi ćemo vam e-poštom poslati naručene prevode, zajedno sa potpunim uputstvom za njihovo instaliranje. Posebne pogodnosti pri kupovini nudimo kupcima više primeraka (licenci) jednog prevoda.

Raspoložive LOKVER baze prevoda

CET i WebSoft su do sada preveli sledeće programe na srpski jezik:

Microsoft Word 2003
Microsoft Word 2002
Microsoft Outlook Express 6
Microsoft Internet Explorer 6
Microsoft Excel 2003
Microsoft Excel 2002
Adobe Reader 6
Adobe Photoshop 7
Adobe Acrobat Reader 5



Ukoliko vaše omiljene aplikacije nije na ovom spisku, posetite našu stranicu na kojoj primamo vaše predloge aplikacija za prevod i pošaljite nam vaše sugestije. Rado ćemo prihvatiti i sve predloge za prevod aplikacija na hrvatski, slovenački ili makedonski jezik, i uvrstiti ih u našu LOKVER bazu.

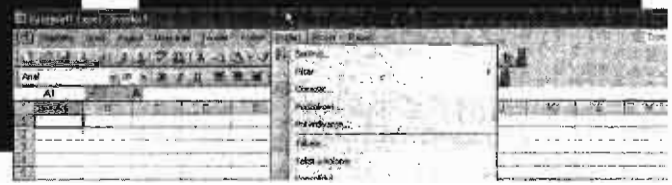
CET i WebSoft će stalno raditi na novim prevodima i objavljivati ih u određenim intervalima.

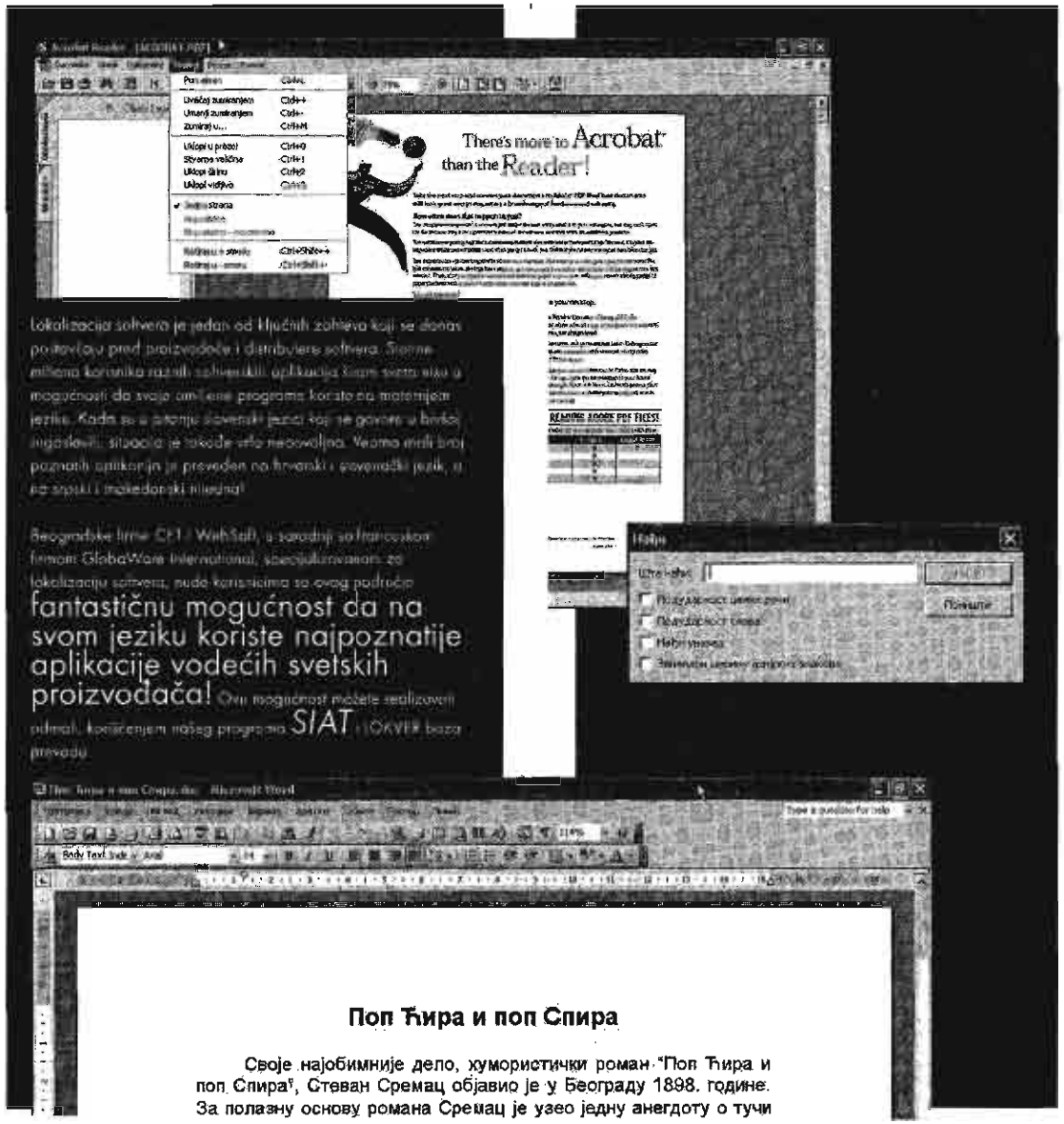
Besplatni prevodi

Sledeće prevode možete preuzeti besplatno:
WinZip 8,
Pasijans

Preuzmite ove prevode odmah, sa naše stranice www.cet.co.yu. Uverite se u efikasnost SIAT-a i kvalitet naših prevoda.

Lokalizovani programi biće podržani odgovarajućom literaturom u izdanju CET-a.





PROIZVOĐAČI I DISTRIBUTERI SOFTVERA

Napredujte brže i ekonomičnije u procesu lokalizacije!
Napredujte brže i ekonomičnije u procesu lokalizacije!
SIAT (program za automatsko prevođenje softverskog interfejsa) omogućava da vaš softver bude prevoden na neki drugi jezik bez izmene izvornog koda! SIAT uvodi novu paradigmu u lokalizaciji softvera!

Pojednostavite postupak lokalizacije!
SIAT softverska mašina u hodu prevodi korisnički interfejs bilo koje Windows aplikacije, bez obzira na njegov originalni jezik ili jezik na kome je prikazan. Svi prevodi se skladište u bazu podataka. Oni se mogu izmeniti u svakom trenutku, a efekat je trenutni. Dok pišete softver, usredsredite se na odlike vaše aplikacije, a SIAT će je učiniti lokalizovanom. Vi ćete proizvesti, distribuirati i podržavati samo JEDAN programski kod, a vaš proizvod će biti dostupan na više jezika. Softverska mašina SIAT je BESPLATNA. Ona sadrži bazu podataka prevoda najčešćih stringova, kao što su „Open", „Quit", „Print" ili „Save". Isprobajte SIAT na svom softveru i uverite se sami u njegovu efikasnost!

Smanjite troškove i skratite vreme izlaska na tržište!
Proces lokalizacije novog softvera se veoma pojednostavljuje jer izostaju projektantski poslovi. To sto nema modifikacije koda znači da nema pisanja koda za posebne jezike, nema dodatnih medijuma za distribuciju, nema potrebe za podrškom različitim izvornim kodovima i bibliotekama, nema grešaka koje su posledica lokalizacije. Skraćenje projektnog vremena direktno smanjuje troškove lokalizacije. Ova tehnologija će vam pomoći da radite sa nižim proizvodnim i administrativnim troškovima, kao i troškovima distribucije i podrške.

Proizvođači i distributeri softvera: CET i WebSoft vam nude i druge, klasične metode lokalizacije softvera na srpski, hrvatski, slovenački i makedonski jezik. Javite nam se!

CET Computer Equipment and Trade
Knez Mihailova 6/VI, 11000 Beograd
011 328-1720, 328-1614
http://www.cet.co.yu, mail@cet.co.yu

Uhvatite korak i ostanite u igri

Ovo su knjige koje sadrže sve odgovore neophodne za efikasan rad sa programima iz paketa Microsoft Office® 2003. U tome vam pomažu jasna uputstva, lekcije sa praktičnim vežbama koje možete da radite samostalno ili u našoj školi pod nadzorom stručnih predavača, saveti i rešenja koja je osmislio Microsoft a prenosi vam CET. Svaka knjiga sadrži CD sa vežbama!



Korak po korak

- Microsoft Office System 2003
- Microsoft Office Access 2003
- Microsoft Office Excel 2003
- Microsoft Office FrontPage 2003
- Microsoft Office Outlook 2003
- Microsoft Office PowerPoint 2003
- Microsoft Office Word 2003
- Microsoft Office XP

Microsoft Press

Serije **Bez po muke**, **Kao od šale** i **Brže, bolje** daju brzo odgovore na pitanja koja vas muče! Bez mnogo priče, direktna i konkretna uputstva, koncizna objašnjenja daju precizne informacije upravo onda kada su vam one potrebne. Postupci koji su vam svakodnevno potrebni objašnjeni su tako da ćete bez po muke moći sami da ih ponovite.



Kao od šale
- Digitalna muzika
- Digitalni fotoaparati
- Digitalni kućni filmovi
- Microsoft Access 2003
- Microsoft Excel 2003
- Microsoft Office 2003
- Microsoft Windows XP
- Microsoft Word 2003
- PC Kao od šale

Bez po muke
- Microsoft Office System 2003
- Microsoft Windows XP

Brže, bolje
- Digitalna fotografija
- Digitalni video
- HTML & XML
- Osnove programiranja
- Sertifikat A+
- Stvaranje stranica za Web

CET vam prenosi znanja koja dolaze pravo sa izvora.
Detaljnije informacije o našim izdanjima potražite:

- U našoj knjižari u Skadarskoj 40b, 011/3243-313
- Na našoj Web lokaciji na adresi http://www.cet.co.yu
- U preko 100 knjižara širom Jugoslavije, u Republici Srpskoj, Bosni i Hercegovini, Makedoniji i Sloveniji

CET
Computer Equipment and Trade

CET vam nudi nenadmašne priručnike koji početnicima, iskusnim korisnicima i profesionalcima omogućavaju da na najbolji način iskoriste računarsku tehnologiju.



Vodič kroz

- Microsoft Office Access 2003, specijalno izdanje
- Microsoft Office Excel 2003, specijalno izdanje
- Microsoft Office Word 2003, specijalno izdanje
- AutoCAD 2004, napredne tehnike
- Cubase VST



Udžbenici za pripremu ispita

- A+ sertifikat
- Analiziranje zahteva i definisanje arhitektonskih rešenja za MS tehnologiju .NET, MCSD, ispit 70-300
- Implementiranje, upravljanje i održavanje mreža Microsoft Windows Server 2003, MCSE, ispit 70-291
- Microsoft Exchange 2000 Server, MCSE, ispit 70-224
- Microsoft Windows 2000 Server, MCSE, ispit 70-215
- Microsoft Windows Server 2003 mrežna infrastruktura – planiranje i održavanje, MCSE, ispit 70-293
- Microsoft Windows Server 2003 okruženje – upravljanje i održavanje, MCSE, ispit 70-290
- Microsoft Windows Server 2003 servisi Active Directory – planiranje, implementiranje i održavanje, MCSE, ispit 70-294
- Microsoft Windows XP Professional, MCSE, ispit 70-270
- Osnove umrežavanja Plus, MCSE, ispit 70-58
- Razvoj Web aplikacija: VB.NET I VC#.NET, MCAD/MCSD, ispiti 70-305 / 70-315
- Razvoj Windows aplikacija VB.NET i VC#.NET, MCAD/MCSD, ispit 70-306 / 70-316
- Razvoj XML servisa i servisnih komponenti: VB.NET i Visual C# .NET, MCAD/MCSD, ispiti 70-310 / 70-320
- Sertifikat Security+
- Windows 2000, Active directory services, MCSE, ispit 70-217
- Windows 2000, administriranje mreže, MCSE, ispit 70-216



Serija **Učionica u knjizi** vam omogućuje da učite radeći na konkretnim projektima. Jednostavna uputstva vas postepeno uvode u nove odlike Adobeovih programa, a pitanja na kraju svake lekcije služe da proverite šta ste naučili. Vežbe sa pratećeg CD-a i iskustvo Adobeovog kreativnog tima su u funkciji brzog i efikasnog savladavanja materije.



Učionica u knjizi

- Adobe Acrobat 6.0 Standard
- Adobe After Effects 6.0
- Adobe Illustrator CS
- Adobe InDesign CS
- Adobe Photoshop CS
- Adobe Premiere Pro



Bukvar za nestrpljive

- Maya 6
- 3DS MAX 6
- CorelDRAW 11
- QuarkXPress 5
- Fireworks
- HTML za World Wide Web
- LightWave 3D 7.5
- Director MX
- Dreamweaver MX
- Flash MX
- Flash MX Napredne tehnike
- FreeHand MX
- Unix



CAD • matematika

- AutoCAD 2005 Osnove
- Vodič kroz AutoCAD 2004 Napredne tehnike
- AutoCAD 2002 za arhitekte
- Mechanical Desktop 6.0
- Inventor 5/5.3
- Naučite SolidWorks
- Uvod u Mathcad 11
- Uvod u MATLAB 7



Svetski bestseleri

CET vam predstavlja naslove koji su više od referentnog priručnika. U njima je izloženo znanje vrhunskih profesionalaca o konkretnim softverskim paketima, koji su to znanje sticali tokom dugog perioda rada, kako u pomaganju pojedincima ili grupama, tako i u vodećim svetskim kompanijama.





Knjige su metodološki organizovane tako da čitaoci mogu brzo i efikasno da nauče izloženu materiju. Uvodna poglavlja su sveobuhvatna i razumljiva za početnike. Autori postepeno uvode tehničke detalje. Kroz posebne postupke opisane u poglavljima za profesionalce, čitaoci uče kako da zagospodare određenim programom. Pored toga, uvršćeni su saveti i trikovi koje su autori naučili tokom dugogodišnje prakse – mnogi se ne nalaze u referentnim priručnicima ili se kroz praksu do njih teško dolazi.





CET
Computer Equipment and Trade
11000 Beograd, Skadarska 40b
tel: (011) 3243-313
http://www.cet.co.yu

РЕПУБЛИКА СРПСКА

CIP – Каталогизација у публикацији
Народна библиотека Србије, Београд

004.41

ФЛИГЕР, Шари Лоренс
Softversko inženjerstvo : teorija i praksa / Shari Lawrence Pfleeger, Joanne M. Atlee ; prevod trećeg izdanja ; [Sveta Jovanović, Jasna Gonda i Rade Janković]. – Beograd : Računarski fakultet : CET, 2006 (Čačak : Svetlost). – xvii, 716 str. : ilustr. ; 24 cm

Prevod dela: Software Engineering. – Tiraž 1.000. – Bibliografija sa napomenama : str. 673-704. – Registar.

ISBN 86-7991-284-0 (CET)
1. Атли, Џоана М.
а) Софтверско инжењерство

COBISS.SR–ID 133055500