Sveučilište u Zagrebu
Prirodoslovno Matematički Fakultet
- Matematički odjel

Robert Manger

SOFTVERSKO INŽENJERSTVO

skripta

Prvo izdanje

Zagreb, rujan 2005.

Predgovor

Ova skripta sadrže predavanja iz kolegija Softversko inženjerstvo na studiju matematike PMF-Matematičkog odjela Sveučilišta u Zagrebu. Tekst se uglavnom oslanja na udžbenik [1], tako da se taj udžbenik može smatrati glavnom literaturom. Knjige [2, 3, 4] nisu neposredno upotrebljene u oblikovanju teksta, no one imaju sličan sadržaj pa mogu poslužiti kao dodatna literatura. Skripta ne uključuju materijale za vježbe iz istog kolegija. Posljednjih godina, za vježbe su se koristili primjeri, zadaci i softverski alati iz [5, 6].

Skripta se sastoje od pet poglavlja, od koji je prvo uvodno, a ostala se bave pojedinim fazama softverskog procesa. Na taj način, slijedi se zamišljeni tijek razvoja softvera, od analize zahtjeva i specifikacije, preko oblikovanja i implementacije, do verifikacije i validacije, te na kraju održavanja odnosno evolucije. Naglasak je na "inženjerskim" (informatičkim) aspektima, budući da se za "managerske" aspekte predviđa posebni kolegij. Tekst se ne opredjeljuje ni za jednu konkretnu metodu razvoja softvera, već umjesto toga nastoji studente upoznati s s različitim (često suprostavljenim) idejama i modelima softverskog inženjerstva, te njihovim prednostima i manama. To još uvijek ostavlja mogućnost da se vježbe izvedu na konkretniji način, dakle primjenom odabrane metode i odgovarajućeg CASE alata.

Gotovo sva poglavlja u ovim skriptama oslanjaju se na priloge. Riječ je o materijalima koji će studentima biti podijeljeni na predavanjima u obliku fotokopija. Većina priloga potječe iz aktualnog (sedmog) ili prethodnog (šestog) izdanja udžbenika [1], a nalaze se i unutar powerpoint-prezentacija koje je pisac udžbenika objavio na web-u. Dodatak na kraju ovih skripti sadrži popis svih korištenih priloga s naputcima gdje se oni mogu pronaći.

U ovim skriptama spominju se brojni daljnji autori i radovi koji su značajni za nastanak pojedinih ideja, modela, metoda, ili alata unutar softverskog inženjerstva. Ta spominjanja sastoje se od navođenja imena autora te godine nastanka njegovog rada. U tekstu nisu navedene precizne reference, ali one bi se po potrebi mogle reproducirati pomoću bibliografije iz knjige [1].

Sadržaj

Poglavlje 1

UVOD U SOFTVERSKO INŽENJERSTVO

1.1 Osnovni pojmovi vezani uz softversko inženjerstvo

Jedan od ciljeva ovog kolegija je da poveže i sistematizira znanje koje smo već stekli u prethodnim "softverskim" kolegijima, kao što su kursevi programskih jezika, računarski praktikumi, kursevi o algoritmima, bazama podataka, mrežama računala i slično. Osnovni pojmovi u ovom kolegiju odnose se na problematiku razvoja softvera i već smo ih susretali u spomenutim prethodnim kolegijima. Ipak, sad tim istim pojmovima nastojimo dati preciznije značenje.

1.1.1 Softverski produkt i softversko inženjerstvo

Softverski produkt je skup računalnih programa i pripadne dokumentacije, stvoren zato da bi se prodao. Može biti razvijen za sasvim određenog korisnika (engleski *bespoke product*, *customized product*) ili općenito za tržište (engleski *generic product*). Softverski produkt često ćemo kraće nazivati *softver* ili (softverski) *sustav*.

Za današnji softver se podrazumijeva da on mora biti kvalitetan. Preciznije, od softverskog produkta se očekuje da se on odlikuje sljedećim atributima kvalitete.

- *Mogućnost održavanja.* Softver se mora moći mijenjati u skladu s promijenjenim potrebama korisnika.
- *Pouzdanost i sigurnost.* Softver se mora ponašati na predvidiv način, te ne smije izazivati fizičke ili ekonomske štete.
- *Efikasnost.* Softver mora imati zadovoljavajuće performanse, te on treba upravljati strojnim resursima na štedljiv način.
- *Upotrebljivost.* Softver treba raditi ono što korisnici od njega očekuju, sučelje mu treba biti zadovoljavajuće, te za njega mora postojati dokumentacija.

Softversko inženjerstvo je znanstvena i stručna disciplina koja se bavi svim aspektima proizvodnje softvera. Dakle, softversko inženjerstvo bavi se modelima, metodama i alatima koji su nam potrebni da bi na što jeftiniji način mogli proizvoditi što kvalitetnije softverske produkte.

Softversko inženjerstvo počelo se razvijati krajem 60-tih godina 20-tog stoljeća, kao odgovor na takozvanu "softversku krizu". Naime, pojavom računala treće generacije (na primjer IBM serija 360) stvorila se potreba za složenijim softverom (na primjer multi-tasking operacijski sustav). Pokrenuti razvojni projekti višestruko su premašili planirane troškove i rokove. Vidjelo se da se dotadašnje neformalne

tehnike individualnog programiranja ne mogu uspješno "skalirati" na velike programe gdje sudjeluje velik broj programera. Osjećala se potreba za složenijim metodama razvoja softvera, koje bi ličile na metode iz tradicionalnih tehničkih struka (na primjer mostogradnja) te bile u stanju kontrolirati kompleksnost velikog softverskog projekta.

Od 60-tih godina do danas, softversko inženjerstvo uspjelo se etablirati kao važni dio tehnike i računarstva. Softver se danas proizvodi daleko predvidljivije i efikasnije nego prije. Ipak, još uvijek postoji prostor za poboljšanje. Softversko inženjerstvo se i dalje intenzivno razvija, disciplina još nije dosegla svoju zrelu fazu, te u njoj nema "jednoumlja".

1.1.2 Softverski proces, metode i alati

Softverski proces je skup aktivnosti i pripadnih rezultata čiji cilj je razvoj ili evolucija softvera. Osnovne aktivnosti unutar softverskog procesa su: specifikacija, oblikovanje, implementacija, verifikacija i validacija, te održavanje odnosno evolucija.

Model za softverski proces je idealizirani prikaz softverskog procesa, kojim se određuje poželjni način odvijanja i međusobnog povezivanja osnovnih aktivnosti. Na primjer, model može zahtijevati sekvencijalno odnosno simultano odvijanje aktivnosti.

Metoda razvoja softvera je profinjenje i konkretizacija odabranog modela za softverski proces. Metoda uvodi specifičnu terminologiju. Također, ona dijeli osnovne aktivnosti u pod-aktivnosti te propisuje što se sve mora raditi unutar pojedine pod-aktivnosti. Dalje, metoda uvodi konkretan način dokumentiranja rezultata pod-aktivnosti (dijagrami, tabele, pseudo-jezik, ...), te daje naputke vezane uz organizaciju rada, stil oblikovanja, stil programiranja, itd.

Starije *funkcionalno-orijentirane* metode poput Yourdonove ili Jacksonove JSD (80-te godine 20-tog stoljeća) slijede logiku starijih funkcionalno-orijentiranih programskih jezika (Cobol, C, Fortran). Novije *objektno-orijentirane* metode predstavljaju nadgradnju objektno-orijentiranih programskih jezika (Java, C++) i danas su se integrirale u zajednički standard *RUP* (engleski *Rational Unified Process* - Booch, Rumbaugh, Jacobson - 90-te godine) koji je utemeljen na notaciji *UML* (engleski *Unified Modelling Language*).

CASE alati (engleski *Computer Aided Software Engineering*) su softverski paketi koji daju automatiziranu podršku za pojedine aktivnosti unutar softverskog procesa. Obično su napravljeni u skladu s određenom metodom razvoja softvera, implementiraju pravila iz te metode, sadrže editore za odgovarajuće dijagrame, te služe za izradu odgovarajuće dokumentacije.

Takozvani *upper-CASE* alati daju podršku za početne aktivnosti unutar softverskog procesa, kao što su specifikacija i oblikovanje. *Lower-CASE* alati podržavaju samu realizaciju softvera, dakle programiranje, verifikaciju i validaciju, te eventualno održavanje.

Na vježbama iz ovog kolegija obradit ćemo funkcionalno-orijentiranu metodu SADIE, te ćemo raditi s odgovarajućim upper-CASE alatom Select Yourdon (podrška za funkcionalnu specifikaciju i oblikovanje). U jednom dijelu vježbi simulirat ćemo primjenu objektno-orijentiranih metoda, te ćemo se služiti upper-CASE alatom za objektno oblikovanje i crtanje UML dijagrama ArgoUML. Na vježbama iz drugih kolegija koristi se lower-CASE alat Microsoft Visual Studio (podrška za implementaciju, testiranje, debugiranje, ...).

1.2 Modeli za softverski proces

U svim modelima više ili manje su prisutne sljedeće osnovne aktivnosti koje čine softverski proces:

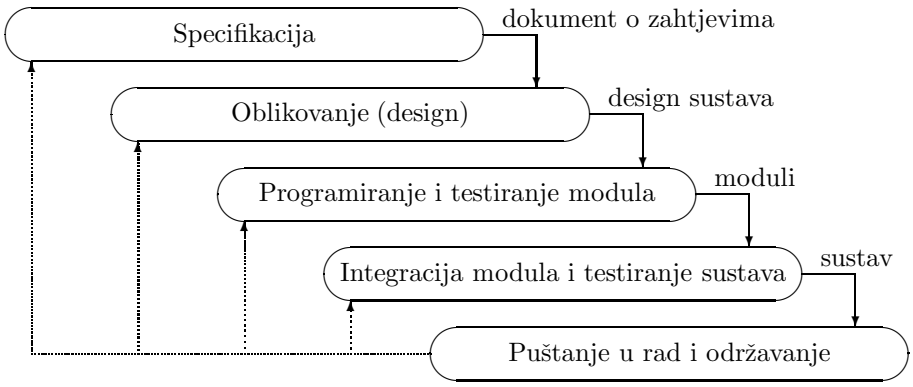
- *Specifikacija.* Analiziraju se zahtjevi korisnika. Utvrđuje se *što* softver treba raditi.
- *Oblikovanje* (engleski *design.*) Oblikuje se građa sustava, način rada komponenti, te sučelje između komponenti. Dakle projektira se rješenje koje određuje *kako* će softver raditi.

- *Implementacija (programiranje).* Oblikovano rješenje realizira se uz pomoć raspoloživih programskih jezika i alata.
- *Verifikacija i validacija.* Provjerava se da li softver radi prema specifikaciji, odnosno da li radi ono što korisnik želi. Obično se svodi na *testiranje*, mada postoje i druge tehnike.
- *Održavanje odnosno evolucija.* Nakon uvođenja u upotrebu, softver se dalje popravlja, mijenja i nadograđuje, u skladu s promijenjenim potrebama korisnika.

Modeli se razlikuju po načinu odvijanja i međusobnog povezivanja osnovnih aktivnosti.

1.2.1 Model vodopada

Model vodopada (engleski *waterfall model*) nastao je u ranim 70-tim godinama 20. stoljeća, kao neposredna analogija s procesima iz drugih inženjerskih struka (na primjer mostogradnja). Softverski proces je građen kao niz vremenski odvojenih aktivnosti, u skladu sa Slikom 1.1.



Slika 1.1: softverski proces prema modelu vodopada.

Prednosti modela vodopada su sljedeće.

- Model omogućuje lagano praćenje stanja u kojem se softverski proces nalazi.
- Model je dobro prihvaćen od managementa.

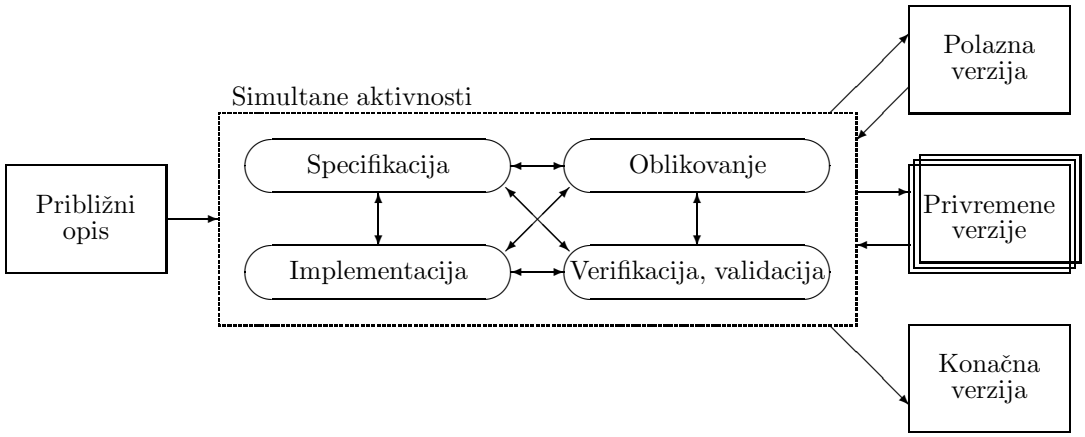
Mane modela vodopada su sljedeće.

- Navedene faze u praksi je teško razdvojiti, pa dolazi do naknadnog otkrivanja grešaka i vraćanja u prethodne faze.
- Zbog tendencije da se zbog poštovanja rokova u određenom trenutku "zamrzne" pojedina faza, može se desiti da je sustav u trenutku puštanja u rad već neažuran i zastario.

Model treba koristiti za velike sustave gdje postoje jasni zahtjevi.

1.2.2 Model evolucijskog razvoja

Model evolucijskog razvoja (engleski *evolutionary development*) nastao je kao protuteža modelu vodopada. Na osnovu približnog opisa problema razvija se polazna verzija sustava koja se pokazuje korisniku. Na osnovu korisnikovih primjedbi, ta polazna verzija se poboljšava, opet pokazuje, itd. Nakon dovoljnog broja iteracija dobiva se konačna verzija sustava. Unutar svake iteracije, osnovne aktivnosti se obavljaju simultano i ne daju se razdvojiti. Postupak je ilustriran Slikom 1.2.



Slika 1.2: evolucijski razvoj softvera.

Evolucijski razvoj čiji cilj je da se s korisnikom istraže zahtjevi te *zaista* proizvede konačni sustav zove se *istraživačko programiranje*. Ukoliko je jedini cilj istraživanje zahtjeva, tada je riječ o *prototipiranju* - vidi Odjeljak 2.3.

Prednosti modela evolucijskog razvoja su sljedeće.

- Model je u stanju proizvesti brzi odgovor na zahtjeve korisnika.
- Postoji mogućnost da se specifikacija postepeno razvija u skladu sa sve boljim korisnikovim razumijevanjem problema.

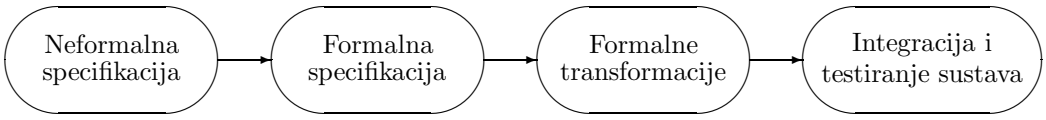
Mane modela evolucijskog razvoja su sljedeće.

- Proces nije transparentan za managere, naime oni ne mogu ocijeniti koliki dio posla je napravljen i kad će sustav biti gotov.
- Konačni sustav obično je loše strukturiran zbog stalnih promjena, te je nepogodan za održavanje.
- Zahtijevaju se posebni alati i natprosječni softverski inženjeri.

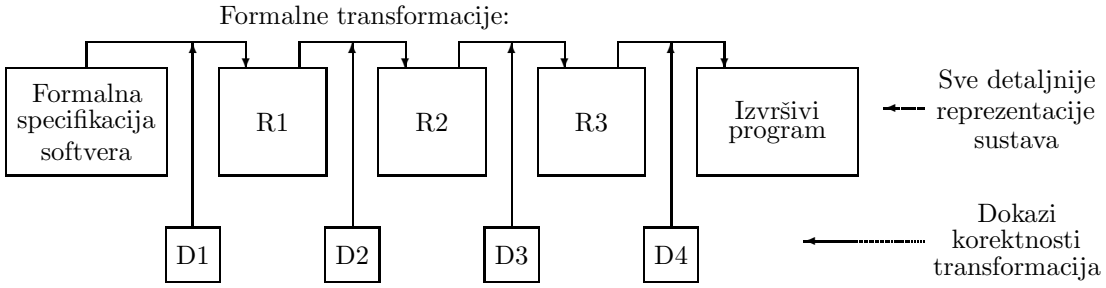
Model se uspješno koristi za razvoj web sjedišta, te za male sustave s kratkim životnim vijekom, pogotovo za sustave s nejasnim zahtjevima.

1.2.3 Model formalnog razvoja

Model formalnog razvoja (engleski *formal systems development*) zasniva se na korištenju takozvanih formalnih metoda za razvoj softvera. Zahtjevi se precizno definiraju na formalni način, korištenjem nedvosmislene matematičke notacije. Zatim se ta formalna specifikacija transformira do programa, nizom koraka koji čuvaju korektnost. Cijeli proces prikazan je u Slikom 1.3, dok je ideja formalnih transformacija detaljnije ilustrirana Slikom 1.4.



Slika 1.3: proces formalnog razvoja softvera.



Slika 1.4: formalne transformacije u sklopu formalnog razvoja softvera.

Prednosti modela formalnog razvoja su sljedeće.

- Nakon izrade formalne specifikacije, sve daljnje faze razvoja softvera mogle bi se automatizirati.
- Postoji "matematički" dokaz da je program točna implementacija polazne specifikacije; nema velike potrebe za testiranjem.

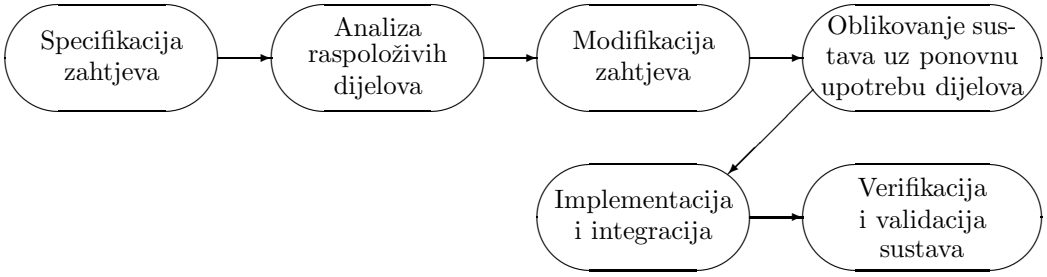
Mane modela formalnog razvoja su sljedeće.

- Izrada formalne specifikacije zahtijeva velik trud i znanje.
- Dokazi korektnosti transformacija postaju preglomazni za iole veće sustave.
- Korisnici ne mogu pratiti razvoj.
- Postojeći specifikacijski jezici nisu pogodni za interaktivne sustave.

Model se za sada relativno malo koristi u praksi, te se preporučuje jedino za sustave gdje se zahtijeva izuzetno velika pouzdanost i sigurnost. Moguće je da će se model znatno više koristiti u budućnosti ukoliko se razviju bolji specifikacijski jezici te bolji alati za automatske transformacije.

1.2.4 Model usmjeren na ponovnu upotrebu

Model usmjeren na ponovnu upotrebu (engleski *reuse-oriented development*) polazi od pretpostavke da već postoje gotove i upotrebljive softverske cjeline, kao što su *COTS sustavi* (engleski *Commercial Off-The-Shelf Systems*) ili dijelovi prije razvijenih sustava. Novi sustav nastoji se u što većoj mjeri realizirati spajanjem postojećih dijelova, u skladu sa Slikom 1.5. Ista ideja prisutna je i u drugim tehničkim disciplinama: novi mehanički ili elektronički proizvod nastoji se sklopiti od standardnih dijelova (vijaka, čipova, ...).



Slika 1.5: proces razvoja softvera uz ponovnu upotrebu.

Prednosti modela usmjerenog na ponovnu upotrebu su sljedeće.

- Smanjuje se količina softvera kojeg stvarno treba razviti, te se tako smanjuje vrijeme, trošak i rizik.
- Stavlja se oslonac na provjerene i dobro testirane dijelove softvera.

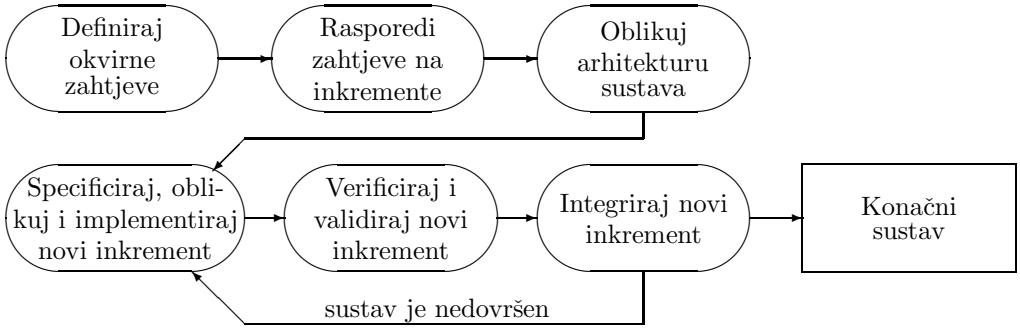
Mane modela usmjerenog na ponovnu upotrebu su sljedeće.

- Zbog kompromisa u specifikaciji moguće je da sustav neće u potpunosti odgovoriti stvarnim potrebama korisnika..
- Djelomično je izgubljena kontrola nad evolucijom sustava, budući da ne upravljamo razvojem novih verzija korištenih dijelova.

Očekuje se da će ovaj model ipak postati prevladavajući u 21. stoljeću, budući da je broj gotovih rješenja sve veći, a korisnici imaju sve manje vremena za čekanje rješenja.

1.2.5 Model inkrementalnog razvoja

Model inkrementalnog razvoja (engleski *incremental development*) može se shvatiti kao hibrid modela vodopada i modela evolucijskog razvoja. Sustav se opet razvija u nizu iteracija. No pojedina iteracija ne dotjeruje već realizirani dio sustava, već mu dodaje sasvim novi dio - inkrement. Razvoj jednog inkrementa unutar jedne iteracije odvija se po bilo kojem modelu - na primjer vodopad. Ideja je prikazana na Slici 1.6.



Slika 1.6: inkrementalni razvoj softvera.

Prednosti modela inkrementalnog razvoja su sljedeće.

- Proces je još uvijek prilično transparentan za managere, budući da je vidljivo do kojeg smo inkrementa došli.
- Korisnici ne moraju dugo čekati da bi dobili prvi inkrement koji zadovoljava njihove najpreče potrebe. Razvoj slijedi prioritete.

Mane modela inkrementalnog razvoja su sljedeće.

- Koji put je teško podijeliti korisničke zahtjeve u smislene inkremente.
- Budući da cjelokupni zahtjevi nisu dovoljno razrađeni na početku, teško je odrediti zajedničke module koji su potrebni raznim inkrementima i koji bi morali biti implementirani u prvom inkrementu.

Ipak, ovo je vrlo upotrebljiv model koji se intenzivno koristi u praksi, na primjer u varijanti “extreme programming”.

1.3 Upravljanje softverskim projektom

Ovaj kolegij bavi se *računarskim* i *tehničkim* aspektima softverskog inženjerstva. No podjednako važan je i *organizacijski (managerski)* aspekt, koji je tema posebnog kolegija, i o kojem ćemo ovdje dati samo uvodne napomene.

Upravljanje softverskim projektom (engleski *software project management*) potrebno je zato da bi se softver razvio na vrijeme i u okvirima planiranih troškova. Posao upravljanja softverskim projektom povjerava se *softverskom manageru.*

1.3.1 Osobine softverskog projekta

Softverski projekt se po sljedećim osobinama razlikuje od klasičnog tehničkog projekta kao što je gradnja mosta ili broda.

- *Produkt je "neopipljiv".* Teško je vidjeti rezultat. Teško je procijeniti koliki dio posla je obavljen. Manager se mora pouzdati u apstraktne dokumente.
- *Nema standardnog procesa.* Postoje razne metode i alati, no ne zna se što je najpogodnije u zadanim okolnostima.
- *Projekt je obično "neponovljiv".* Stara iskustva obično nisu primjenjiva. Pojavljuju se nepredviđeni problemi. Tehnologija se brzo mijenja.

Zbog ovih osobina, upravljanje softverskim projektom je izuzetno težak managerski zadatak, te zahtijeva izuzetno dobrog managera.

1.3.2 Poslovi softverskog managera

Poslovi softverskog managera među ostalim uključuju sljedeće:

- pisanje prijedloga projekta,
- procjenjivanje troškova projekta,
- planiranje i praćenje projekta,
- izbor i ocjenjivanje suradnika,
- pisanje izvještaja i prezentiranje.

1.3.3 Planiranje i praćenje projekta

Planiranje i praćenje projekta predstavlja redoviti, svakodnevni i najegzaktniji posao softverskog managera. Planiranje i praćenje uključuje sljedeće:

- definiranje projektnih aktivnosti, njihovog trajanja i međuzavisnosti,
- određivanje kalendara početka i završetka svake aktivnosti,
- raspoređivanje ljudi i drugih resursa na aktivnosti,
- praćenje izvršenja aktivnosti,
- povremena revizija svih parametara.

Za planiranje i praćenje projekta, manager koristi egzaktne modele i metode, poput mrežnih planova, analize kritičnih putova, Ganttovih dijagrama. Te metode i modeli implementirani su u odgovarajućim alatima za upravljanje projektima, na primjer Microsoft Project.

U prilozima 1.1 - 1.4 vidimo primjer projekta koji se sastoji od 12 aktivnosti sa zadanim trajanjima i međuovisnostima. Skup aktivnosti prikazan je kao mreža. Pronađene su kritične aktivnosti, dakle one koje se nalaze na najduljem putu između čvorova mreže koji odgovaraju početku odnosno kraju projekta. Manager treba posvetiti posebnu pažnju kritičnim aktivnostima, zato jer bi njihovo kašnjenje uzrokovalo kašnjenje cijelog projekta. Kalendar odvijanja aktivnosti najprije je utvrđen na mreži, a zatim je prikazan je kao Ganttov dijagram. Zatim su ljudi raspoređeni na aktivnosti, pa je nacrtan Ganttov dijagram zauzetosti ljudi.

Poglavlje 2

ZAHTJEVI I SPECIFIKACIJA

2.1 Općenito o zahtjevima i specifikaciji

Specifikacija je početna faza softverskog procesa. Analiziraju se *zahtjevi* na budući sustav. Rezultat je *dokument* o zahtjevima, koji opisuje *što* sustav treba raditi, po mogućnosti bez prejudiciranja kako da se to postigne. Specifikacijom se bavi tim sastavljen od razvijača softvera i budućih korisnika.

2.1.1 Vrste zahtjeva

Same zahtjeve dijelimo na dvije vrste:

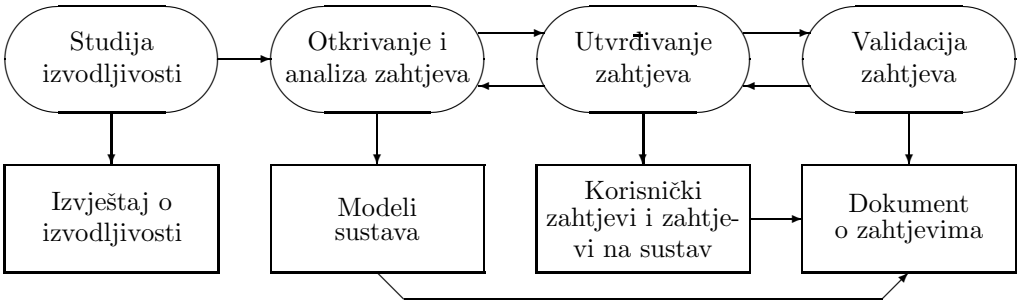
- *Funkcionalni zahtjevi.* Opisuju funkcije sustava, dakle usluge koje bi on trebao obavljati, izlaze koje on daje za zadane ulaze, te njegovo ponašanje u pojedinim situacijama.
- *Nefunkcionalni zahtjevi.* Izražavaju ograničenja na funkcije sustava, na primjer traženo vrijeme odziva, traženu razinu pouzdanosti, dozvoljeno zauzeće memorije, obavezu korištenja određenog programskog jezika, poštivanje određenih standarda.

2.1.2 Pod-aktivnosti unutar specifikacije

Aktivnost specifikacije detaljnije se može podijeliti u sljedeće podaktivnosti.

- *Studija izvodljivosti.* Procjenjuje se da li se uočene potrebe korisnika mogu zadovoljiti uz pomoć dostupnih hardverskih i softverskih tehnologija, da li bi predloženi sustav bio isplativ u poslovnom smislu, te da li sustav može biti razvijen s raspoloživim budžetom.
- *Otkrivanje i analiza zahtjeva.* Otkrivaju se zahtjevi, tako da se promatraju postojeći sustavi, analiziraju radni procesi, intervjuiraju budući korisnici i njihovi manageri, itd. Na taj način stvaraju se modeli sustava (vidi Odjeljak 2.2), a ponekad i prototipovi (vidi Odjeljak 2.3), sve u cilju boljeg razumijevanja sustava kojeg treba stvoriti.
- *Utvrđivanje zahtjeva.* Informacije skupljene analizom pretvaraju se u tekstove koji definiraju zahtjeve. Postoje dvije razine u opisivanju zahtjeva: "korisnički zahtjevi" i "zahtjevi na sustav".
- *Validacija zahtjeva.* Provjerava se da li su zahtjevi realistični (mogu se ostvariti raspoloživom tehnologijom i budžetom), konzistentni (nisu u međusobnom konfliktu), te potpuni (uključene su sve potrebne funkcije i ograničenja).

Način odvijanja odvijanja ovih pod-aktivnosti prikazan je na Slici 2.1.



Slika 2.1: pod-aktivnosti koje čine aktivnost specifikacije.

2.1.3 Vrste dokumenata koji opisuju zahtjeve

Rezultat specifikacije su dokumenti koji opisuju zahtjeve. Ti dokumenti razlikuju se po svom sadržaju, načinu opisivanja, te detaljnosti. Uvodimo sljedeće nazive za pojedine vrste dokumenata.

- *Korisnički zahtjevi.* To je manje precizan tekst u prirodnom jeziku, popraćen dijagramima i tablicama, opisuje funkcije sustava i ograničenja pod kojima sustav radi. Prilagođen je krajnjim korisnicima.
- *Zahtjevi na sustav.* Riječ je o detaljnom i preciznom opisu funkcija sustava i ograničenja. Može služiti kao temelj ugovoru između kupca i razvijača softvera. Služi softverskim inženjerima kao polazište za oblikovanje. Pisan je u donekle strukturiranom prirodnom jeziku.
- *Modeli sustava.* To su dijagrami koji opisuju ponašanje sustava na način koji je precizniji i jezgrovitiji od prirodnog jezika (vidi Odjeljak 2.2). Nastaju tijekom analize zahtjeva kao sredstvo komunikacije između razvijača softvera i budućih korisnika.
- *Dokument o zahtjevima.* Riječ je o konačnom rezultatu specifikacije, dobivenom kao unija svih prethodno opisanih dokumenata.
- *Specifikacija softverskog designa.* Ovaj dokument nije obavezan. Predstavlja apstraktni opis građe softvera i funkcija koje on obavlja, pisan u formalnom jeziku (vidi Odjeljak 2.4). Pretpostavlja određenu arhitekturu sustava, te na osnovu nje dalje razrađuje "zahtjeve na sustav". Eventualno se stvara kao most između aktivnosti specifikacije i oblikovanja. Služi razvijačima softvera kao polazište za daljnje oblikovanje i implementaciju.

U Prilozima 2.1 i 2.2 imamo najprije primjer razlike između korisničkih zahtjeva i zahtjeva na sustav, a zatim prijedlog strukture dokumenta o zahtjevima.

2.1.4 Problemi koji se pojavljuju kod specifikacije

Kod specifikacije se mogu pojaviti sljedeći problemi.

- Ukoliko novi sustav treba promijeniti sadašnji način rada, tada ga je teško definirati budući da s njim još nema iskustava.
- Razni korisnici imaju različite zahtjeve koji mogu biti u konfliktu. Konačni zahtjevi su nužno kompromis.
- Financijeri sustava i njegovi korisnici obično nisu isti ljudi. Kupac postavlja zahtjeve motivirane organizacijskim ili budžetskim ograničenjima koja su u konfliktu s korisničkim zahtjevima.
- Poslovno i tehničko okruženje se mijenja. Time se mijenjaju i zahtjevi.

Ukoliko se zahtjevi ne mogu jasno izraziti, pribjegava se prototipiranju (vidi poglevlje 2.3).

2.2 Modeliranje sustava

Tijekom otkrivanja i analize zahtjeva sastavljaju se *modeli* postojećeg ili budućeg sustava. Ti modeli opisuju sustav na grafički način pomoću dijagrama. Precizniji su i jezgrovitiji od ekvivalentnog opisa pomoću prirodnog jezika. Još uvijek su razumljivi korisnicima. Modeli također predstavljaju važan most između specifikacije i oblikovanja.

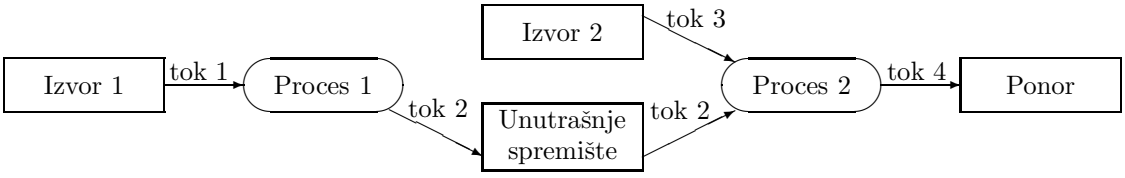
Model uvijek daje apstraktnu (pojednostavnjenu) sliku sustava. Također, on promatra sustav iz neke određene perspektive, pa ističe neke njegove osobine a zanemaruje druge. Najvažnije perspektive su:

- *Kontekst.* Određuje se granica između sustava i njegove okoline, te sučelja koja se uspostavljaju na toj granici.
- *Ponašanje.* Promatraju se transformacije podataka, reakcije sustava na događaje, te promjene njegovih stanja.
- *Struktura.* Modelira se arhitektura samog sustava ili građa podataka koje on obrađuje.

Da bi dobili cjelovitu sliku o sustavu, moramo imati nekoliko komplementarnih modela koji ga promatraju iz različitih perspektiva. Svaka metoda razvoja softvera propisuje određeni broj modela.

2.2.1 Model toka podataka

Model toka podataka (engleski *data flow model*) promatra sustav iz perspektive ponašanja, te opisuje transformacije (obradu) podataka. Sustav se modelira kao skup *procesa* (funkcija) koje transformiraju podatke, *tokova podataka* između tih procesa, te *spremišta* (ili izvora ili ponora) za podatke. Na dijagramu su procesi prikazani kao ovali, tokovi kao strelice, a spremišta kao pravokutnici, u skladu sa Slikom 2.2. Modeliranje toka podataka postalo je popularno nakon pojave knjige T. de Marco: "Structured Analysis and System Specification" - 1978, te je sastavni dio starijih funkcionalno-orijentiranih metoda.



Slika 2.2: grafički prikaz modela toka podataka.

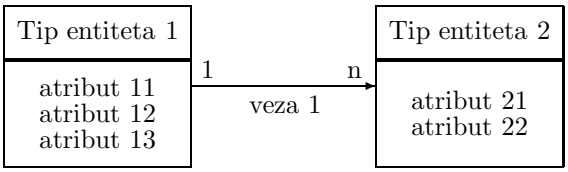
Kao primjer pogledajmo Prilog 2.3. Prikazano je kako se u nekoj organizaciji obrađuje narudžba za opremu (na primjer računalo). Dijagrami mogu biti složeni u hijerarhiju s obzirom na razinu apstrakcije. Tako na primjer Prilog 2.4 prikazuje veći postupak nabave i instaliranja opreme - mali postupak sa Priloga 2.3 ovdje je skupljen u jedan oval "Place equipment order".

Model toka podataka može služiti i za prikaz sustava iz perspektive konteksta: u tu svrhu cijeli sustav prikazujemo kao jedan proces okružen vanjskim izvorima i ponorima podataka. Druga mogućnost je da na složenijem dijagramu prikažemo i sustav i njegovu okolinu, te da sam sustav uokvirimo - vidi Prilog 2.4.

Model toka podataka ne objašnjava strukturu podataka u razmatranim tokovima. Također, on ništa ne govori o toku kontrole (kad se pokreće koji proces), ni o učestalosti obrade podataka. Nije čak ni jasno da li proces koji prima dva ulazna toka treba te tokove primati istovremeno, ili ih može obrađivati zasebno.

2.2.2 Model entiteta, veza i atributa

Model entiteta, veza i atributa (engleska kratica *ERA* model) promatra sustav iz perspektive strukture, te opisuje građu podataka. Može biti riječ o podacima u spremištima ili onima koji prolaze kroz sustav kao tokovi. Navode se *entiteti*, *veze* među entitetima, te *atributi* koji se pripisuju entitetu. Bilježi se i funkcionalnost veza (1:1, 1:n, m:n). Postoje različiti načini crtanja dijagrama. Mi ćemo "zloupotrebiti" UML notaciju, tako da entitet interpretiramo kao pojednostavnjenu klasu objekata koja ima atribute ali nema operacije - vidi Sliku 2.3.



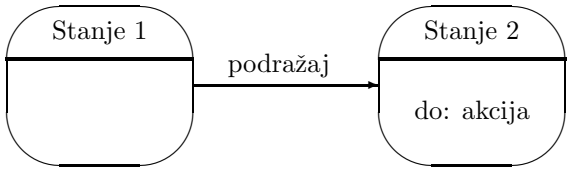
Slika 2.3: grafički prikaz modela entiteta, veza i atributa.

Model je bio predložen 1976. godine u radu P. Chen-a, te je kasnije bio proširivan od Codd-a, Hamer & McLead-a, te Hull & King-a u 80-tim godinama. Sastavni je dio starijih funkcionalno-orijentiranih metoda razvoja softvera. Na Prilogu 2.5 vidi se ERA model za dio baze podataka unutar knjižničnog informacijskog sustava.

2.2.3 Model promjene stanja

Model promjene stanja (engleski *state transition model* ili *state machine model*) spada u one koji promatraju sustav iz perspektive ponašanja. Opisuje se ponašanje sustava koji se u svakom trenutku nalazi u jednom od mogućih *stanja*. Određeni *podražaj* (događaj, ulaz) uzrokovat će prelazak u drugo stanje, i pritom će se izvesti neka *akcija*. Model u neku ruku prikazuje tok kontrole. Pritom on ne prikazuje tok podataka.

Postoje opet različiti načini crtanja dijagrama. Mi ćemo koristiti UML notaciju, koja zapravo služi za modeliranje ponašanja objekata. Stanja su prikazana kao ovali, a podražaji kao strelice. Akcija koja se izvodi dolaskom u neko stanje upisuje se u oval tog stanja iza riječi "do:". Sve se to vidi na Slici 2.4.



Slika 2.4: grafički prikaz modela promjene stanja.

Model promjene stanja inače se u matematici zove konačni automat. Pojavio se u funkcionalno-orijentiranim metodama koje su prilagođene razvoju sustava u realnom vremenu (Ward & Mellor, Harel - 80-te godine). Također je kasnije prihvaćen u objektno-orijentiranim metodama (RUP i UML).

Na Prilogu 2.6 nalazi se dijagram promjene stanja mikrovalne pećnice. Tabele u Prilogu 2.7 detaljnije objašnjavaju stanja i podražaje. Problem s modelom promjene stanja je da broj mogućih stanja može jako narasti. Tada se pribjegava uvođenju super-stanja (koja objedinjavaju nekoliko bliskih stanja) i stvaranju hijerarhije dijagrama. Prilog 2.8 profinjuje (super) stanje "Operation" iz Priloga 2.6.

2.2.4 Objektni modeli

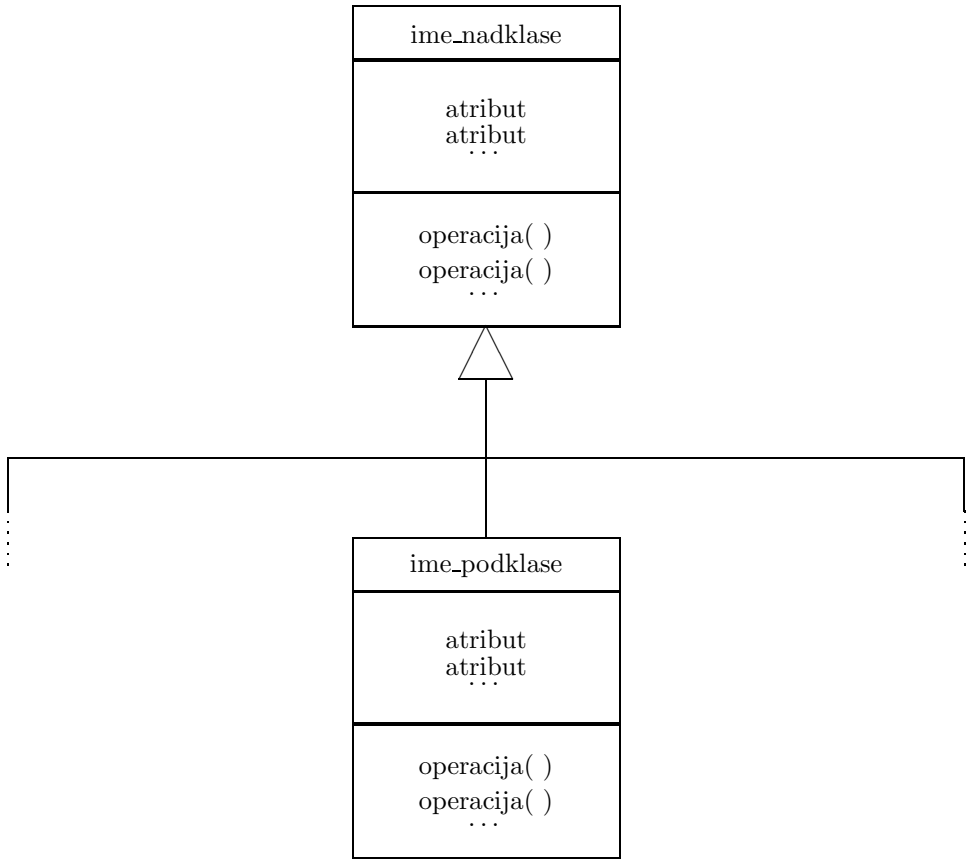
Objektni modeli (engleski *object models*) nastoje objediniti perspektive strukture i ponašanja. Opisuju sustav tako što identificiraju *klase objekata*, te odnose među tim klasama. Objekt i klasa su pojmovi koji su nam poznati iz objektno-orijentiranih programskih jezika (Java, C++). Objekt se sastoji od

atributa (podataka) i pripadnih operacija (metoda, funkcija). Klasa je skup istovrsnih objekata. Na taj način objektni modeli kombiniraju svojstva modela toka podataka odnosno modela entiteta, veza i atributa.

Objektni modeli pojavili su se u novijim objektno-orijentiranim metodama za razvoj softvera (Booch, Rumbaugh, Jacobson, Coad & Yourdon - rane 90-te godine 20. stoljeća). Prva trojica autora su krajem 90-tih u okviru zajedničke tvrtke Rational, uskiladili svoje pristupe tako da je nastala objedinjena metoda Rational Unified Process - RUP i standardna notacija (način crtanja dijagrama) Unified Modelling Language - UML.

Postoji nekoliko vrsta objektnih modela, koji se razlikuju po razmatranom tipu veze između klasa. U nastavku ćemo detaljnije opisati tri vrste.

Model nasljeđivanja pokazuje veze nasljeđivanja između klasa, tj. hijerarhiju nad-klasa i pod-klasa. Način crtanja dijagrama vidi se na Slici 2.5. Nasljeđivanje znači da pod-klasa ima sve atribute i operacije kao njena nad-klasa, te još neke svoje vlastite atribute i operacije. Dozvoljeno je i višestruko nasljeđivanje, što na razini specifikacije pravi manje problema nego u implemantaciji.

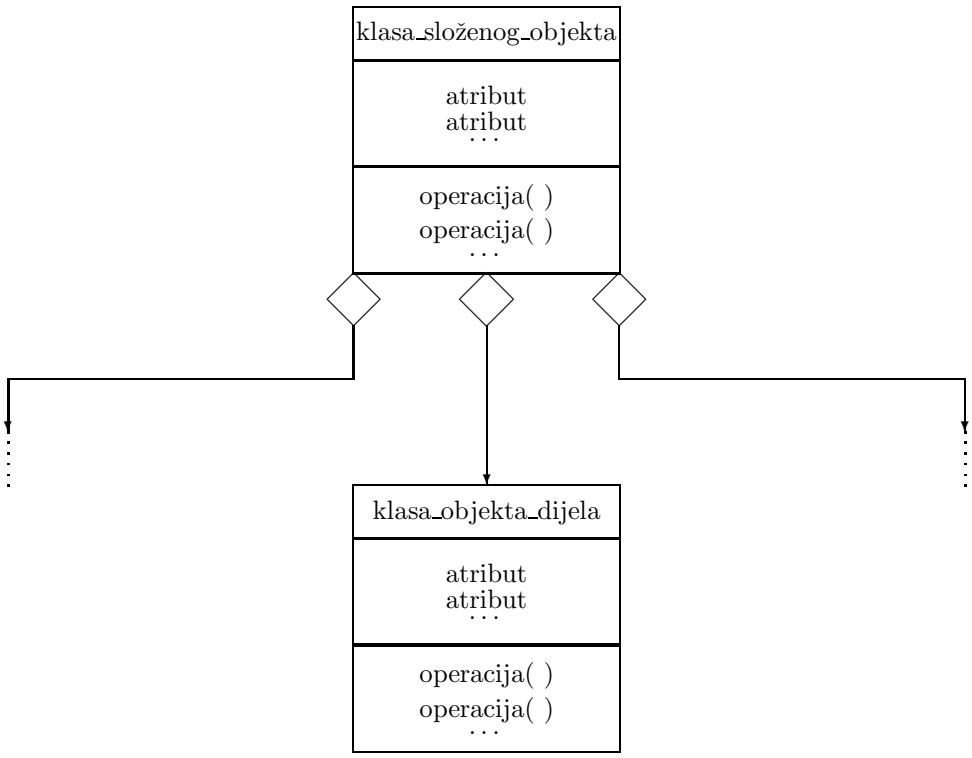


Slika 2.5: grafički prikaz modela nasljeđivanja.

Prilog 2.9 prikazuje hijerarhiju klasa za predmete pohranjene u knjižnici. Prilog 2.10 je hijerarhija klasa za korisnike knjižnice. Prilog 2.11 predstavlja primjer višestrukog nasljeđivanja.

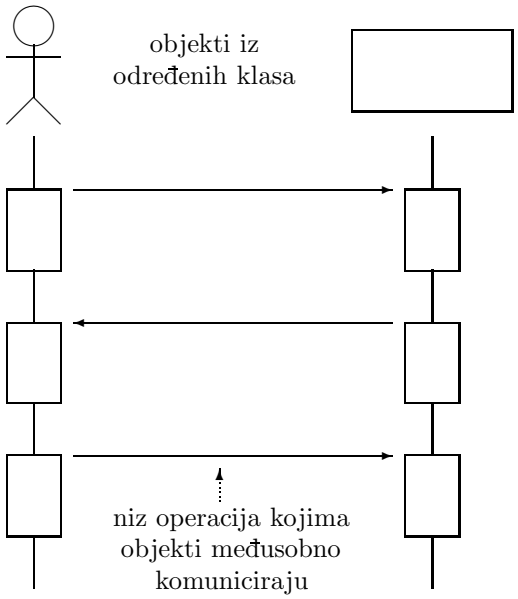
Model agregacije pokazuje veze uključivanja, tj. pokazuje koji jednostavniji objekti se pojavljuju kao sastavni dijelovi složenijeg objekta. Način crtanja dijagrama pokazan je na Slici 2.6.

Prilog 2.12 prikazuje klasu za složeni objekt - kolegij, koji se sastoji od jednostavnijih objekata - predavačevih bilješki, slajdova, zadataka za studente, videozapisa ....



Slika 2.6: grafički prikaz modela agregacije.

Model ponašanja objekata pokazuje veze između klasa koje se uspostavljaju time što objekti jedne klase pokreću operacije druge klase. Crtaju se sekvencijski dijagrami poput onog na Slici 2.7, gdje se vide mogući scenariji komunikacije između objekata.



Slika 2.7: grafički prikaz modela ponašanja objekata.

Prilog 2.13 prikazuje scenarij, gdje korisnik najprije pristupa katalogu knjižnice da vidi da li je određeni dokument dostupan u elektroničkom obliku, pa zatim traži da mu se taj elektronički dokument dostavi preko mreže. Zbog zaštite autorskih prava, od korisnika se zahtijeva da prihvati ugovor o licenciranju. Mrežni poslužitelj na kraju šalje dokument u komprimiranom obliku.

Objektni modeli iz naših primjera bili su nacrtani po pravilima UML. Ista UML notacija dalje se koristi u fazi oblikovanja. Objektne metode poput RUP ne prave jasnu razliku između specifikacije i oblikovanja.

2.2.5 Modeliranje pomoću CASE alata

Modeliranje sustava obično se obavlja uz pomoć pogodnog *upper-CASE alata* za specifikaciju i oblikovanje. Takav alat sadrži editore za sve potrebne dijagrame. Svi modeli spremaju se u zajednički repozitorij. Unutar repozitorija gradi se *rječnik podataka* - lista imena i pripadnih opisa za sve objekte, entitete, atribute, procese, tokove, operacije, ... koji su se pojavili u našim modelima. Preko rječnika podataka moguće je osigurati konzistenciju imena, dakle da isto ime svugdje označava istu stvar, a različita imena različite stvari.

2.3 Upotreba prototipova

Korisnici budućeg sustava često ne uspijevaju jasno izraziti svoje zahtjeve, budući da ne mogu predvidjeti kako će taj sustav utjecati na njihove radne navike, u kakvoj će interakciji biti s drugim sustavima, te u kojoj mjeri on može automatizirati radne procese. Tada se pribjegava izradi *prototipa*. Riječ je o jednostavnom i brzo razvijenom programu, koji oponaša budući sustav, lako se mijenja i nadograđuje, te omogućuje isprobavanje raznih ideja, koncepcija i opcija. Prototip podržava dvije pod-aktivnosti unutar aktivnosti specifikacije:

- *Otkrivanje zahtjeva.* Korisnici eksperimentiraju, pa tako otkrivaju svoje potrebe i dobivaju nove ideje o tome što bi sustav trebao raditi.
- *Validacija zahtjeva.* Uočavaju se greške i propusti u polaznim zahtjevima, na primjer nepotpuni ili nekonzistentni zahtjevi.

Osim toga, prototip može donijeti sljedeće dodatne koristi.

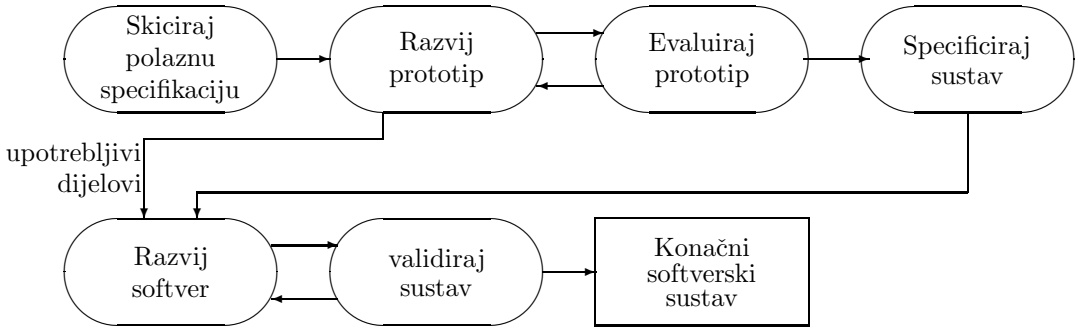
- Otkrivaju se nesporazumi između razvijača softvera i korisnika.
- Lakše se identificiraju komplicirane funkcije koje zahtijevaju pozornost.
- Managementu se demonstrira izvedivost i korisnost sustava.

2.3.1 Mjesto prototipiranja unutar softverskog procesa

U praksi se pojam prototipiranja često miješa s pojmom istraživačkog programiranja, dakle razvoja softvera po modelu evolucijskog razvoja. Razlike između ta dva pojma su sljedeće.

- Cilj istraživačkog programiranja je postepeno dotjerivanje "prototipa" u skladu sa zahtjevima koji se otkrivaju, sve dok se taj "prototip" ne pretvori u konačni sustav. Prioritet kod implementiranja zahtjeva imaju oni koji su najjasniji i korisnicima najpotrebniji.
- Cilj "pravog" prototipiranja je otkrivanje i validacija zahtjeva. Prioritet kod implementiranja zahtjeva imaju baš oni koji su nejasni i koje treba istražiti. Nakon utvrđivanja zahtjeva prototip se "baca", a sustav se dalje oblikuje i implementira na konvencionalni način.

U slučaju "pravog" prototipiranja, cjelokupni softverski proces izgleda kao na Slici 2.8. Pritom treba biti oprezan u izboru dijelova iz prototipa, budući da su oni često loše strukturirani, loše dokumentirani, imaju slabe performanse, slabu pouzdanost, te ne zadovoljavaju zahtjeve sigurnosti.



Slika 2.8: softverski proces s uključenim prototipiranjem.

2.3.2 Tehnike za prototipiranje

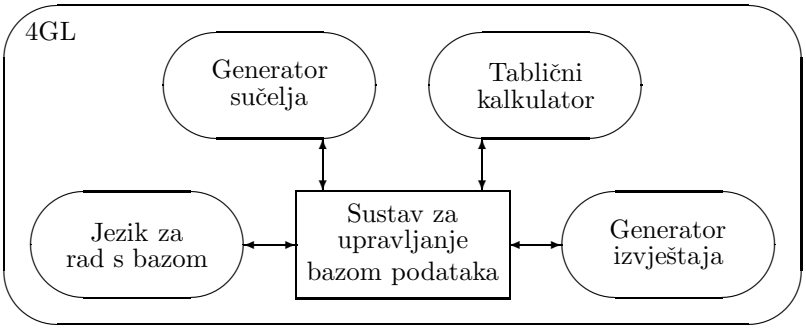
Da bi prototipiranje imalo smisla, sam prototip se mora stvoriti na brz, jeftin i fleksibilan način. Za to su nam potrebne odgovarajuće razvojne tehnike, koje možemo ovako klasificirati.

Razvoj u dinamičnim jezicima visoke razine. Riječ je o programskim jezicima koji su velikom dijelom interpretirani, te zahtijevaju glomaznu run-time okolinu. Programi se pišu brzo i jednostavno te se na dinamički način mijenjaju, makar su njihove performanse obično slabe zbog interpretiranja i run-time podrške. Primjeri takvih jezika vide se u Tablici 2.1.

Jezik	Tip	Područje primjene
Smalltalk	Objektno-orijentirani	Interaktivni sustavi
Java	Objektno-orijentirani	Interaktivni mrežni sustavi
Prolog	Zasnovan na logici	Automatsko zaključivanje
Lisp	Zasnovan na listama	Simbolička obrada podataka

Tablica 2.1: jezici visoke razine za prototipiranje.

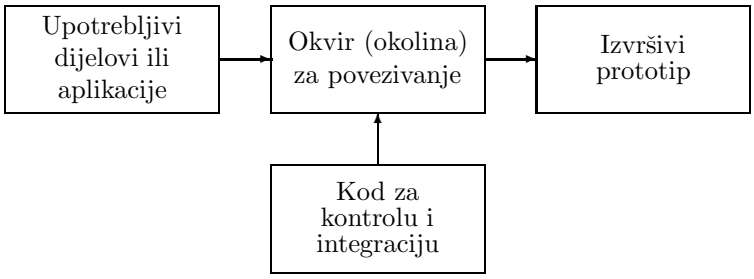
Programiranje za baze podataka. Svi komercijalni sustavi za upravljanje bazama podataka opskrbljeni su vlastitim alatima za izradu aplikacija koje obavljaju uobičajene radnje s podacima: ažuriranje, pretraživanje, izrada izvještaja. Takvi alati nazivaju se *jezici 4. generacije* (engleska kratica *4GL*) i sastoje se od dijelova koji se vide na Slici 2.9. Jezik za rad s bazom je u velikoj mjeri neproceduralan, te uključuje SQL. Generator sučelja omogućuje oblikovanje zaslonskih formulara (prozora) ili web stranica za unos ili prikaz podataka iz baze. Spreadsheet omogućuje analizu, manipulaciju i računanje s podacima. Generator izvještaja omogućuje oblikovanje štampanih izvještaja.



Slika 2.9: sastavni dijelovi jezika 4. generacije (4GL).

Primjeri 4GL-a mogu se naći unutar baza Oracle, DB2 ili Informix. Izrada aplikacija (ustvari prototipova) pomoću 4GL je brza i jednostavna, te ne zahtijeva puno programiranja. No dobivena aplikacija je prilično spora i rastrošna. Daljnja mana 4GL je da su oni nestandardni i neprenošljivi s jedne baze na drugu.

Povezivanje gotovih dijelova ili aplikacija. Prototip se slaže od već gotovih dijelova. U tu svrhu moramo imati na raspolaganju dovoljan broj upotrebljivih gotovih dijelova, te okvir ili mehanizam za povezivanje tih dijelova. Također moramo biti spremni na određene kompromise u specifikaciji samog prototipa. Opća shema za povezivanje dijelova ili aplikacija izgleda kao na Slici 2.10.



Slika 2.10: povezivanje gotovih dijelova softvera u prototip.

Navest ćemo nekoliko primjera za ovakvu tehniku prototipiranja.

- *Microsoft OLE* (engleski *Object Linking and Embedding*). Okvir za povezivanje je "compound document" koji u sebi sadrži razne "objekte", na primjer tekst, sliku, zvučni zapis, proračunsku tablicu. Kad korisnik klikne na objekt, aktivira se odgovarajuća aplikacija koja obrađuje dotičnu vrtu objekata, na primjer Word, Excel, Media Player, ..., itd.
- *Microsoft Visual Basic*. Prototip se razvija "vizualnim" programiranjem. Okvir za povezivanje je prozor u kojeg se interaktivno ubacuju gotove komponente grafičkog sučelja poput tekstualnih polja, izbornika, gumba, itd. Uz pojedinu komponentu veže se funkcija pisana u Visual Basicu koja će se pokrenuti kad korisnik nešto napravi s tom komponentom.
- *Skriptni jezici* poput Shell, Perl, TCL/TK ili Python. Okvir za povezivanje je sama skripta: njene naredbe mogu biti komande kojima se pozivaju gotove aplikacije ili uključuju komponente grafičkog sučelja.

2.3.3 Prototipiranje korisničkog sučelja

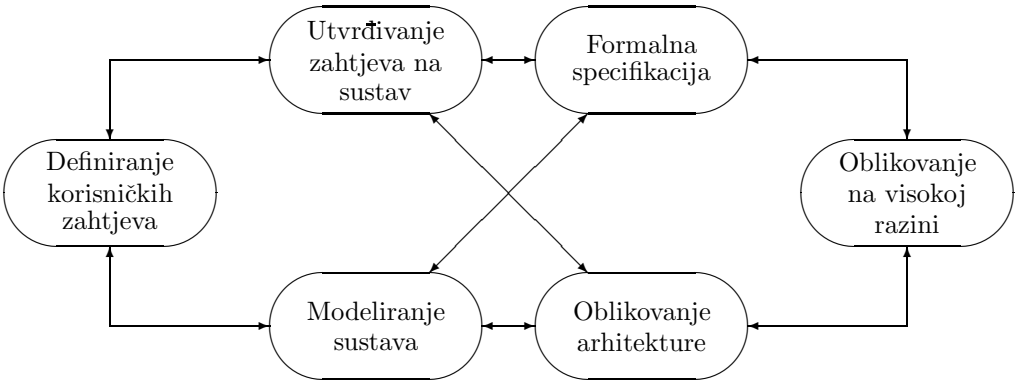
Pokazalo se da je prototipiranje vrlo pogodna metoda za specificiranje korisničkog sučelja. Naime, zahtjeve na sučelje teško je opisati riječima, pogotovo kad je riječ o grafičkom sučelju. Do pogodnog sučelja dolazi se eksperimentiranjem. Dovoljan je prototip koji implementira samo sučelje bez odgovarajuće funkcionalnosti - tzv. "čarobnjak iz Oza". Pogodni alati su: jezici 4. generacije, Visual Basic, TCL/TK, alati za web stranice, i drugi.

2.4 Formalna specifikacija

Formalna specifikacija omogućuje da se zahtjevi zapišu na sasvim precizan i nedvosmislen način. Umjesto prirodnog jezika koristi se jezik čiji rječnik, sintaksa i semantika su formalno definirani. Taj jezik zasnovan je na matematičkim pojmovima, obično iz teorije skupova, algebre ili logike.

2.4.1 Uloga, prednosti i mane formalne specifikacije

Formalna specifikacija je neophodna ukoliko se služimo modelom formalnog razvoja softvera (vidi Odjeljak 1.2). Kod drugih modela softverskog procesa ona nije nužna, no može se koristiti ukoliko želimo razviti najpreciznu vrstu opisa zahtjeva, a to je specifikacija softverskog designa (vidi Odjeljak 2.1). U tom drugom slučaju, mjesto formalne specifikacije je na granici između specifikacije i oblikovanja, prema Slici 2.11. Rezultati formalne specifikacije prvenstveno su namijenjeni razvijačima softvera, kao polazište za daljnje oblikovanje, implementaciju i verifikaciju.



Slika 2.11: mjesto formalne specifikacije u softverskom procesu.

Prednosti formalne specifikacije su sljedeće.

- Bolji uvid u zahtjeve, otklanjanje nesporazuma, smanjenje mogućnosti greške (što sve pridonosi pouzdanosti softvera).
- Mogućnost analiziranja zahtjeva matematičkim metodama (potpunost, konzistentnost).
- Može služiti kao podloga za formalnu verifikaciju implementiranog sustava.
- Mogućnost "animacije" specifikacije u svrhu prototipiranja.

Mane formalne specifikacije su sljedeće.

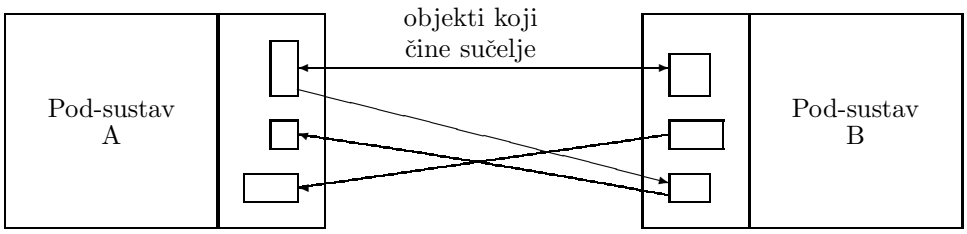
- Nerazumljiva je korisnicima i managementu.
- Zahtijeva posebno osposobljene softverske inženjere.
- Nije pogodna za interaktivne sustave ili grafička sučelja.
- Ne da se dobro skalirati, naime za imalo veći sustav količina posla je prevelika.

U skladu s ovim prednostima i manama, formalne metode se za sada koriste jedino za razvoj manjih dijelova pojedinih sustava, tamo gdje se zahtijeva izuzetno velika pouzdanost i sigurnost.

2.4.2 Formalna specifikacija sučelja

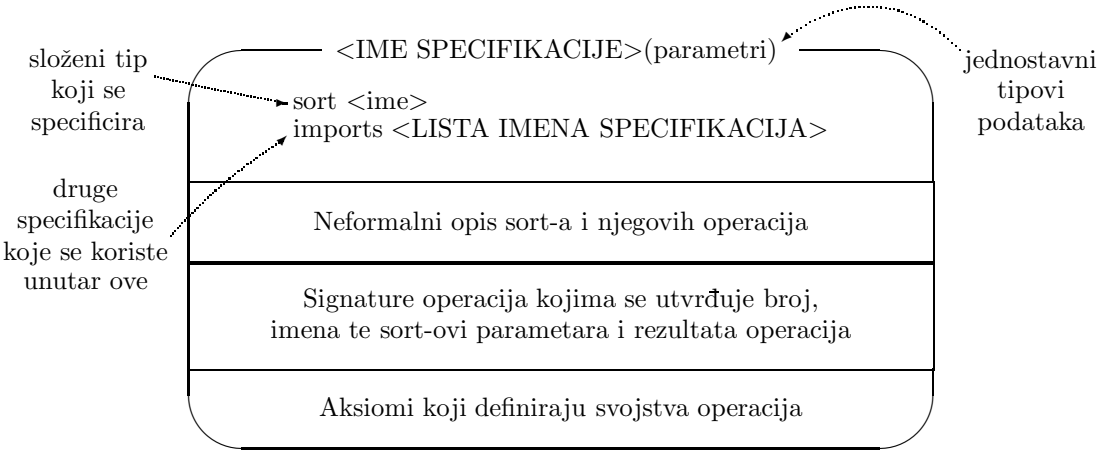
Veliki sustavi sastoje se od pod-sustava koji su u međusobnoj interakciji. Važan dio specifikacije sustava je definiranje sučelja pojedinih pod-sustava. Nakon što su ta sučelja utvrđena, svaki pod-sustav može se dalje razvijati neovisno od drugih.

Sučelje pod-sustava obično se definira kao skup apstraktnih tipova podataka ili objekata, kao što je ilustrirano Slikom 2.12. Dakle, definiraju se podaci i operacije kojima se može pristupiti izvana. Specifikacija sučelja svodi se na specifikaciju svakog od korištenih apstraktnih tipova podataka.



Slika 2.12: sučelje podsustava prikazano pomoću objekata.

U našim primjerima koristit ćemo *algebarski* specifikacijski jezik LARCH (Guttag i drugi, 1993). Specifikacija jednog apstraktnog tipa izgleda kao na Slici 2.13.



Slika 2.13: općenita struktura algebarske specifikacije.

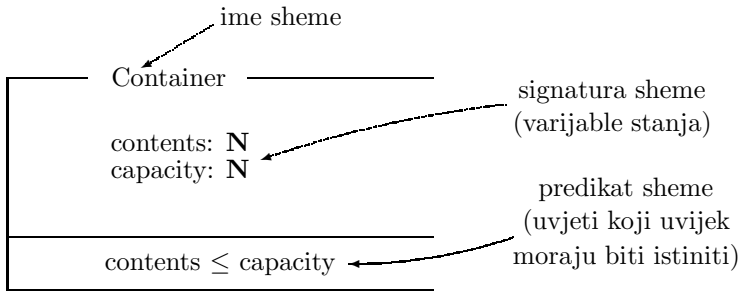
Prilog 2.14 sadrži specifikaciju apstraktnog tipa podataka LIST (kao u kolegiju Strukture podataka i algoritmi). Specifikacija je parametrizirana tipom elemenata koji se ubacuju u listu. Aksiomi zaista osiguravaju da operacije s listom imaju očekivana svojstva. Na primjer, korištenjem aksioma moguće je dokazati da je `Tail([5,7,9]) = [7,9]`.

Prilog 2.15 sadrži specifikaciju nadziranog zračnog sektora u kontroli zračnog prometa. Unutar sektora nalaze se avioni od kojih svaki ima drukčiji identifikator `Call-sign`, te se nalazi na drugoj visini `Height`. Pojavljuju se operacije kojima avion ulazi ili izlazi iz sektora, ili mijenja visinu, ili pak čitamo njegovu visinu. Aksiomi opet osiguravaju da operacije imaju očekivana svojstva. Na primjer, operacija `Move` mijenja visinu aviona jedino ako je nova visina slobodna, te je tada učinak promjene isti kao da je avion izašao iz sektora i zatim se vratio na novoj visini.

2.4.3 Formalna specifikacija ponašanja

Algebarske tehnike iz prethodnog odjeljka postaju vrlo nespretne ukoliko rezultat operacije ovisi o stanju objekta na kojeg se ta operacija primjenjuje. Tada je bolje koristiti formalnu specifikaciju koja se zasniva na matematičkom modelu stanja sustava. Operacije se specificiraju tako da se definira kako one mijenjaju stanje sustava. Na taj način specificira se ponašanje sustava. Najpoznatiji *modelni* specifikacijski jezik je *Z* (čitaj "zed" - Spivey, 1992). Specifikacija u Z-u sastoji se od malih komadića koji se zovu *scheme*. Opći oblik vidi se na Slici 2.14.

U prilozima imamo primjer djelomične specifikacije sustava s insulinskom pumpom. Sustav u pravilnim vremenskim intervalima mjeri razinu glukoze u krvi pacijenta-dijabetičara, te u slučaju porasta glukoze automatski uštrcava u krv potrebnu količinu insulina, čime bi se razina glukoze trebala opet normalizirati. Prilog 2.16 prikazuje arhitekturu sustava - osnovni podsustavi su: spremnik insulina, igla za uštrcavanje, pumpa koja prebacuje insulin iz spremnika u iglu, senzor koji mjeri razinu insulina u krvi, kontroler koji upravlja sustavom, sat, alarm i dva displeja.



Slika 2.14: općenita struktura Z-scheme.

U Prilogu 2.17 vidi se osnovna Z-shema `Insulin_Pump`, koja modelira stanje sustava. Varijable stanja su uglavnom cijeli brojevi, ili Boolean ili stringovi. Ulazne varijable imaju ime s "?", izlazne s "!", a ostale su interne.

- `reading?` ... izmjerena vrijednost glukoze u krvi,
- `dose, cumulative_dose` ... količina insulina koja će upravo biti uštrcana, te ukupna količina uštrcana u nekom proteklom razdoblju,
- `r0, r1, r2` ... tri prethodno izmjerene vrijednosti glukoze, služe za procjenu stupnja promjene,
- `capacity` ... kapacitet spremnika insulina,
- `alarm!` ... označava da li je uključen alarm,
- `pump!` ... kontrolni signal koji se šalje pumpi,
- `display1` ... poruka koja se ispisuje na prvom displeju
- `display2` ... brojčana vrijednost uštrcane doze insulina koja se ispisuje na drugom displeju.

Shema `DOSAGE` u Prilogu 2.18 računa dozu insulina koju treba uštrcati. "Importiraju" se sve varijable i predikati iz prethodne sheme. Znak $\triangle$ znači da `DOSAGE` predstavlja operaciju koja će mijenjati varijable iz `Insulin_Pump`. Imena varijabli bez znaka ' označavaju vrijednosti prije izvršavanja operacije, a imena s ' označavaju promijenjene vrijednosti nakon operacije.

Sheme `DISPLAY` i `ALARM` u Prilogu 2.19 predstavljaju operaciju ispisa poruka na displejeve, odnosno operaciju uključivanja ili isključivanja alarma.

Prikazana specifikacija sa četiri Z-sheme je nepotpuna, jer na primjer fali modeliranje ponašanja sustava u vremenu. Naime trebalo bi još modelirati događaj mjerenja glukoze svakih 10 minuta, te događaj slanja signala pumpi. Također, modelirani tajming trebao bi riješiti mogući konflikt u upotrebi displeja 1, koji ponekad opisuje stanje spremnika, a ponekad razinu glukoze u krvi.

Poglavlje 3

OBLIKOVANJE I IMPLEMENTACIJA

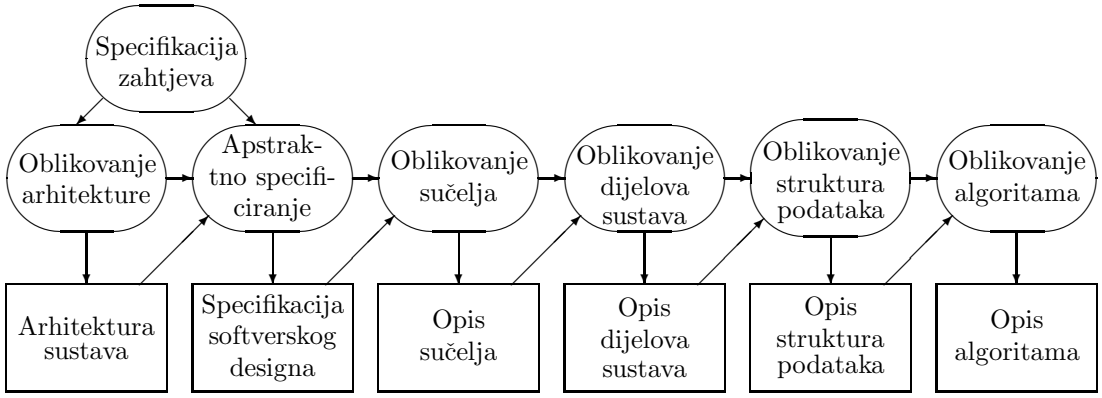
3.1 Općenito o oblikovanju i implementaciji

Nakon što je specifikacija ustanovila što softverski sustav treba raditi, *oblikovanje* treba utvrditi *kako* će sustav raditi. Rezultat oblikovanja je *design* sustava: precizni opis građe sustava, dijelova od kojih se on sastoji, sučelja između dijelova, korisničkog sučelja, te eventualno struktura podataka i algoritama koji se koriste. Oblikovanje je iterativni postupak, tj. do designa se dolazi postepenim profinjavanjem i razradom kroz više iteracija.

Nakon oblikovanja slijedi *implementacija*, tj. svi dijelovi predviđeni designom realiziraju se uz pomoć pogodnog programskog jezika. Zadnje iteracije oblikovanja obično se preklapaju s implementacijom. Također, implementacija se donekle preklapa s verifikacijom (testiranjem), budući da se programeri odmah bave debuggiranjem.

3.1.1 Pod-aktivnosti unutar oblikovanja

Oblikovanje je kreativna aktivnost koju je teško precizno opisati. Ipak, možemo reći da se oblikovanje sastoji od 6 pod-aktivnosti u skladu sa Slikom 3.1. Svaka pod-aktivnost daje svoje izlazne rezultate.



Slika 3.1: proces i rezultati oblikovanja.

U oblikovanju arhitekture uočavaju se i dokumentiraju pod-sustavi koji čine sustav, te veze među tim pod-sustavima (vidi Odjeljak 3.2). Unutar apstraktnog specificiranja, za svaki pod-sustav piše se specifikacija softverskog designa, dakle apstraktni i vrlo precizni opis zahtjeva. U sklopu oblikovanja sučelja, za svaki pod-sustav se oblikuje i dokumentira sučelje prema drugim pod-sustavima i prema

korisniku. Oblikovanje dijelova obavlja se tako da se svaki pod-sustav dalje rastavlja na sastavne dijelove: opisuje se svaki dio zasebno, a veći dijelovi se dalje rastavljaju na manje. U oblikovanju struktura podataka ili algoritama detaljno se oblikuju i opisuju složenije strukture podataka odnosno algoritmi koji se pojavljuju u sustavu.

3.1.2 Dekompozicija većih dijelova sustava u manje

Oblikovanje se u velikoj mjeri svodi na dekompoziciju većih dijelova u manje. Najprije kod oblikovanja arhitekture, cijeli sustav rastavljamo na pod-sustave. Kasnije kod oblikovanja dijelova, svaki pod-sustav dalje rastavljamo na dijelove, pa te dijelove na još manje dijelove, itd. Dekompozicija se obično obavlja u skladu s jednim od sljedeća dva pristupa oblikovanju:

- *Funkcionalni pristup.* Sustav se oblikuje sa stanovišta funkcija (procesa). Znači, dijelovi su moduli i pojedine funkcije. Budući da se kreće od procesa na najvišoj razini, pa ih se postepeno razgrađuje u sve manje procese, dobiva se hijerarhijski sustav kojeg je lako implementirati u klasičnim programskim jezicima (glavna funkcija, pod-funkcija, pod-pod-funkcija itd). Stanje sustava prikazano je globalnim podacima koji su dostupni raznim funkcijama. Ovakav pristup koristi se u klasičnim funkcionalno-orijentiranim metodama za razvoj softvera, a bit će također ilustriran na vježbama.
- *Objektni pristup.* Sustav se oblikuje kao skup objekata koji međusobno komuniciraju. Znači, dijelovi su objekti odnosno klase. Građa sustava obično nije hijerarhijska, a može se promatrati iz perpektive nasljeđivanja, agregacije, odnosno korištenja operacija među klasama. Sustav se izravno može implementirati u objektno-orijentiranim programskim jezicima. Nema globalnih podataka, a stanje sustava je zbroj stanja pojedinih objekata. Ovakav pristup koristi se u objektno-orijentiranim metodama za razvoj softvera, i bit će detaljnije opisan u Odjeljku 3.4.

3.1.3 Modeli sustava koji se koriste u oblikovanju

Modeli stvoreni tijekom specifikacije dalje se profinjavaju tijekom oblikovanja. Na taj način modeli predstavljaju važnu sponu između specifikacije i oblikovanja. Dok u specifikaciji često modeliramo aplikacijsku domenu ili stari način rada, u oblikovanju bi trebali modelirati budući sustav.

Kod funkcionalnog pristupa oblikovanju, polazište je model toka podataka. Procesi iz tog modela shvaćaju se kao pod-sustavi ili moduli koje treba realizirati pomoću funkcija. Daljnjim profinjavanjem dobivamo tzv. *strukturalni model* sustava, koji pokazuje kako se pojedina funkcija realizira pozivanjem podređenih funkcija. Strukturalni model crta se kao dijagram, gdje pravokutnici prikazuju funkcije (nadređena je iznad podređene), a spojnice označavaju pozive. U slučaju objektnog pristupa oblikovanju, polazište su objektni modeli (nasljeđivanje, agregacija, ponašanje). U te modele dodaju se novi objekti koji su na primjer potrebni za unutrašnje funkcioniranje kompliciranih objekata ili služe za razradu korisničkog sučelja. Također, oblikovanjem se do kraja definira sučelje svakog objekta.

Prilozi 3.1 i 3.2 predstavljaju objektni model odnosno model toka podataka za slične sustave koji obrađuju fakture. Objektni model je nacrtan u skladu s UML notacijom - crtkane strelice znače da objekti jedne klase koriste operacije druge klase. Prilog 3.3 sadrži primjer strukturalnog modela.

3.1.4 Utjecaj oblikovanja na kvalitetu softvera

Očito je da design softvera bitno utječe na njegovu kvalitetu. Postavlja se pitanje: kako treba oblikovati softver da bi on bio kvalitetan? Odgovor ovisi o tome koji od atributa kvalitete softvera nam je najvažniji.

Na primjer, ako nam je prvenstveno stalo do *mogućnosti održavanja*, tada je dobro da se design sastoji od velikog broja malih dijelova. Unutar pojedinog dijela treba postojati jaka "kohezija" njegovih još manjih dijelova, a veze između osnovnih dijelova trebaju biti relativno "labave". Takav design dobiva se primjenom objektnog pristupa.

S druge strane, ako nam je prvenstveno stalo do *efikasnosti* (performansi), tada je dobro da se design sastoji od malog broja relativno velikih dijelova, tako da se smanji komuniciranje između tih dijelova. Poželjno je da razne funkcije direktno pristupaju podacima, što stvara potrebu za globalnim podacima. Takav design moguće je dobiti primjenom funkcionalnog pristupa.

3.1.5 CASE alati za oblikovanje i implementaciju

Budući da se u oblikovanju koriste slični modeli sustava kao u specifikaciji, podršku za oblikovanje daju isti upper-CASE alati koje smo već spomenuli kad smo govorili o specifikaciji. S druge strane, za implementaciju te kasnije testiranje i debuggiranje znatno su pogodniji tzv. lower-CASE alati, poput Microsoft Visual Studio. Tipični lower-CASE alat sadrži editor za pisanje izvornog programskog koda, compiler, linker, biblioteku standardnih funkcija ili klasa, debugger, statički odnosno dinamički analizator koda, itd.

3.2 Oblikovanje arhitekture sustava

Oblikovanje arhitekture sustava predstavlja početnu fazu oblikovanja. Sustav se dekomponira na nekoliko pod-sustava, od kojih svaki obavlja određeni skup zadataka. Okvirno se određuje način komuniciranja pod-sustava, te način njihove kontrole.

Oblikovanje arhitekture obično dolazi prije detaljne specifikacije softverskog designa. Idealno, specifikacija ne bi smjela sadržavati informacije o designu. No u praksi je to nemoguće postići, osim za vrlo male sustave. Oblikovanje arhitekture je nužno da bi se bolje strukturirala sama specifikacija.

3.2.1 Pod-sustavi i odnosi među njima

Pod-sustav definiramo kao relativno samostalnu cjelinu, čije funkcioniranje uglavnom ne ovisi o servisima koje pružaju drugi pod-sustavi. Pod-sustav je sastavljen od svojih dijelova (moduli, objekti, ...). On ima definirano sučelje za komunikaciju s drugim pod-sustavima, te se podvrgava određenim režimima kontrole.

Da bi oblikovali arhitekturu nekog sustava, nije dovoljno samo definirati njegove pod-sustave, već također treba odrediti odnose između tih pod-sustava. Primijetimo da postoje dvije bitno različite vrste odnosa pod-sustava: međusobna komunikacija, odnosno međusobna kontrola. Zato se u sklopu oblikovanja arhitekture pojavljuju sljedeće dvije pod-aktivnosti.

- *Strukturiranje sustava*. Sustav se rastavlja na pod-sustave. Utvrđuje se komunikacija između pod-sustava, dakle razmjena podataka.
- *Modeliranje kontrole*. Određuje se tok kontrole između pod-sustava, dakle tko s kime upravlja.

Svaku od ovih pod-aktivnosti detaljnije ćemo opisati u zasebnom pod-odjeljku.

3.2.2 Strukturiranje sustava

Rezultat strukturiranja predočuje se kao blok dijagram, gdje su blokovi pod-sustavi, a strelice označavaju komunikaciju. Pojedini blok može sadržavati pod-blokove.

Prilog 3.4 je blok dijagram sustava za pakiranje uz pomoć robota. Predmeti stižu na tekućoj traci, robot ih uzima, prepoznaje, pakira na odgovarajući način, te zapakirane stavlja na drugu tekuću traku. Prilozi 3.5 i 3.6 sadrže blok dijagrame za još dva sustava: alarm protiv provalnika, odnosno kontrola leta.

Primjeri u prilozima pokazuju kako se za svaki sustav može razviti specifična struktura. No postoje i uobičajeni modeli strukture koji su se pokazali upotrebljivi za razne sustave. U nastavku ćemo opisati tri takva modela.

Model s repozitorijem. Svi zajednički podaci čuvaju se u središnjoj bazi podataka (repozitoriju), kojem mogu pristupiti svi pod-sustavi. Komunikacija između pod-sustava odvija se tako da jedan pod-sustav zapiše podatke u bazu, a drugi pod-sustav pročita te podatke. Sam repozitorij također treba shvatiti kao jedan (specijalizirani) pod-sustav. Struktura je vidljiva na Slici 3.2.

Slika 3.2: model za strukturu sustava s repozitorijem.

Ovo je uobičajena struktura za sustave koji rade s velikim količinama podataka (informacijski sustav banke ili poduzeća, CAD/CAM sustav, CASE alat). Konkretan primjer je CASE alat Select Yourdon kojeg koristimo na vježbama, ili informacijski sustav ISVU. Prilog 3.7 prikazuje zamišljeni CASE alat.

Prednosti modela s repozitorijem:

- pojednostavnjena je komunikacija jer je izbjegnuto direktno slanje podataka iz pod-sustava u pod-sustav;
- administriranje podataka je centralizirano pa se pod-sustavi ne trebaju time baviti;
- jednostavno je dodati novi pod-sustav, on samo mora biti kompatibilan s utvrđenim modelom podataka.

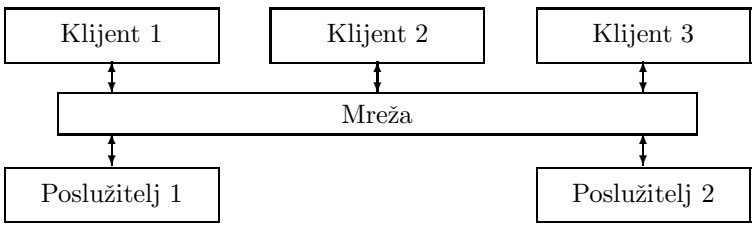
Mane modela s repozitorijem:

- podsustavi se moraju pokoriti zajedničkom modelu podataka, što može loše utjecati na performanse;
- različiti pod-sustavi mogu imati različite zahtjeve u pogledu administriranja podataka, no model ih prisiljava na zajedničku "politiku";
- teško je distribuirati repozitorij na više umreženih strojeva, pa repozitorij postaje usko grlo.

Model klijent-poslužitelj. Sustav se sastoji od sljedećih dijelova:

- *poslužitelji:* pod-sustavi koji nude usluge drugim pod-sustavima,
- *klijenti:* pod-sustavi koji traže usluge ponuđene od poslužitelja,
- *mreža:* omogućuje komunikaciju klijenata i poslužitelja.

Klijent mora znati imena dostupnih poslužitelja i njihove usluge. Poslužitelj ne mora znati ni imena ni broj klijenata. Struktura je vidljiva na Slici 3.3.



Slika 3.3: model za strukturu sustava klijent-poslužitelj.

Ovo je uobičajena struktura za mrežno-orijentirane sustave. Dobar primjer je world-wide-web, gdje su poslužitelji web serveri poput Apache, klijenti su web browseri poput Microsoft Internet Explorer, a mreža je Internet. Prilog 3.8 prikazuje zamišljenu multimedijsku biblioteku.

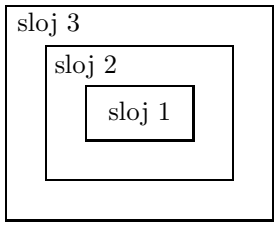
Prednosti modela klijent-poslužitelj:

- omogućuje distribuciju sustava na mrežu računala;
- omogućuje lagano dodavanje novog poslužitelja ili poboljšanje rada postojećeg;
- dopušta da svaki poslužitelj ima svoj optimizirani model podataka.

Mane modela klijent-poslužitelj:

- svaki poslužitelj mora se sam brinuti za administriranje svojih podataka;
- svaki klijent mora znati koje usluge se nalaze na kojem poslužitelju.

Model apstraktnog stroja (slojeviti model). Sustav je organiziran kao niz pod-sustava koji se zovu *slojevi*. Svaki sloj definira "apstraktni stroj" čiji strojni jezik (ustvari skup usluga) služi za implementaciju idućeg sloja. Struktura je ilustrirana Slikom 3.4.



Slika 3.4: slojeviti model za strukturu sustava.

Ovakva struktura pojavljuje se kod sustava gdje želimo ostvariti neovisnost rješenja o hardverskoj platformi. Primjeri su: Java, Motif/X-Toolkit/Xlib, OSI model za mrežne protokole sa 7 slojeva. Prilog 3.9 prikazuje zamišljeni sustav za upravljanje verzijama objekata.

Prednosti slojevitog modela:

- podržan je inkrementalni razvoj sustava;
- bilo koji sloj može se promijeniti ukoliko zadrži isto sučelje;
- ako se promijeni sučelje jednog sloja, pogođeni su samo susjedni slojevi;
- sustav se lako prenosi na drugu platformu, jer su specifičnosti konkretnog računala skrivene u unutrašnjim slojevima.

Mane slojevitog modela:

- koji put je teško podijeliti sustav u slojeve, jer osnovne usluge donjih slojeva mogu biti potrebne i znatno višim slojevima;
- mogu nastupiti problemi s performansama, zbog višestrukog interpretiranja.

3.2.3 Modeliranje kontrole

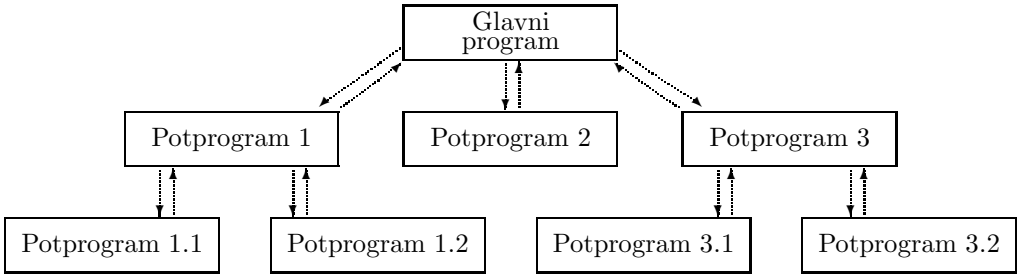
Postoje dva bitno različita pristupa kako se može ostvariti tok kontrole između pod-sustava:

- *Centralizirana kontrola:* jedan pod-sustav igra ulogu "kontrolora", dakle on pokreće i zaustavlja ostale pod-sustave.
- *Kontrola upravljana događajima* (engleski *event driven*): kontrola je distribuirana u tom smislu što pojedini pod-sustavi samostalno reagiraju na vanjske događaje. Ti događaji mogu biti generirani od drugih pod-sustava ili od okoline sustava.

Pod pojmom "događaj" mislimo na signal koji može poprimiti zadani raspon vrijednosti. To nije obični ulazni podatak, budući da vrijeme njegovog pojavljivanja nije pod kontrolom procesa koji ga obrađuje.

Postoje opet standardni modeli kontrole, koji su se pokazali upotrebljivi u raznim sustavima. Opisat ćemo četiri takva modela, od kojih prva dva spadaju među centralizirane, a druga dva među one upravljane događajima. Svaki model kontrole može se kombinirati s bilo kojim modelom za strukturu sustava.

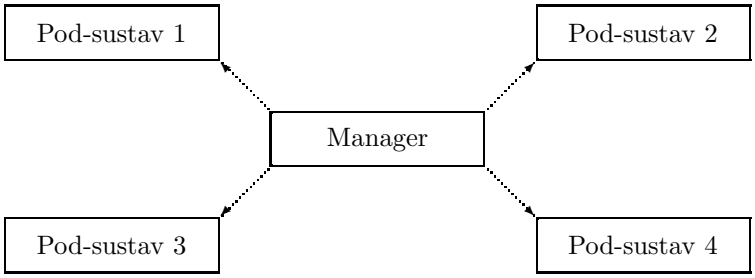
"Call-return" model. Spada među modele sa centraliziranom kontrolom. Riječ je o ideji koja je implementirana u klasičnim programskim jezicima. Kontrola kreće s vrha hijerarhije pod-sustava (iz glavnog programa) i prosljeđuje se na niže članove hijerarhije (preko poziva potprograma). Model je prikazan na Slici 3.5.



Slika 3.5: call-return model za kontrolu.

Ovaj model je primjenjiv samo za sekvencijalne sustave: naime u jednom trenutku samo jedan pod-sustav je aktivan. Prednost modela je mogućnost jednostavne analize toka kontrole. Mana je ta što je teško obraditi iznimne situacije (engleski: exceptions) koje remete normalni rad.

Model managera. Jedan pod-sustav kontrolira ostale, tako što im šalje signale za početak, kraj, odnosno sinkronizaciju njihovog rada. Svi pod-sustavi u principu rade paralelno, makar je to moguće simulirati i u sekvencijalnom sustavu. Manager izvršava beskonačnu petlju u kojoj ispituje varijable stanja da bi ustanovio koje pod-sustave treba pokrenuti ili zaustaviti. Model je predočen Slikom 3.6.

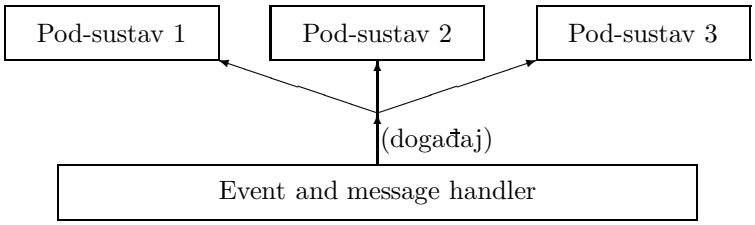


Slika 3.6: kontrola prema modelu managera.

Ovaj model sa centraliziranom kontrolom pogodan je za takozvane "meke" sustave u realnom vremenu: pod-sustavi su vezani uz senzore i aktuatore, a zahtjevi za brzinom odziva u realnom vremenu nisu prestrogi.

"Broadcast" model. Spada u modele upravljane događajima. Bilo koji događaj emitira se svim pod-sustavima. Odgovara samo onaj pod-sustav koji je napravljen da obrađuje taj događaj. Glavna razlika u odnosu na prethodni model je u tome što upravljanje nije prepušteno jednom pod-sustavu, već svaki pod-sustav samostalno odlučuje da li će reagirati na događaj ili neće. Model je ilustriran Slikom 3.7.

Ovaj model pogodan je za sustave u realnom vremenu sa "srednje tvrdim" zahtjevima na brzinu odgovora na događaje. Moguća je varijanta sa selektivnim emitiranjem događaja samo onim pod-sustavima koji su "pretplaćeni" na određene vrste događaja.



Slika 3.7: broadcast model za kontrolu.

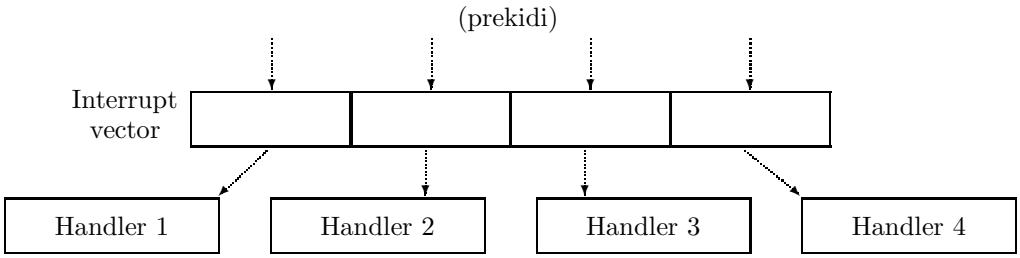
Prednosti broadcast modela:

- lagano se dodaje novi pod-sustav;
- bilo koji pod-sustav lagano može aktivirati bilo koji drugi, bez da zna njegovo ime i lokaciju (tako što generira odgovarajući događaj);
- primjenjiv je u slučaju distribuiranih sustava, dakle na mreži računala.

Mane broadcast modela:

- pod-sustavi ne znaju da li će i kada neki događaj (kojeg su sami generirali) biti obrađen;
- mogući su konflikti ako više pod-sustava obrađuje isti događaj.

Model upravljan prekidom (engleski: interrupt driven) također je upravljan događajima. Svakoj vrsti vanjskog događaja pridružen je jedan hardverski prekid (engleski: interrupt). Svakom prekidu pridružen je posebni pod-sustav - takozvani interrupt handler. Kad procesor primi prekid odgovarajuće vrste, hardverska sklopka prekida trenutni proces i prebacuje kontrolu na pripadni handler. Taj handler obradi događaj, nakon čega procesor nastavlja onaj proces koji je bio prekinut. Model se vidi na Slici 3.8.



Slika 3.8: model za kontrolu upravljan prekidom.

Ovaj model koristi se za "tvrde" sustave u realnom vremenu. Može se kombinirati s nekim modelom centralizirane kontrole.

Prednosti modela upravljanog prekidom:

- daje najbrži mogući odgovor na odabrane događaje.

Mane modela upravljanog prekidom:

- nepogodan je za testiranje;
- otežana je nadogradnja ukoliko je broj prekida potrošen.

3.3 Arhitekture distribuiranih sustava

Distribuirani sustav je sustav čiji dijelovi su raspoređeni na više umreženih računala. Svi današnji veliki sustavi su distribuirani. Nekadašnja mainframe računala s izravno spojenim terminalima zamijenili su skupovi umreženih poslužiteljskih i osobnih računala. Umjesto privatnih mreža s posebnom namjenom, za komunikaciju se koriste javne mreže Internet tipa. Očekuje se da će razvoj brzih bežičnih mreža omogućiti i uključivanje ručnih uređaja poput palmtopa ili mobitela - na taj način dobit će se "sveprisutni" distribuirani sustavi.

Jedan od ključnih aspekata u oblikovanju distribuiranih sustava je odabir modela za strukturu sustava (vidi Odjeljak 3.2). Obradit ćemo dva generička modela, od kojih se drugi može shvatiti kao poopćenje prvoga: arhitekture s klijentima i poslužiteljima, odnosno arhitekture s distribuiranim objektima.

3.3.1 Prednosti i mane distribuiranih sustava

Poželjna svojstva, a ujedno i prednosti distribuiranih sustava su:

- *Dijeljenje resursa.* Moguće je s jednog računala koristiti hardverske ili softverske resurse koji pripadaju drugom računalu, na primjer disk, štampač, datoteku, compiler.
- *Otvorenost.* Moguće je međusobno povezati hardver i softver različitih proizvođača. To zahtijeva poštivanje određenih standarda za komunikaciju.
- *Paralelnost.* Više procesa može se istovremeno odvijati na različitim računalima, te po potrebi međusobno komunicirati preko mreže.
- *Skalabilnost.* Performanse sustava mogu se u principu povećati dodavanjem novih računala.
- *Robustnost* (engleski *fault tolerance*). U slučaju kvara jednog računala u principu je moguće poslove preraspodijeliti na preostala računala, tako da sustav i dalje radi uz nešto oslabljene performanse.
- *Transparentnost.* Korisniku se sustav može predočiti kao integrirana cjelina, dakle korisnik ne mora znati ni brinuti o tome gdje se fizički nalaze resursi koje on koristi.

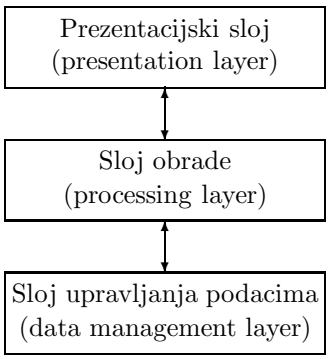
Mane distribuiranih sustava su:

- *Složenost* u odnosu na centralizirane sustave. Teško ih je testirati. Paralelni rad može dovesti do suptilnih grešaka. Koriste se složeni softveri za mrežnu komunikaciju.
- *Smanjena sigurnost.* Podaci putuju mrežom pa ih je moguće "prisluškivati" ili čak mijenjati. Napadač se može lažno predstaviti kao dio sustava.
- *Otežano upravljanje.* Veći broj raznorodnih računala teže je održavati nego jedno računalo.
- *Nepredvidivost kakvoće usluge.* Brzina odziva ovisi o ukupnom opterećenju (javne) mreže na koje nemamo utjecaja.

3.3.2 Arhitekture s klijentima i poslužiteljima

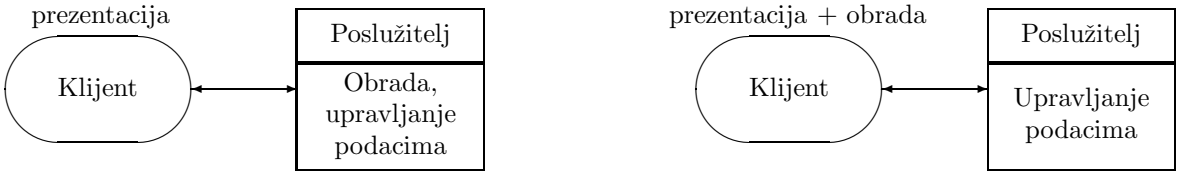
Model za strukturu sustava zasnovan na klijentima i poslužiteljima već smo ih spomenuli u Odjeljku 3.2. Sustav se sastoji od klijenata i poslužitelja povezanih mrežom. Poslužitelji daju usluge, klijenti traže usluge. Dozvoljavamo da poslužitelj zatraži uslugu od drugog poslužitelja, no ne i od klijenta. Klijenti moraju biti svjesni postojanja poslužitelja, no ne moraju znati za druge klijente. I klijenti i poslužitelji su shvaćeni kao procesi (softverske komponente) koji se na različite načine mogu pridružiti procesorima (hardverskim komponentama). Prilog 3.10 prikazuje sustav koji se sastoji od 12 klijenata i 4 poslužitelja. U Prilogu 3.11 vidi se razmještaj tih procesa na 8 računala, tako da jedno računalo izvodi više klijentskih odnosno poslužiteljskih procesa.

Arhitektura sustava s klijentima i poslužiteljima treba odražavati logičku strukturu aplikacije. Jedan način gledanja na aplikaciju je njena podjela na *tri sloja* (engleski *layers*), u skladu sa Slikom 3.9. Dakle razlikujemo sloj za prezentaciju, obradu odnosno upravljanje podacima. Prezentacijski sloj bavi se prikazivanjem informacija korisniku te interakcijom aplikacije i korisnika. Sloj obrade implementira logiku same aplikacije (engleski: business rules). Sloj upravljanja podacima izvršava transakcije nad bazom podataka. Ova tri sloja treba razlikovati zato što se svaki od njih može staviti na drugo računalo.



Slika 3.9: podjela aplikacije na tri sloja.

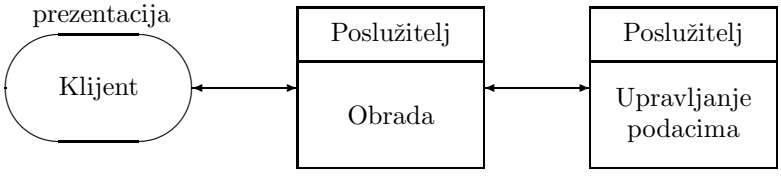
Jednostavni razmještaj slojeva na računala zove se *dvoredni* (engleski *two-tier*), gdje je aplikacija raspoređena na jedno poslužiteljsko računalo i mnogo klijentskih računala. Pritom u skladu sa Slikom 3.10 postoje dvije mogućnosti, budući da tri sloja treba smjestiti na dva računala.



Slika 3.10: dvoredni razmještaji slojeva aplikacije na računala.

Kod modela *mršavog klijenta* (engleski *thin client*) većinu posla obavlja poslužitelj, a klijent se bavi samo prezentacijom. Primjeri su: aplikacija s web sučeljem (klijent je zapravo web browser), ili stara "baštinjena" mainframe aplikacija (klijent je emulator tekstualnog terminala). Kod modela *debelog klijenta* (engleski *fat client*) klijent izvršava veći dio aplikacije, a poslužitelj se bavi isključivo transakcijama nad bazom podataka. U Prilogu 3.12 imamo primjer sustava bankomata, gdje je bankomat klijent, a bančin mainframe je poslužitelj.

Prednost mršavog klijenta je da klijentsko računalo ne mora biti pravi PC nego može biti i nešto jednostavnije i jeftinije; mana je da je poslužitelj usko grlo. Prednost debelog klijenta je bolji raspored tereta i veća autonomija klijenta; mana je složenije administriranje jer se nova verzija aplikacije mora distribuirati na svaki od klijenata. Hibrid između mršavog i debelog klijenta predstavljaju aplikacije zasnovane na Java appletima koji se s poslužitelja "download-aju" na klijenta.



Slika 3.11: troredni razmještaj slojeva aplikacije na računala.

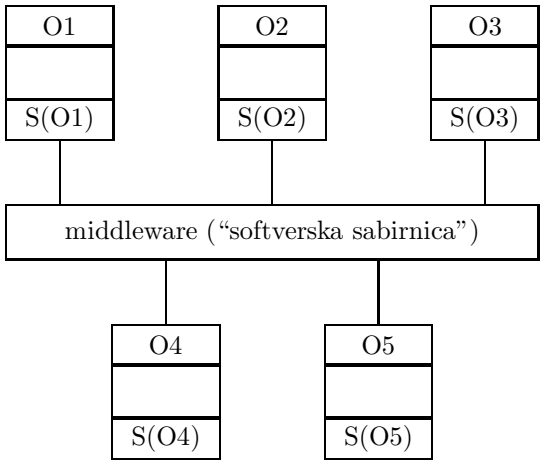
Složeniji raspored slojeva na računala zove se *troredni* (engleski *three-tier*) gdje je svaki sloj na zasebnom računalu. Ako se na bilo kojem sloju pojavi usko grlo, moguće je povećati broj računala. Troredni raspored prikazan je na Slici 3.11.

Prilog 3.13 prikazuje sustav za Internet-bankarstvo. Korisnici ustvari gledaju web formulare pomoću običnog web browsera. Te web formulare dinamički oblikuje web server, služeći se podacima o bankovnim računima kojima upravlja database server. Ako promet postane prevelik, dodaje se više web servera.

3.3.3 Arhitekture s distribuiranim objektima

Kod ovog modela za strukturu sustava ne pravi se razlika između procesa-klijenata i procesa-poslužitelja. Sustav se promatra kao skup objekata koji su raspoređeni na raznim računalima i koji međusobno komuniciraju preko mreže. Dakle dijelovi sustava su objekti od kojih svaki daje sučelje na skup usluga koje on pruža. Isti objekt može u jednom trenutku tražiti uslugu od drugoga, a kasnije on može pružati uslugu drugome. Fizički raspored objekata po računalima je irelevantan za funkcioniranje sustava.

Komunikacija distribuiranih objekata ostvaruje se pomoću posredničkog softvera, takozvanog *middleware*-a. Kao što hardverska sabirnica omogućuje komuniciranje različitih PC kartica umetnutih u utore za proširenje, tako middleware igra ulogu "softverske sabirnice". Middleware je nužan zato što objekti mogu biti implementirani u različitim programskim jezicima, zato što mogu raditi na različitim platformama, te zato što njihova stvarna imena i fizičke lokacije ne trebaju biti poznate drugim objektima. Zahvaljujući middleware-u, komunikacija objekata odvija se na transparentan i fleksibilan način. Cijela arhitektura prikazana je na Slici 3.12.



Slika 3.12: distribuirani objekti i njihova komunikacija preko "middleware-a".

U ovom trenutku postoje dva važna standarda za middleware.

- *CORBA* (engleski *Common Object Request Broker Architecture*). To je skup standarda koje je definirao OMG. Konzorcij OMG (engleska kratica od Object Management Group) uključuje kompanije Sun, HP, IBM i druge. Postoje implementacije CORBA-e za UNIX, Linux i Microsoft Windows.

- *DCOM* (engleska kratica od *Distributed Component Object Model*). To je standard razvijen i implementiran od Microsofta, te integriran u operacijske sustave Microsoft Windows. Model distribuiranog računanja je manje općenit od CORBA-inog, te je ograničen na komuniciranje Windows računala.

Fizički raspored middleware-a po računalima je takav da svako računalo koje ima objekte također ima i svoju kopiju middleware-a. Dva objekta koja se nalaze na istom računalu komuniciraju preko lokalne kopije middleware-a, dok se komunikacija objekata na različitim računalima ostvaruje tako da pripadni middleware-i "razgovaraju".

Arhitektura s distribuiranim objektima je dobra zato jer ona u najvećoj mjeri može iskoristiti prednosti distribuiranih sustava.

- Fizički raspored objekata je irelevantan sa stanovišta korektnog rada aplikacije, pa se može podešavati tako da osigura što bolje performanse.
- Sustav je otvoren u smislu da pojedini objekti mogu biti razvijeni u različitim programskim jezicima za različite platforme.
- Sustav je skalabilan u smislu da je lagano dodati nove objekte ili kopije istog objekta da bi se otklonila uska grla.
- Sustav se dinamički može rekonfigurirati, tako da objekti u toku rada sele s jednog računala na drugo, u skladu s promijenjenim zahtjevima.

U Prilogu 3.14 vidi se arhitektura s distribuiranim objektima za "data mining" sustav. Taj sustav nastoji otkriti zakonitosti među podacima koji su spremljeni u različitim bazama podataka i odnose se na prodaju više vrsta roba u više robnih kuća. Na primjer, jedna takva zakonitost bila bi: "Muški kupac od 30-tak godina koji u petak popodne dolazi kupiti pelene vjerojatno će kupiti i pivo". Svaka od baza podataka prikazana je kao distribuirani objekt s read-only pristupom. Svaki od objekata-integratora bavi se drukčijom vrstom odnosa među podacima: na primjer jedan integrator proučava varijacije prodaje roba ovisne o godišnjim dobima, a drugi se zanima za korelacije između različitih tipova roba. Objekt-vizualizator grafički prikazuje otkrivene zakonitosti. Vidimo da većina objekata i traži i daje usluge.

3.4 Objektni pristup oblikovanju

Riječ je o takvom pristupu oblikovanju gdje projektant prvenstveno razmišlja o "objektima" (predmetima, osobama, pojavama, ...) a manje o "funkcijama" (operacijama, aktivnostima, procesima, ...). Pristup je postao popularan u 80-tim u 90-tim godinama 20. stoljeća, a bio je najprije motiviran razvojem grafičkih sučelja. Danas je to uobičajeni način oblikovanja softvera.

3.4.1 Objekti i klase

Osnovni pojmovi unutar objektnog oblikovanja su objekt i klasa.

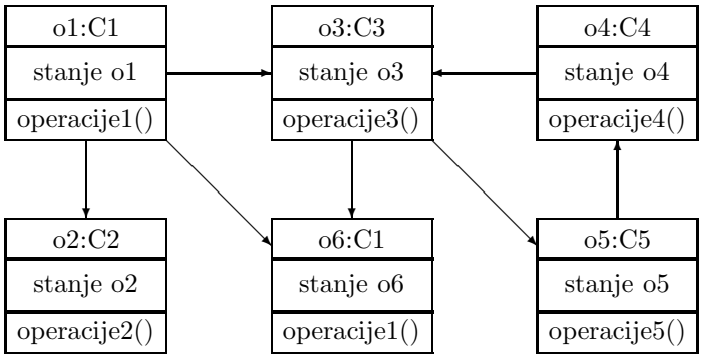
- *Objekt* je entitet uz kojeg je vezan skup atributa i skup operacija. Vrijednosti atributa vezanih uz objekt određuju stanje tog objekta. Operacije vezane uz objekt mijenjaju vrijednosti objektovih atributa, te na taj način djeluju na stanje tog objekta.
- *Klasa* je skup istovrsnih objekata Definicija klase služi kao obrazac za stvaranje objekata. Definicija klase sadrži deklaracije svih atributa i operacija vezanih uz objekt iz te klase.

Prilog 3.15 definira klasu objekata koji odgovaraju zaposlenicima neke firme. Koristi se UML notacija. U skladu s UML-om, ime klase piše na vrhu pravokutnika, deklaracije atributa navedene su u gornjem odjeljku, a deklaracije operacija u donjem odjeljku. U UML-u termin "operacija" znači specifikacija neke akcije, a termin "metoda" znači implementacija operacije.

3.4.2 Svojstva objektno-oblikovanog sustava

Konačni rezultat objektnog oblikovanja je objektno-oblikovani sustav (design). On se sastoji od objekata koji su u međusobnoj interakciji - vidi Sliku 3.13. Svaki objekt pripada nekoj klasi. Interakcija objekata odvija se na način da jedan objekt (klijent) pokrene operaciju drugog objekta (poslužitelja). U tom smislu, operacije se mogu tumačiti kao servisi, a svaki objekt može se pojaviti u ulozi klijenta i u ulozi poslužitelja. Prilikom poziva operacija, objekti razmjenjuju podatke (parametre i rezultate).

U objektno-oblikovanom sustavu ne postoje globalne funkcije koje bi se izvršavale neovisno o objektima, već je funkcionalnost sustava izražena isključivo u terminima operacija vezanih uz objekte. Slično, ne postoje globalne varijable, već se stanje sustava dobiva kao zbroj stanja pojedinih objekata.



Slika 3.13: sustav sastavljen od objekata u međusobnoj interakciji.

Objekti unutar objektno-oblikovanog sustava u principu se izvršavaju paralelno. To daje mogućnost da se sustav distribuira na više računala, ili da se istovremeni rad objekata simulira na jednom računalu.

Novi objektno-oblikovani sustavi mogu biti razvijeni uz upotrebu objekata koji su bili stvoreni za prethodne sustave. To smanjuje cijenu razvoja softvera i vodi ka upotrebi standardnih objekata. Ipak, pokazuje se da su za ponovnu upotrebu pogodnije veće cjeline od pojedinih objekata (vidi Odjeljak 3.6).

Važno svojstvo objektno-oblikovanog sustava je da on omogućuje laganu evoluciju. Naime, sustav se sastoji od objekata, dakle dijelova s jakom unutarnjom kohezijom i labavim vezama prema van. Evolucija sustava zahtijevat će promjenu unutarnje građe pojedinog objekta ili dodavanje novih objekata - takvi zahvati nemaju značajnog učinka na ostatak sustava (vidi primjer na kraju ovog odjeljka).

3.4.3 Mjesto objektnog oblikovanja unutar softverskog procesa

Objektno oblikovanje sastavni je dio objektno-orijentiranih metoda za razvoj softvera (Booch, Rumbaugh, Jacobson, RUP/UML). Osim oblikovanja, te metode uključuju objektnu analizu (specifikaciju), te objektno programiranje. I analiza i oblikovanje bave se uočavanjem objekata i klasa te razvijanjem objektnih modela. Razlika između analize i oblikovanja nije čvrsto određena i svodi se na sljedeće.

- U analizi se koncentriramo na modele aplikacijske domene, dok se u oblikovanju bavimo modeliranjem budućeg softvera.
- Analiza je manje precizna, tj, uočavaju se samo najvažniji objekti. Oblikovanje dodaje nove objekte koji su na primjer potrebni za korisničko sučelje ili za unutrašnje funkcioniranje složenih objekata.
- Analiza ne ulazi u detalje sučelja objekata, dok oblikovanje do kraja definira to sučelje.

Objektno programiranje bavi se realizacijom objektno-oblikovanog sustava pomoću objektno-orijentiranog programskog jezika kao što je Java ili C++. Posao je relativno jednostavan, budući da objektno-orijentirani jezici uključuju sve što je potrebno za neposredno definiranje klasa, stvaranje objekata, te pozivanje operacija.

3.4.4 Pod-aktivnosti unutar objektnog oblikovanja

Većina objektno-orijentiranih metoda za razvoj softvera unutar objektnog oblikovanja predviđaju sljedeće pod-aktivnosti:

- razumijevanje konteksta (okoline) i načina upotrebe sustava,
- oblikovanje arhitekture sustava,
- identificiranje glavnih objekata (klasa) u sustavu,
- razvoj objektnih modela,
- specificiranje sučelja za objekte.

Te pod-aktivnosti su u priličnoj mjeri isprepletene, tj. teško ih je obavljati u nekom fiksiranom redoslijedu.

Kontekst sustava opisuje se statičkim kontekstnim modelima, tako da se odrede drugi sustavi koji čine okolinu našeg sustava i veze između tih drugih sustava i našeg sustava.

Načini upotrebe sustava dokumentiraju se dinamičkim modelima ponašanja, koji prikazuju interakciju sustava s okolinom.

Oblikovanje arhitekture sastoji se od identificiranja pod-sustava i veza među pod-sustavima. Pritom se pod-sustav u UML-notaciji može prikazati kao "package", dakle skup objekata.

Objektni modeli su oni isti koje smo spominjali kod specifikacije, dakle: model nasljeđivanja, agregacije, te ponašanja objekata. Također se koristi model promjene stanja objekta.

Sučelje objekta sastoji se od sučelja operacija koje će biti dostupne drugim objektima. Definiraju se broj i tipovi parametara za svaku operaciju, te tip rezultata operacije.

3.4.5 Primjer objektnog oblikovanja

U ovom primjeru oblikovat ćemo *meteorološku stanicu*. Ta stanica dio je većeg sustava opisanog sljedećim neformalnim tekstom.

> Sustav za iscrtavanje meteoroloških karata ima zadaću stvaranja meteoroloških karata u pravilnim vremenskim razmacima, korištenjem podataka koji su skupljeni s udaljenih nenadziranih meteoroloških stanica, te podataka iz drugih izvora kao što su baloni ili sateliti. Meteorološka stanica šalje svoje podatke u područno računalo kao odgovor na zahtjev iz tog računala. Područno računalo provjerava i integrira podatke, te ih arhivira. Na osnovu arhiviranih podataka i baze praznih digitaliziranih karata stvaraju se lokalne meteorološke karte, koje se mogu štampati ili prikazati u raznim formatima.

Prilog 3.16 prikazuje arhitekturu cijelog sustava u UML notaciji. Budući da je naša namjera modelirati samo meteorološku stanicu, ova slika može se interpretirati kao statički model za razumijevanje konteksta (okoline) za stanicu. Prilog 3.17 dobiven je analizom zahtjeva na stanicu iz prethodnog opisa, te predstavlja dinamički model interakcije stanice s okolinom. Daljnja razrada jedne interakcije vidi se u Prilogu 3.18.

Informacije iz Priloga 3.16 i 3.17 omogućuju nam da predložimo arhitekturu za našu meteorološku stanicu. Riječ je o slojevitoj arhitekturi od tri sloja - svaki sloj tumači se kao UML-ov "package" - sve je to prikazano u Prilogu 3.19.

Dalje slijedi pod-aktivnost identificiranja objekata. Neke od klasa vide se u Prilogu 3.20. Objekti iz klasa `GroundThermometer`, `Anemometer` i `Barometer` odgovaraju fizičkim instrumentima. Klasa

`WeatherStation` daje sučelje stanice s okolinom, pa sadrži interakcije iz Priloga 3.17. Klasa `WeatherData` enkapsulira sažete podatke iz različitih instrumenata, a njene operacije bave se skupljanjem i sažimanjem.

U nastavku se bavimo izradom objektnih modela. Prilog 3.21 spada među modele ponašanja i on pokazuje veze između klasa ili paketa koje nastaju zato što neki objekt iz jedne klase ili paketa koristi operacije objekta iz druge klase ili paketa. Prilog 3.22 prikazuje niz interakcija između objekata koje nastaju kad područno računalo zatraži podatke od meteorološke stanice. Područno računalo predstavlja vanjski svijet (korisnika) za našu stanicu, i zato je prikazano simbolom čovječuljka. Prilog 3.23 predstavlja dijagram promjene stanja objekta iz klase `WeatherStation`. Debeli kružić označava polazno stanje, a strelice prikazuju prelaske u daljnja stanja ovisno o primljenim porukama (pokrenutim operacijama) ili drugim signalima.

Na kraju oblikovanja meteorološke stanice, bavimo se detaljnom specifikacijom sučelja objekata. Prilog 3.24 sadrži sučelje objekta iz klase `WeatherStation` zapisano u sintaksi Jave.

Da bi ilustrirali kako objektno-oblikovani sustav lagano može evoluirati, pretpostavimo da se naknadno pojavio zahtjev da meteorološka stanica mora mjeriti i zagađenost zraka. To zahtijeva sljedeće izmjene.

- Dodaje se klasa `AirQuality`, na istoj razini kao `WeatherData`, koja enkapsulira sažete podatke o zagađenosti.
- U klasu `WeatherStation` dodaje se operacija `ReportAirQuality()`, kojom se podaci o zagađenosti šalju prema područnom računalu.
- Dodaju se klase `NOMeter`, `SmokeMeter` i `BenzeneMeter`, koje odgovaraju fizičkim instrumentima za mjerenje razine dušičnog oksida, dima i benzena.

Sve ove promjene prikazane su u Prilogu 3.25. Dakle vidi se izmijenjena klasa `WeatherStation` i nove klase. Ove promjene nemaju nikakav učinak na funkcioniranje ostalih dijelova sustava.

3.5 Oblikovanje korisničkog sučelja

Čest je slučaj da se korisničko sučelje definira već u fazi specifikacije. Naime:

- sučelje može poslužiti kao način da se opiše što sustav treba raditi,
- sučelje može biti odabrano od korisnika, u skladu s navikama i standardima.

Ipak, strogo govoreći, definiranje sučelja spada u oblikovanje, pogotovo ako je riječ o složenijem sustavu s nestandardnim načinom komuniciranja s korisnikom. Sučelje je važno svojstvo po kojem će cijeli sustav biti ocijenjen, pa mu treba posvetiti veliku pažnju.

S obzirom na raspoloživi hardver, razlikujemo *tekstualno* i *grafičko* korisničko sučelje. Danas prevladava grafičko sučelje, a njegova svojstva su sljedeća.

- Veći broj prozora omogućuje istovremeni prikaz raznih informacija na istom zaslonu.
- Ikone omogućuju minimizirani slikovni prikaz datoteki, procesa ili prozora.
- Izbornici omogućuju biranje naredbi bez njihovog utipkavanja.
- Pokazivački uređaji (miš i slični) omogućuju biranje s izbornika ili pokazivanje mjesta unutar prozora koje je od interesa.
- Slike i crteži mogu se kombinirati s tekstom na istom zaslonu.

Sama aktivnost oblikovanja sučelja obično se odvija na iterativni način, u suradnji s korisnikom, uz upotrebu prototipova (vidi Odjeljak 2.3).

3.5.1 Principi oblikovanja korisničkog sučelja

Kod oblikovanja korisničkog sučelja moramo uzeti u obzir fizičke i mentalne mogućnosti ljudi koji će koristiti softver. Drugim riječima, moramo poštivati sljedeće principe dobrog oblikovanja.

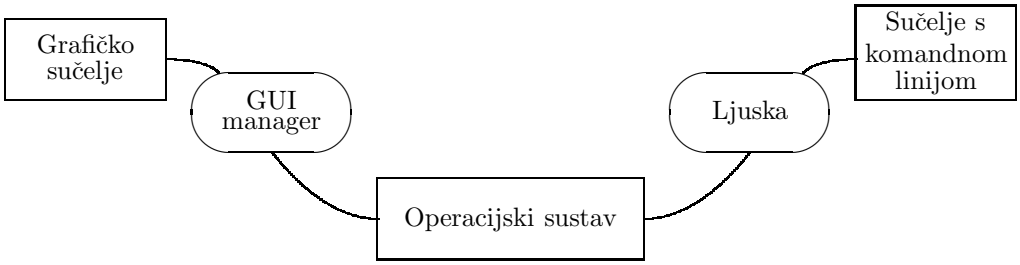
- *Bliskost korisniku.* Sučelje treba koristiti termine i predodžbe koje su bliske dotičnoj grupi ljudi, dakle jezik aplikacijske domene a ne jezik informatike.
- *Konzistentnost.* Slične operacije svugdje se trebaju obavljati na sličan način.
- *Minimum iznenađenja.* Korisnici ne smiju biti iznenađeni ponašanjem sustava ili neobaviješteni o njegovim radnjama.
- *Mogućnost oporavka.* Moraju biti uključeni mehanizmi koji omogućuju korisnicima da isprave svoje greške, na primjer sustav mora tražiti od korisnika potvrdu destruktivne akcije ili mu omogućiti operaciju "undo".
- *Vođenje korisnika.* Sustav mora proizvoditi poruke koje prate korisnikove akcije, te mora pružiti pomoć korisniku ovisnu o kontekstu.
- *Uvažavanje različitosti korisnika.* Sučelje se treba moći prilagoditi potrebama različitih vrsta korisnika, na primjer početnika, povremenih korisnika, "power user-a" i drugih.

3.5.2 Interakcija korisnika sa sustavom

Prvi važni aspekt designa korišničkog sučelja je interakcija korisnika sa sustavom, dakle način na koji korisnik daje informacije ili zahtjeve sustavu. Postoji sljedećih pet načina.

- *Direktna manipulacija.* Korisnik "rukuje" objektima na zaslonu. Na primjer, da bi pobrisao datoteku, korisnik mišem odvlači njenu ikonu u "koš za smeće". Prednosti su brza i intuitivna interakcija, te lagano učenje. Mane su da se postavljaju veliki zahtjevi na hardver, te da je za komplicirane operacije teško naći očiglednu metaforu.
- *Izbor s izbornika.* Korisnik bira naredbu s izbornika, često u kombinaciji sa selekcijom objekta na zaslonu. Na primjer, da bi pobrisao datoteku, korisnik selektira odgovarajuću ikonu, te zatim na izborniku bira naredbu "Delete". Prednosti: ne moraju se pamtiti imena naredbi, nema tipkanja, izbjegavaju se neke vrste grešaka (neprimjenjive opcije na izborniku automatski se blokiraju). Mane: prespora interakcija za iskusne korisnike, postaje nezgrapno ako ima puno naredbi i opcija.
- *Ispunjavanje formulara.* Korisnik ispunjava polja na formularu te zatim pritisne gumb za akciju. Na primjer, da bi pobrisao datoteku, korisnik upiše njeno ime u odgovarajuće polje te zatim aktivira gumb "Delete". Prednosti: lagano za naučiti, pogodno onda kad ima puno podataka. Mana: može zauzeti puno mjesta na zaslonu.
- *Korištenje komandnog jezika.* Korisnik na "komandnu liniju" upisuje naredbu i njene opcije. Na primjer, da bi pobrisao datoteku, korisnik upisuje "`rm -i ime_datoteke`". Prednosti: snažan i fleksibilan način, mogućnost kombiniranja naredbi (pipelining, shell skripte, ...). Mane: korisnik mora naučiti jezik, moguće su sintaktičke i druge greške.
- *Korištenje prirodnog jezika.* Korisnik upisuje (ili čak izgovara) naredbu u prirodnom jeziku. Na primjer, korisnik upisuje rečenicu "pobriši tu-i-tu datoteku". Prednosti: dobro za povremene korisnike. Mane: zahtijeva više pisanja nego komandni jezik, razumijevanje prirodnog jezika je nepouzdano pa su mogući nesporazumi.

Povremenim korisnicima više odgovara direktna manipulacija ili izbornici, dok "power user-i" često preferiraju tekstualne komande uključujući i "keyboard shortcuts". Zato mnogi sustavi podržavaju više načina interakcije. Na primjer, rad s UNIX-om ili Linux-om moguć je preko grafičkog sučelja ili preko tekstualne komandne linije (ljuske), kao što je ilustrirano na Slici 3.14.



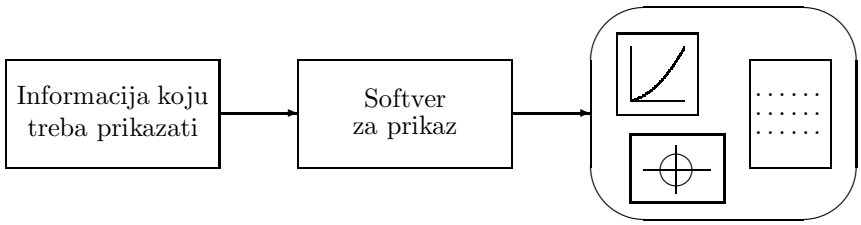
Slika 3.14: višestruko korisničko sučelje.

3.5.3 Prikazivanje informacija

Drugi važni aspekt designa korisničkog sučelja je prikazivanje informacija, dakle način na koji sustav korisniku predočava informacije. Ista informacija obično se može prikazati na tekstualni (numerički) ili na grafički način. Da bi odlučili kako da se prikaže pojedina informacija, moramo uzeti u obzir sljedeće faktore.

- Da li korisnika zanima precizna ili samo približna vrijednost?
- Da li su važne apsolutne ili relativne vrijednosti ili odnosi?
- Kako se brzo vrijednosti mijenjaju i da li se promjene odmah trebaju pokazati korisniku?
- Da li korisnik treba poduzeti neku akciju kao odgovor na promjene?

Idealno bi bilo kad bi se softver za prikaz informacije odvojio od same informacije, kao što je prikazano na Slici 3.15. To bi omogućilo da se način prikazivanja mijenja neovisno o dijelu sustava koji proizvodi podatke.



Slika 3.15: razlikovanje informacije od njenog prikaza.

Za podatke koji se ne mijenjaju za vrijeme jedne seanse obično je pogodan tekstualni prikaz - naime on je precizan i ne zauzima mnogo mjesta. Promjenjive podatke ili podatke gdje su važni relativni odnosi treba prikazivati grafički. Prilog 3.26 sadrži dva načina prikazivanja podataka o mjesečnoj prodaji nekog proizvoda. Prilog 3.27 ilustrira razne grafičke prikaze promjenjivih podataka, a Prilog 3.28 prikaz relativnih vrijednosti. Precizne alfanumeričke informacije mogu se nadopuniti grafičkim naglascima kao u Prilogu 3.29.

U slučajevima kad treba prikazati jako velike količine informacija, smišljaju se posebne vizualizacije. Slijedi nekoliko primjera.

- Informacije o vremenu skupljene iz raznih izvora prikazuju se na karti kao izobare, fronte, ciklone, anticiklone itd.
- Stanje telefonske mreže prikazuje se kao skup čvorova i spojnica.
- Stanje tehnološkog procesa u kemijskoj tvornici prikazuje se tako da se pokažu pritisci i temperature u povezanom skupu spremnika.

- Skup web stranica prikazuje se kao stablo.
- Model molekule prikazuje se u tri dimenzije, tako da se slika može okretati u svim smjerovima.

3.5.4 Vođenje korisnika

Treći važni aspekt designa korisničkog sučelja je vođenje korisnika, dakle način na koji sustav prati korisnikov rad i pomaže mu. Vođenje korisnika obavlja se na tri razine:

- poruke koje sustav sam proizvodi kao odgovor na korisnikove akcije, dakle obavijesti, upozorenja, te poruke o greški;
- upute (engleski: on-line help) koje sustav prikazuje na zahtjev korisnika;
- korisnička dokumentacija koja se isporučuje zajedno sa sustavom.

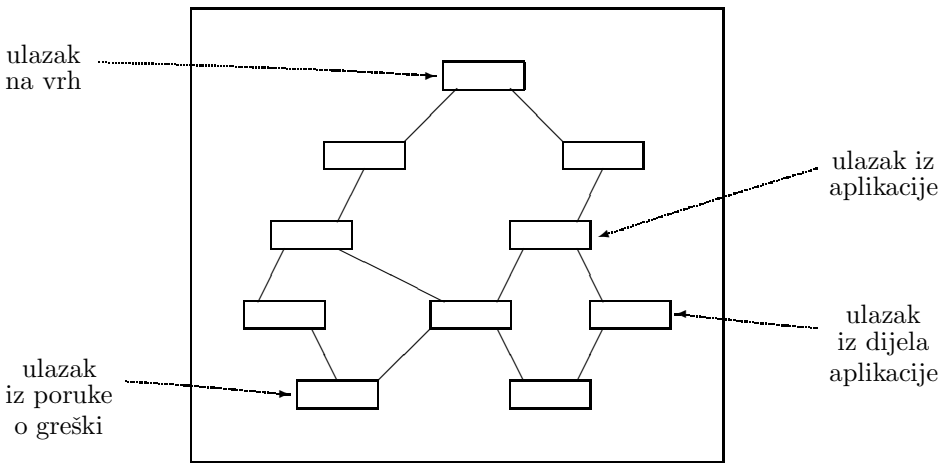
Svaku od tih razina objasnit ćemo detaljnije u posebnom odlomku.

Poruke treba oblikovati uzimajući u obzir sljedeće faktore:

- kontekst: sadržaj poruke prilagođava se situaciji u kojoj se korisnik nalazi;
- iskustvo korisnika: opširni tekstovi za početnike, sasvim kratki za iskusne;
- stručni profil korisnika: terminologija iz korisnikove struke;
- bonton i stil: uljudnost, pozitivni pristup, izbjegavanje dosjetki i duhovitosti;
- kultura: u skladu s običajima dotične zemlje.

Poruke o *greškama* moraju biti posebno uljudne, koncizne, konstruktivne i pisane u korisniku bliskoj terminologiji. One ne smiju optuživati korisnika da je napravio grešku, već trebaju na posredan način objasniti situaciju. Dalje, poruke o greškama trebaju objasniti kako da se greška ispravi, te trebaju biti vezane na kontekstno-ovisni help. Prilozi 3.30 i 3.31 sadrže primjer loše odnosno dobro oblikovane porike vezane uz informacijski sustav bolnice.

Upute obično imaju složenu mrežnu strukturu, kao što je prikazano na Slici 3.16. Dakle upute se sastoje od "kartica" teksta i međusobnih referenci. Zapravo je riječ o hijerarhiji (poglavlja, potpoglavlja, ...) s dodatnim unakrsnim vezama. Općenite informacije su na vrhu hijerarhije. Sustav treba korisniku osigurati razna mjesta ulaska u ovu mrežu, ovisno o tome da li on želi općenite informacije, upute o pojedinoj aplikaciji, precizne upute o određenoj naredbi unutar aplikacije, ili pak detaljnije objašnjenje vezano uz poruku o greški.



Slika 3.16: mrežna struktura tekstova s uputama.

Problem s ovakvom mrežnom strukturom je da se korisnik u njoj lako može izgubiti, pogotovo ako je ušao u njen donji dio. Donekle se može pomoći ako se ispisuju dodatne informacije u više prozora kao u Prilogu 3.32. Ti prozori mogu biti aktivni, to jest oni mogu služiti za brzo manevriranje po mreži.

Korisnička dokumentacija opisuje funkcije sustava na korisnicima razumljiv način. Distribuira se zajedno sa sustavom na papirnatom ili elektroničkom mediju. Sastoji se barem od sljedećih dokumenata.

- *Funkcionalni opis.* Namijenjen je evaluatorima sustava, dakle osobama koje će odlučiti da li da se koristi taj sustav ili neki drugi. Daje se grubi opis što sustav može a što ne može raditi. Priloženi su primjeri, tabele i dijagrami. Ne objašnjavaju se operativni detalji.
- *Vodič za instalaciju.* Namijenjen je sistem-administratoru ili korisniku. Opisuje instalaciju sustava na zadanoj vrsti računala. Sadrži opis distribucijskog medija, definiciju minimalne hardverske i softverske konfiguracije potrebne za pokretanje sustava, proceduru instaliranja i podešavanja.
- *Uvodni priručnik.* Namijenjen je novim korisnicima. Daje neformalni uvod u sustav, te opisuje njegovu "normalnu" upotrebu. Pisan je u formi "tečaja", sadrži mnogo primjera, te uputa kako izbjeći uobičajene greške.
- *Referentni priručnik.* Namijenjen je iskusnim korisnicima. U potpunosti, te na vrlo precizan način dokumentira sve funkcije sustava, sve oblike njegove upotrebe, sve greške koje mogu nastupiti. Nije u formi tečaja. Stil je formalan, struktura je stroga. Snalaženje u tekstu osigurano je preko indeksa pojmova.
- *Priručnik za administriranje.* Namijenjen je sistem-administratorima ili operatorima. Opisuje aktivnosti poput backup-a, upravljanja resursima, praćenja performansi, podešavanja rada, evidentiranja korisnika, postavljanja zaštite.

3.6 Ponovna upotreba softvera

U Odjeljku 1.2 već smo spomenuli model za softverski proces zasnovan na ponovnoj upotrebu softvera. Bit tog modela je u tome da se novi sustav u što većoj mjeri nastoji realizirati korištenjem dijelova iz već postojećih sustava. Naveli smo prednosti i mane takvog razvoja softvera.

Da bi ponovna upotreba softvera bila moguća, nužna su sljedeća tri uvjeta.

- Mora postojati katalog upotrebljivih dijelova softvera.
- Korisnik gotovih dijelova mora biti uvjeren u kvalitetu tih dijelova.
- Dijelovi u katalogu moraju biti dobro dokumentirani.

U ovom odjeljku detaljnije ćemo objasniti tri konkretna načina kako se može ostvariti ponovna upotreba softvera.

3.6.1 Razvoj zasnovan na komponentama

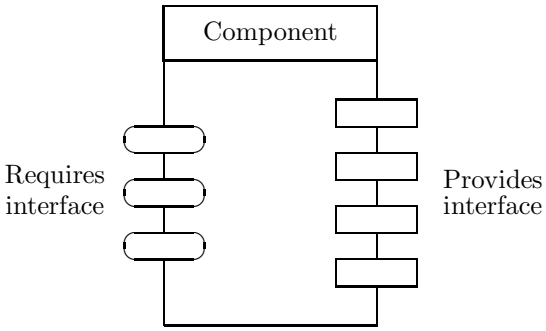
Riječ je o načinu razvoja softvera uz ponovnu upotrebu koji se pojavio u kasnim 90-tim godinama 20. stoljeća. Nastao je kao rezultat frustracije činjenicom da objektni pristup nije doveo do intenzivne ponovne upotrebe kao što se ispočetka očekivalo. Naime, uvidjelo se da su klase objekata premale cjeline da bi se uspješno prodavale kao dijelovi za ponovnu upotrebu (da bi se klasa ponovno upotrebila obično je potrebno ulaziti u njen izvorni kod). Umjesto pojedinih klasa, nastoje se stvoriti veće i apstraktnije cjeline za ponovnu upotrebu koje se zovu *komponente*. Komponentu treba promatrati kao samostalnog pružatelja usluga. Kad naš sustav treba neku uslugu, on poziva komponentu ne brinući o tome gdje se ona izvršava i u kojem programskom jeziku je ona bila razvijena.

Osnovna svojstva komponente su sljedeća.

- Komponenta je nezavisna izvršiva cjelina. Nije nam dostupan njen izvorni kod, tako da se ona ne kompilira zajedno s ostalim dijelovima našeg sustava.
- Komponenta objavljuje svoje sučelje, i sve operacije s njom odvijaju se kroz to sučelje. Sučelje komponente izražava se u terminima parametriziranih operacija. Unutarnje stanje komponente nam nije vidljivo.

Sučelje komponente sastoji se od dva dijela, kao što je prikazano na Slici 3.17.

- "Provides interface" definira servise koje komponenta pruža.
- "Requires interface" definira koji servisi moraju biti dostupni samoj komponenti u sustavu koji koristi tu komponentu. Ako tih servisa nema, komponenta neće moći raditi.



Slika 3.17: komponenta i njena sučelja.

U Prilogu 3.33 vidi se komponenta koja pruža usluge štampanja. Servisi koje ona pruža su: štampanje dokumenta, uvid u stanje reda vezanog uz pojedini štampač, registriranje ili de-registriranje štampača, prebacivanje posla štampanja iz reda jednog u red drugog štampača, izbacivanje posla iz reda za štampač. Zahtijevani servisi su `GetPDFIle` koji daje datoteku s opisom štampača i `PrinterInt` koji prosljeđuje naredbe određenom štampaču.

Danas postoje tvrtke koje proizvode ovakve komponente i prodaju ih drugim softverskim kućama. Komponenta za prodaju mora biti dovoljno općenita, pažljivo verificirana, te dobro dokumentirana.

Veličina komponente može varirati od veličine jedne bibliotečne funkcije, pa sve do veličine cijele aplikacije poput Microsoft Excel. Meyer (1999) razlikuje sljedećih pet stupnjeva veličine.

1. *Apstrakcija funkcije.* Komponenta implementira jednu funkciju kao što je sinus ili kvadratni korijen. "Provides interface" je ta funkcija.
2. *Slučajno grupiranje.* Komponenta je skup labavo povezanih entiteta kao što su deklaracije podataka, funkcije, itd. "Provides interface" sastoji se od imena tih entiteta.
3. *Apstraktni tip podataka.* Komponenta predstavlja apstraktni tip ili klasu u smislu objektno-orijentiranih programskih jezika. "Provides interface" sastoji se od operacija za stvaranje, promjenu ili čitanje primjerka tog apstraktnog tipa ili klase.
4. *Apstrakcija klastera.* Komponenta je skupina povezanih klasa koje rade zajedno. Takva skupina također se naziva "framework". "Provides interface" je unija svih sučelja za klase koje čine framework.
5. *Apstrakcija sustava.* Komponenta predstavlja samostalni sustav. Takva komponenta obično se naziva COTS sustav (engleska kratica od Commercial Off-The-Shelf System). "Provides interface" je takozvani API (engleska kratica od Application Programming Interface) koji je definiran zato da omogući programima pristup do naredbi i operacija unutar sustava.

Po današnjim mjerilima, rentabilnost ponovne upotrebe softvera zaista se postiže tek kod stupnjeva 4. i 5. Slijede primjeri frameworka (klastera).

- *Infrastrukturni framework.* Podržava razvoj infrastrukture kao što je komunikacija, grafičko korisničko sučelje, "parsiranje" komandi, i drugo.
- *Middleware framework.* Implementira odabrani standard za komunikaciju komponenti. Spominjali smo u Odjeljku 3.3. Primjeri: CORBA, Microsoft COM, Microsoft DCOM, JavaBeans.
- *Enterprise application framework.* Implementira operacije iz specifičnih aplikacijskih domena, na primjer financijski sustavi, knjižnice, ...

Dalje slijede primjeri COTS sustava.

- Sustav za upravljanje bazom podataka (DBMS), na primjer Oracle, Microsoft SQL Server, IBM DB2, MySQL.
- Tablični kalkulator poput Microsoft Excel.
- Emulator neuronskih mreža poput StatSoft Statistica.
- Kalkulator matematičkih funkcija poput WRI Mathematica.

3.6.2 Porodice aplikacija

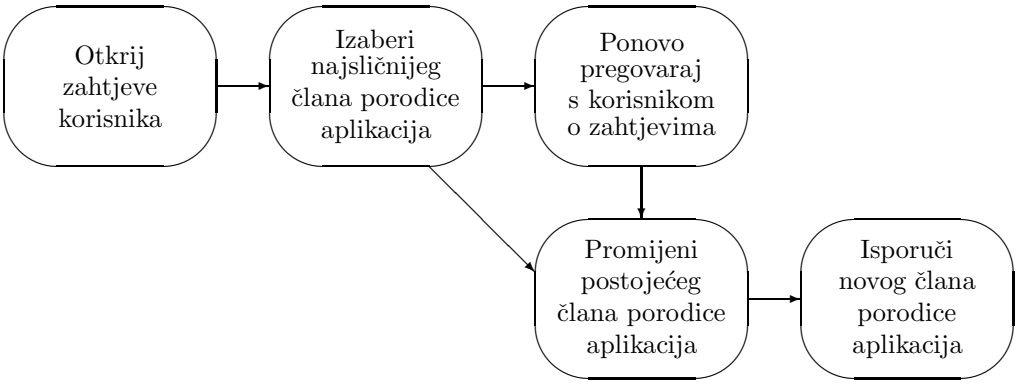
Riječ je o načinu ponovne upotrebe softvera koji je prisutan već desetljećima. *Porodica aplikacija* ili *produktna linija* je skup srodnih aplikacija koje imaju zajedničku arhitekturu. Svaki član porodice je specijaliziran u nekom smislu. Zajednička jezgra svih članova porodice ponovo se upotrebljava svaki put kad se pojavi potreba za novim članom porodice. Razvoj novog člana sastoji se od pisanja dodatnih dijelova ili od prepravljanja postojećih dijelova u nekoj od starih aplikacija.

Postoji nekoliko tipova specijalizacije jedne aplikacije unutar porodice.

- *Platformska specijalizacija.* Razne verzije u biti iste aplikacije razvijaju se za različite platforme (na primjer Microsoft Windows, SUN Solaris, Linux, ...). Funkcionalnost u svim verzijama je ista, a promijenjeni su oni dijelovi koji komuniciraju s hardverom i operacijskim sustavom.
- *Konfiguracijska specijalizacija.* Razne verzije aplikacije stvaraju se zato da bi rukovale s različitim vanjskim uređajima. Na primjer, sustav za dojavljivanje požara postoji u više verzija ovisno o tipu radio sustava. Funkcionalnost verzija varira s obzirom na drukčiju funkcionalnost vanjskih uređaja.
- *Funkcionalna specijalizacija.* Razne verzije aplikacije stvaraju se za kupce s različitim zahtjevima. Na primjer, informacijski sustav knjižnice modificira se za gradsku, odnosno školsku, odnosno znanstvenu knjižnicu. Moguće je da će neki dijelovi biti promijenjeni, te da će znanstvena knjižnica imati dodatne dijelove koji se odnose na znanstvene časopise.

Kao primjer ponovne upotrebe softvera korištenjem porodice aplikacija pogledajmo arhitekturu sustava za upravljanje inventarom u Primjeru 3.34. Riječ je o slojevitoj arhitekturi. Prati se raspored nekih komada opreme unutar neke organizacije. Ukoliko je riječ o elektrani i njenim generatorima i transformatorima, tada oprema u pravilu ne mijenja svoju lokaciju. Ukoliko isti sustav koristimo za praćenje opreme na fakultetu, morat ćemo dodati modul za premještanje na primjer računala iz jednog laboratorija u drugi, pa ćemo na taj način dobiti novog člana porodice aplikacija. Prilog 3.35 prikazuje značajnije promijenjenu verziju polazne aplikacije koja ovaj put služi za raspoređivanje vatrogasnih vozila ili vozila hitne pomoći; dodani su specifični moduli za planiranje rute vozila, za rad s kartama, za sučelje prema komunikacijskom sustavu itd. Baza podataka je podijeljena u tri dijela koji se odnose na vozila, drugu opremu i karte.

Slika 3.18 prikazuje korake u razvoju novog člana porodice aplikacija. Ponovno pregovaranje o zahtjevima je poželjno zato da bi se minimizirale promjene u postojećoj (najsličnijoj) verziji aplikacije. Ukoliko korisnik pristane na kompromise u zahtjevima, tada će nova verzija biti jeftinija i brže isporučena.



Slika 3.18: razvoj novog člana porodice apkikacija.

Uzastopni razvoj novih članova porodice aplikacija dovest će do postepenog kvarenja polazne arhitekture. U jednom trenutku softverska kuća mora se odlučiti na razvoj nove (bolje) generičke aplikacije. Ta nova aplikacija ne nasljeđuje programski kod nego samo znanje i iskustvo stečeno kroz razvoj polazne porodice aplikacija.

3.6.3 Obrasci za oblikovanje

Obrazac za oblikovanje (engleski *design pattern*) je apstraktni design. On definira uspješno rješenje za dobro poznati problem koji se često pojavljuje u raznim kontekstima. Ukoliko se susretnemo s takvim problemom, tada ga možemo riješiti primjenom odgovarajućeg obrasca. Na taj način dolazi do ponovne upotrebe, ali ne programskog koda nego designa. Rješenje koje ponovno upotrebljavamo oslobođeno je implementacijskih detalja, što može biti prednost budući da detalji stare implementacije mogu biti neprimjenjivi unutar nove.

Obrasci za oblikovanje nastaju kao pokušaj da se dokumentira skupljena mudrost i iskustvo u rješavanju problema. Skupljanje mudrosti prisutno je u gotovo svom granama računarstva: tradicionalne knjige o strukturama podataka i algoritmima također nastoje opisati provjerena rješenja za česte programerske probleme. Ipak, kad govorimo o obrascima za oblikovanje, tada mislimo na noviji pristup (Gamma et al, 1995) koji je nastao u okviru objektnog oblikovanja, gdje se rješenje izražava kao objektni model od nekoliko apstraktnih klasa.

U obrascu za oblikovanje moraju biti prisutna sljedeća četiri elementa.

1. Smisleno ime za obrazac.
2. Opis problema, iz kojeg je vidljivo kad se obrazac može primijeniti.
3. Opis rješenja, koji objašnjava dijelove predloženog designa, te odnose između tih dijelova. Opis rješenja obično se naslanja na dijagram (klase, nasljeđivanje, druge vrste veza, ...).
4. Opis posljedica, koji nabraja prednosti i mane primjene predloženog obrasca.

Kao primjer obrasca za oblikovanje, navodimo "Observer pattern" u Prilogu 3.36. Ovaj obrazac koristi se onda kad se traže različiti prikazi stanja istog objekta. Obrazac razdvaja objekt i njegove prikaze. To je ilustrirano Prilogom 3.37 gdje vidimo dva grafička prikaza istog skupa podataka. Rješenje unutar "Observer pattern" dano je objektnim modelom u Prilogu 3.38 nacrtanim prema pravilima UML. Rješenje je riječima objašnjeno u polaznom Prilogu 3.36.

Poglavlje 4

VERIFIKACIJA I VALIDACIJA

4.1 Općenito o verifikaciji i validaciji

Verifikacija i validacija (V&V) je skup aktivnosti kojima se nastoji utvrditi da li softver odgovara specifikaciji, te da li odgovara potrebama korisnika. Konačni cilj V&V je uvjeriti se da je softver dovoljno dobar i pouzdan da može poslužiti svojoj svrsi. To ne znači da softver mora biti sasvim bez greške, već samo da on mora biti dovoljno dobar i pouzdan za predviđeni oblik korištenja. Stupanj toleriranja grešaka ovisi o funkciji koju softver obavlja, očekivanjima korisnika, te o tržišnim uvjetima.

4.1.1 Razlika između verifikacije i validacije

Razlika između ova dva slična pojma je sljedeća.

- *Verifikacija* (Are we building the product right?) je provjera da li softver odgovara specifikaciji, dakle dokumentu o zahtjevima.
- *Validacija* (Are we building the right product?) je provjera da li softver zadovoljava "stvarnim" potrebama korisnika.

Razlika nastaje zato što specifikacija ne uspijeva u potpunosti zabilježiti stvarne potrebe korisnika.

4.1.2 Metode za verifikaciju i validaciju

Metode za verifikaciju i validaciju ugrubo se mogu podijeliti u sljedeće dvije skupine.

- *Statička V&V* (vidi Odjeljak 4.2). Svodi se na analizu i provjeru dokumenata, modela sustava, designa, odnosno programskog koda. Odvija se *bez* stvarnog izvođenja softvera.
- *Testiranje* (vidi Odjeljak 4.3). Svodi se na pokusno izvođenje softvera ili njegovih dijelova, na umjetno pripravljenim podacima, uz pažljivo analiziranje rezultata.

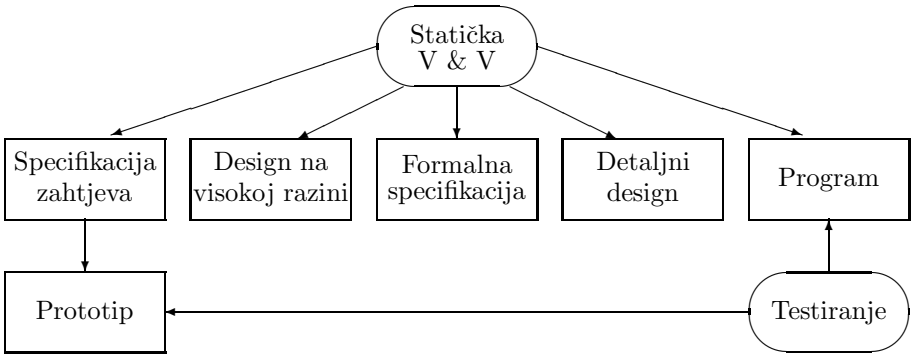
Testiranje je uobičajena metoda, no statička V&V ima sljedeće prednosti.

- Jedna statička provjera može otkriti puno grešaka. Testiranje obično od jednom otkriva samo jednu grešku budući da se nakon greške program "ruši" ili radi s krivim podacima.
- Statička provjera omogućuje bolje korištenje znanja o programiranju ili aplikacijskoj domeni. Osobe koje obavljaju provjeru znaju koje vrste grešaka se često pojavljuju u dotičnom programskom jeziku ili aplikaciji, te se koncentriraju na takve greške.

Većina autora smatra da se statičkim metodama može uz male troškove otkriti 60-90% grešaka, nakon čega testiranje ide brže, a ukupni troškovi su manji. Znači, dvije vrste metoda su komplementarne te se obje trebaju primjenjivati.

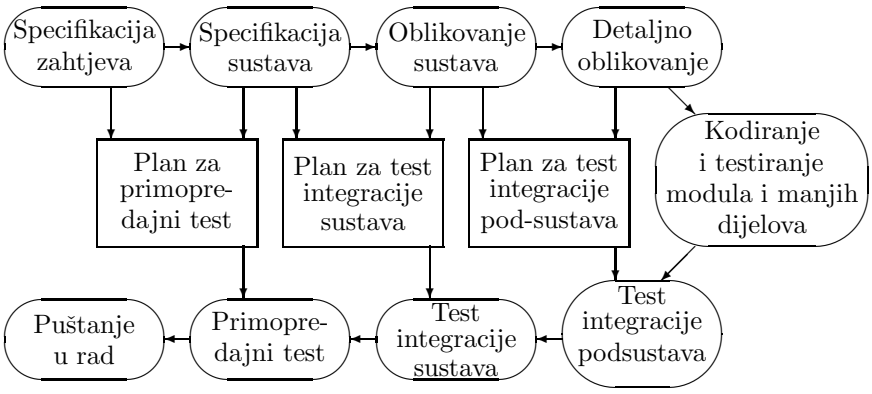
4.1.3 Mjesto verifikacije i validacije unutar softverskog procesa

V&V se prvenstveno primjenjuju na programski kod, no poželjno je da se u što većoj mjeri primjenjuju i u ranijim fazama razvoja softvera, kao što je prikazano na Slici 4.1. Vidimo da se statička provjera može obavljati u raznim fazama softverskog procesa, dok je testiranje primjenjivo samo na program odnosno prototip. Program se najprije provjerava statičkim metodama, a nakon toga se još podvrgava testiranju.



Slika 4.1: primjena verifikacije i validacije u raznim fazama razvoja softvera.

Da bi V&V aktivnosti dale očekivane efekte, planovi testiranja trebali bi nastati već u fazi specifikacije i oblikovanja sustava, kao što je prikazano na Slici 4.2. Plan za testiranje softvera je dokument sa sljedećim dijelovima: opis procesa testiranja, popis zahtjeva koji će se testirati, popis dijelova softvera koji će se testirati, kalendar testiranja, opis načina bilježenja rezultata testiranja, popis hardvera i softvera koji je potreban za testiranje, ograničenja koja utječu na proces testiranja.



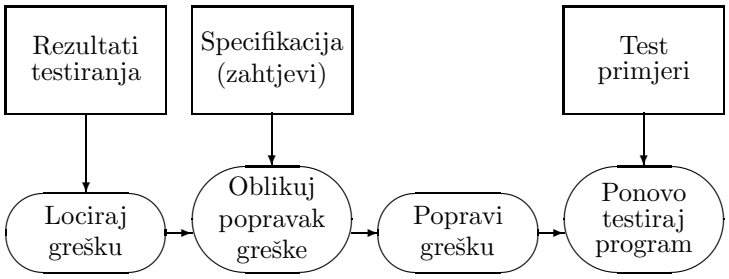
Slika 4.2: nastanak planova za testiranje softvera.

4.1.4 Odnos verifikacije, validacije i debuggiranja

Tijekom V&V otkrivaju se greške. Softver se mijenja da bi se greške popravile. Taj proces *debuggiranja* često je integriran s aktivnostima V&V. Ipak, V&V i debuggiranje su različiti procesi. Naime:

- V&V je proces koji utvrđuje *postojanje* grešaka u softveru.
- Debuggiranje je proces koji *locira* (otkriva uzrok) i *popravlja* te greške.

Detaljni opis procesa debuggiranja vidljiv je na Slici 4.3. Debuggiranje je kreativni proces kojeg je teško precizno opisati. Vješti programeri koji provode debuggiranje pouzdaju se u svoje iskustvo, poznavanje čestih grešaka i poznavanje svojstava korištenog programskog jezika, pa na taj način otkrivaju uzrok greške. Pritom programerima mogu biti od pomoći interaktivni debuggeri kakvi postoje u lower-CASE alatima.



Slika 4.3: proces debuggiranja.

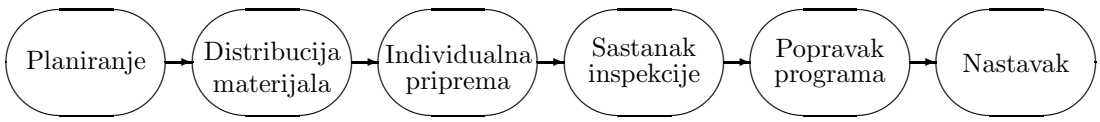
4.2 Statička verifikacija

Vidjeli smo da se statička V&V može obavljati u raznim fazama softverskog procesa. Ipak, u ovom odjeljku ograničit ćemo se na najčešći oblik takve V&V, a to je *statička verifikacija* programa. Ona se svodi na analizu i pregled programskog koda, bez stvarnog izvođenja programa. Primjenjuje se prije testiranja programa, da bi se otkrila glavnina grešaka. Nakog toga testiranje ide znatno brže, te su ukupni troškovi verifikacije znatno manji. U sljedeća tri pod-odjeljka obradit ćemo tri tehnike koje spadaju u statičku verifikaciju.

4.2.1 Inspekcija programa

Inspekcija programa prakticirala se u velikim firmama poput IBM i HP tijekom 70-tih, 80-tih i 90-tih godina 20. stoljeća. Riječ je o reguliranom procesu u kojem sudjeluje grupa ljudi sa strogo zadanim ulogama:

- *autor ili vlasnik*: programer odgovoran za program i za naknadni popravak grešaka;
- *inspektor*: pronalazi greške, propuste i nekonzistentnosti u programu;
- *čitač*: glasno čita programski kod na sastanku;
- *zapisničar*: bilježi rezultate sastanka;
- *moderator*: planira i organizira sastanke, upravlja procesom.



Slika 4.4: proces inspekcije programa.

Sam proces odvija se u skladu sa Slikom 4.4 Najvažnija faza je dobro pripremljen sastanak (do 2 sata) na kojem inspekcijska grupa pregledava program i otkriva greške. Na sastanku se ne raspravlja o tome kako će se greške popraviti, već je to naknadni posao autora. Moderator u fazi nastavka odlučuje da li je potrebno ponoviti cijeli proces. Da bi sastanak inspekcije uspio, važno je da:

- postoji precizna specifikacija (odgovarajućeg dijela) programa;
- postoji ažurna i dovršena verzija programskog koda;
- postoji "check-lista" čestih programerskih grešaka;
- članovi grupe su upoznati s organizacijskim standardima i žele surađivati.

Primjer moguće check-liste čestih programerskih grešaka vidi se u Prilogu 4.1.

4.2.2 Automatska statička analiza

Automatska statička analiza obavlja se pokretanjem softverskih alata koji prolaze kroz izvorni tekst našeg programa i otkrivaju moguće greške i anomalije u tom tekstu. Skreće se pažnja na "sumnjiva mjesta" u našem programu koja bi mogla sadržavati greške. Ispisuju se poruke. Sam popis sumnjivih situacija koje upućuju na grešku ovisi o programskom jeziku. Jedan mogući popis nalazi se u Prilogu 4.2.

Prilog 4.3 prikazuje seansu analiziranja (malog i besmislenog) C programa pomoću standardnog UNIX alata za statičku analizu `lint`. U gornjem dijelu priloga vidi se sam C kod; zatim se vidi da `cc` compiler ne javlja nikakve greške; nakon toga poziva se `lint`; dalje se vide `lint`-ove poruke (upozorenja) - na primjer varijable `c` i `i` bile su korištene prije nego što su bile inicijalizirane, a funkcija `printarray()` bila je pozivana s krivim brojem ili tipom parametara.

Nema sumnje da za jezike poput C statička analiza predstavlja efikasnu tehniku za otkrivanje programerskih grešaka. No u "strožim" jezicima poput Jave izbjegnute su neke jezične konstrukcije koje lako dovode do greške, na primjer sve varijable moraju biti inicijalizirane, nema `goto` naredbi (pa je teže stvoriti nedostupni kod), upravljanje memorijom je automatsko (nema pointera). Takav pristup sprečavanja grešaka umjesto njihovog kasnijeg otkrivanja pokazuje se efikasnim načinom poboljšavanja pouzdanosti programa. Zato je statička analiza u slučaju Java programa manje isplativa nego u slučaju C programa.

4.2.3 Formalna verifikacija

Formalna verifikacija svodi se na matematičko dokazivanje da je program konzistentan sa svojom specifikacijom. Moguća je samo pod sljedećim uvjetima:

- semantika programskog jezika je formalno definirana;
- postoji formalna specifikacija za program.

Dokaz se provodi tako da se polazni uvjeti iz specifikacije i same naredbe iz programa uzmu kao činjenice, pa se iz njih izvode konačni uvjeti koji po specifikaciji moraju biti zadovoljeni nakon izvršenja programa.

Idejom formalne specifikacije intenzivno su se bavila mnoga velika imena računarstva tijekom 70-tih godina 20. stoljeća (Hoare, Dijkstra, Wirth) - ipak nije se došlo dalje od malih akademskih primjera. Ideja je i dalje njeguje u krugovima pobornika formalnih metoda za razvoj softvera. U praksi se ova tehnika obično reducira na malo strožu inspekciju: razvijaju se strogi (no ne sasvim formalni) argumenti da program radi korektno. Primjer takvog pristupa koji uključuje strogu inspekciju je "Cleanroom Software Development" (IBM - 90-te godine 20. stoljeća - autor Mills i drugi).

4.3 Testiranje softvera

Testiranje je pokusno izvođenje softvera ili njegovih dijelova na umjetno pripravljenim podacima, uz pažljivo analiziranje rezultata. Svrha testiranja može biti verifikacija ili validacija. U ovom odjeljku ograničavamo se na testiranje u svrhu verifikacije, dakle na testiranje kojim se provjerava da li softver radi prema svojoj specifikaciji.

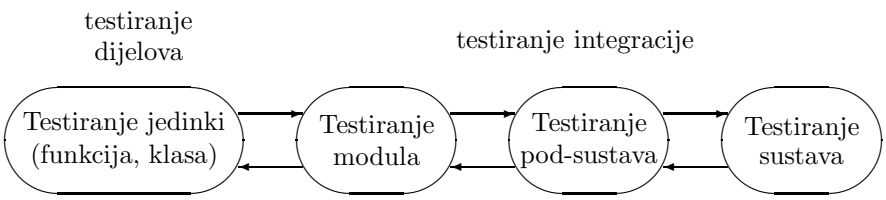
4.3.1 Vrste i faze testiranja

S obzirom na vrstu zahtjeva koje provjeravamo te način zadavanja test podataka, razlikujemo dvije vrste testiranja.

- *Testiranje grešaka.* Služi za provjeru funkcionalnih zahtjeva. Namjera je otkriti odstupanja između softvera i njegove specifikacije. Ta odstupanja su posljedica grešaka u softveru. Test-podaci su konstruirani tako da otkriju prisustvo greške, a ne tako da simuliraju normalnu upotrebu softvera.

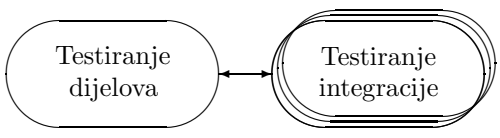
- *Statističko testiranje.* Služi za provjeru nekih oblika ne-funkcionalnih zahtjeva. Namjera je na primjer izmjeriti performanse softvera ili stupanj njegove pouzdanosti. Test-podaci su odabrani tako da liče na stvarne korisničke podatke, te da odražavaju njihovu stvarnu statističku razdiobu. Obično je riječ o velikom broju slučajno odabranih test primjera, što omogućuje statističku procjenu performansi ili pouzdanosti.

U nastavku ovog odjeljka ograničavamo se na testiranje grešaka. Takvo testiranje smatra se uspješnim onda kad ono pokaže da softver ne radi dobro. Nakon testiranja grešaka slijedi postupak debuggiranja (vidi Poglevlje 4.1). Primijetimo da testiranje može pokazati da u softveru postoje greške, no ne može dokazati da grešaka nema.



Slika 4.5: proces testiranja grešaka - detaljni prikaz.

Proces testiranja grešaka obično se sastoji od nekoliko faza, koje uzimaju u obzir da veliki sustav ima složenu građu. Detaljni prikaz faza vidi se na Slici 4.5. Sažetiji prikaz tih istih faza nalazi se na Slici 4.6. S jedne strane testiramo sastavne dijelove softvera, s druge strane testiramo da li su ti dijelovi na ispravan način integrirani u veće cjeline, te da li su te veće cjeline ispravno integrirane u još veće cjeline, itd. U fazi testiranja integracije, testiranje se fokusira na interakcije između dijelova, te na funkcionalnost cjeline. Ipak, često se dešava da se u toj fazi otkrivaju i greške u dijelovima. Zbog toga se faze mogu ponavljati. Testiranje dijelova obično provode osobe koje su razvile te dijelove, a testiranje integracije je obično posao nezavisnog tima.

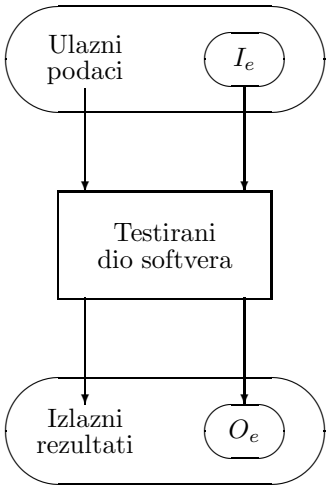


Slika 4.6: proces testiranja grešaka - sažeti prikaz.

4.3.2 Testiranje dijelova softvera

Da bi testiranje dijela softvera bilo uspješno, važno je znati odabrati test podatke. U nastavku ćemo promatrati nekoliko pristupa koji se razlikuju po načinu odabira test podataka.

Funkcionalno testiranje (engleski *black-box testing*). Test-primjeri izvode se isključivo iz specifikacije dotičnog dijela softvera. Ili, drukčije rečeno, softver se promatra kao "crna kutija" o kojoj jedino znamo da bi ona (prema specifikaciji) za zadane ulaze trebala proizvesti zadane izlaze. Zbog postojanja grešaka, ulazi iz određenog (nepoznatog) skupa I_e izazvat će neispravno ponašanje, koje ćemo prepoznati po izlazima iz skupa O_e. To je sve prikazano na Slici 4.7. U postupku testiranja nastojimo izabrati ulaze za koje postoji velika vjerojatnost da pripadaju skupu I_e. Izbor takvih test primjera najčešće se zasniva na iskustvu softverskog inženjera. Ipak, moguć je i sistematičniji pristup zasnovan na podjeli skupa svih mogućih ulaznih podataka (domene) na *klase*. Ovdje riječ "klasa" nije upotrebljena u smislu objektno-orijentiranog pristupa, nego u smislu matematičkog pojma "klasa ekvivalencije". Očekujemo da će se program ponašati slično za sve podatke iz iste klase. Biramo barem po jedan test primjer iz svake klase. Također je dobro isprobati "rubove" klasa.



Slika 4.7: funkcionalno testiranje grešaka (black box).

Kao ilustraciju, promatramo proceduru za traženje elementa `Key` u polju `T` koja bi trebala zadovoljavati specifikaciju iz Priloga 4.4. Procedura vraća odgovor `Found` da li je `Key` pronađen u `T`, te indeks `L` na kojem se `Key` nalazi u `T`. Iz same specifikacije, jasno je da domenu možemo podijeliti na dvije klase:

- ulazi gdje je `Key` zaista element polja `T` (`Found = true`);
- ulazi gdje `Key` nije element polja `T` (`Found = false`).

Dalje, iskustva u radu s poljima govore nam da treba probati:

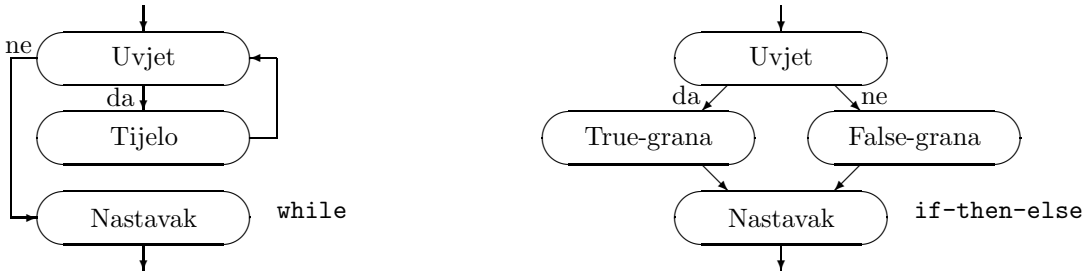
- polje `T` s ekstremnom duljinom, u ovom slučaju s jednim elementom;
- element `Key` na rubovima odnosno na sredini polja `T`.

Kad kombiniramo ova dva načina podjele domene na klase, dobivamo particiju iz Priloga 4.5 koja se sastoji od 6 klasa. Shodno tome, biramo 6 test-primjera koji se vide u istom Prilogu 4.5.

Strukturalno testiranje (engleski *white-box testing*). Dodatni test-primjeri izvode se na osnovu poznavanja interne strukture testiranog dijela softvera i korištenih algoritama. Dakle služimo se analizom programskog koda dotičnog dijela softvera da bi odabrali dodatne test primjere, tj. da bi bolje podijelili domenu na klase.

Kao primjer, pogledajmo u Prilogu 4.6 implementaciju u Javi naše procedure za traženje elementa u polju. Koristi se algoritam binarnog traženja. Specifikacija je sada malo stroža, naime polje mora biti uzlazno sortirano. Promatranjem programskog koda uočavamo da svaki korak algoritma dijeli polje na tri dijela. Zato osim izbora elementa `Key` na rubovima te na sredini polja `T` uvodimo i dva nova izbora: neposredno ispred i iza srednjeg elementa u polju - vidi Prilog 4.7. Nakon revizije starih test primjera (polje mora biti sortirano), te nakon dodavanja dvaju novih test primjera, dobivamo ukupno 8 test primjera iz Priloga 4.8.

Testiranje po putovima (engleski *path testing*). Riječ je o posebnoj vrsti strukturalnog testiranja, gdje se test primjeri biraju tako da se isproba svaki "nezavisni put" kroz dijagram toka kontrole testiranog dijela softvera. Dakle najprije se mora nacrtati dijagram toka kontrole dotičnog dijela softvera. Dijagram se crta tako da se naredbe `while` i `if - then - else` prikažu obrascima sa Slike 4.8, a ostale naredbe se prikažu svaka po jednim čvorom. Uočavaju se čvorovi koji predstavljaju "početak" i "kraj" toka kontrole. Testiraje po putovima zahtijeva da se konstruiraju takvi test primjeri koji će isprobati svaki od nezavisnih putova kroz dijagram. Nezavisni put je onaj koji ide od "početka" do "kraja" i prolazi bar jednim novim lukom.



Slika 4.8: obrasci za crtanje dijagrama toka kontrole.

Za Java implementaciju binarnog traženja iz Priloga 4.6 odgovarajući dijagram toka kontrole je u Prilogu 4.9. Brojevi čvorova na dijagramu odgovaraju brojevima redaka u Java kodu. Početak toka kontrole je čvor 1 a kraj je čvor 14. Postoje 4 nezavisna puta, i to:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14;
1, 2, 3, 4, 5, 14;
1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...;
1, 2, 3, 4, 5, 6, 7, 11, 13, 5, ...;.

Ako se u test primjerima prođe svaki od ovih putova, tada smo sigurni da je:

- svaka naredba bila izvršena bar jednom,
- svako grananje bilo isprobano za slučaj istine i za slučaj laži.

Maksimalni broj nezavisnih putova u dijagramu jednak je takozvanoj *ciklomatskoj složenosti* dijagrama (McCabe, 1976), koja se računa kao:

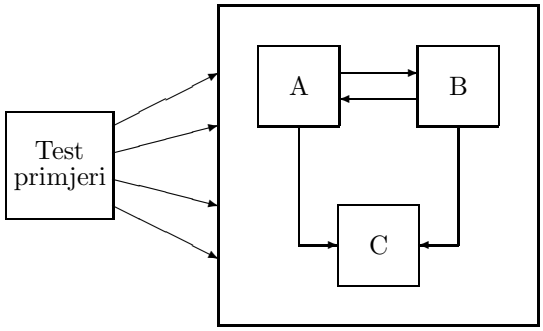
$$(\text{broj lukova}) - (\text{broj čvorova}) + 2.$$

Pokazano je da je ciklomatska složenost za program bez `goto` naredbi jednaka broju uvjeta u `if`, `while` i sličnim naredbama, plus 1.

U slučaju naše implementacije binarnog traženja lako bi konstruirali 4 test primjera koji odgovaraju uočenim nezavisnim putovima. No kod kompliciranijih rutina (mnogo grananja i petlji) to bi moglo biti znatno teže. Zato se pribjegava takozvanim *dinamičkim analizatorima* programa, koji mjere broj izvršavanja pojedine naredbe tijekom testiranja - na taj način otkrivamo koji dijelovi programa još nisu bili dovoljno testirani pa smišljamo nove test primjere.

4.3.3 Testiranje integracije

Testiranje integracije provodi se nakon što se manji dijelovi softvera udruže u veću cjelinu. Cilj testiranja je otkriti greške koje nastaju zato što dijelovi na pogrešan način koriste međusobna sučelja. U skladu sa Slikom 4.9, test primjeri se primjenjuju na cjelinu, a ne na pojedine dijelove.



Slika 4.9: primjena test primjera kod testiranja integracije.

Ova vrsta testiranja osobito je važna za objektno-oblikovane sustave. Naime, objekti su definirani preko svojih sučelja, te se ponovo upotrebljavaju u kombinaciji s drugim objektima u drugim sustavima - tada nastaju greške čiji uzrok je interakcija objekata, a ne rad pojedinog objekta.

Postoje različiti tipovi sučelja između udruženih dijelova. Nabrajamo četiri tipa.

- *Proceduralno sučelje.* Jedan dio softvera poziva proceduru koja pripada drugom dijelu softvera.
- *Parametarsko sučelje.* Podatak ili referenca na njega prenosi se kao parametar u pozivu procedure iz jednog dijela softvera u drugi.
- *Slanje poruka.* Jedan dio softvera zahtijeva uslugu od drugog dijela tako što mu pošalje poruku. Povratna poruka uključuje rezultat izvršene usluge.
- *Zajednička memorija.* Blok memorije dostupan je raznim dijelovima softvera. Jedan dio sprema podatke u blok, a drugi ih čita iz bloka.

Greške u korištenju sučelja mogu se podijeliti na sljedeće tri vrste.

- *Nerazumijevanje sučelja.* Na primjer kod proceduralnog sučelja, dio koji poziva rutinu za binarno traženje ne zna da polje mora biti sortirano.
- *Pogrešna upotreba sučelja.* Na primjer, krivi tip podatka u parametarskom sučelju.
- *Greške u timingu.* Na primjer, kod slanja poruka ili kod zajedničke memorije. Dijelovi koji proizvode odnosno troše podatke rade na različitim brzinama.

Testiranje integracije posebno je teško zato što se greške mogu pojaviti samo u iznimnim situacijama (veliko opterećenje sustava) ili pak na neočekivanim mjestima (u sasvim drugom dijelu softvera). Slijede općeniti naputci za testiranje integracije.

- Pregledaj programski kod svakog dijela softvera i popiši sve interakcije jednog dijela s drugim dijelovima.
- U slučaju proceduralnog sučelja probaj situaciju kad procedura vraća status greške.
- U slučaju parametarskog sučelja oblikuj testova koji koriste ekstremne vrijednosti parametara.
- Ako se prenašaju pointeri, isprobaj primjere s nul-pointerima.
- Ako se razmjenjuju poruke, probaj stvoriti znatno više poruka nego što se očekuje u normalnim okolnostima.
- Ako se koristi zajednička memorija, variraj redoslijed aktiviranja dijelova softvera koji pristupaju toj memoriji.

Softverski sustavi obično sadrže nekoliko razina integracije. Na primjer, najmanje jedinke udružuju se u module, koji se spajaju u pod-sustave, koji čine sustav. Zbog više razina integracije, testiranje integracije mora se ponoviti više puta. Pritom redoslijed testiranja može biti jedan od sljedeća dva.

- *Top-down:* najprije se testira integracija na najvišoj razini, zatim na nižoj razini, ..., na kraju na najnižoj razini. Pritom se nepostojeći ili netestirani dijelovi s niže razine zamjenjuju s nadomjescima - takozvanim "stub"-ovima. Stub ima isto sučelje kao odgovarajući dio ali vrlo ograničenu funkcionalnost.
- *Bottom-up:* najprije se testira integracija na najnižoj razini, zatim na idućoj višoj razini, ..., na kraju na najvišoj razini. Nisu potrebni stub-ovi jer za svaku iduću razinu integracije već imamo implementirane i testirane sve sastavne dijelove. Ipak, potrebni su test-driveri koji pozivaju manje dijelove softvera i simuliraju njihovu okolinu.

Dva redoslijeda testiranja integracije ilustrirana su u Prilogu 4.10.

Prednost top-down redoslijeda je da on ranije otkriva eventualne greške u arhitekturi. također, psihološka prednost top-down-a je da se relativno rano dobiva sustav koji radi (doduše s ograničenom funkcionalnošću) i koji se može pokazivati. Mana top-down-a je da se troši vrijeme na izradu stub-ova.

Prednost bottom-up redoslijeda je da se lakše uključuju gotove komponente. Također, prednost bottom-up-a je da arhitektura ne mora biti dovršena sve do zadnjeg časa. Mana bottom-up-a je da se troši vrijeme na izradu test drivera.

4.3.4 CASE alati za testiranje

Način na koji se odvija testiranje softvera obično je specifičan za određenu softversku kuću ili aplikaciju. Zato razvijači softvera nastoje stvoriti svoju vlastitu kombinaciju CASE alata za testiranje, služeći se kupljenim ili samostalno implementiranim komponentama. Uobičajeni alati koji se koriste za testiranje su sljedeći.

- *Test manager:* upravlja testiranjem tako da pokreće softver koji se testira za razne test podatke.
- *Generator test podataka:* generira test-primjere koji odgovaraju specifikaciji, i to izborom iz baze primjera ili generiranjem slučajnih vrijednosti korektnog oblika
- *Prorok:* daje prognozirane (očekivane) rezultate testa. Prorok može biti prethodna verzija softvera kojeg testiramo, ili prototip.
- *Komparator datoteki:* otkriva razlike između stvarnih i očekivanih rezultata testa.
- *Generator izvještaja:* oblikuje izvještaj o rezultatima testiranja.
- *Dinamički analizator:* broji koliko puta se pojedina naredba programa izvršila tijekom testiranja.
- *Simulator:* simulira na primjer ciljno računalo na kojem se testirani softver konačno treba izvršavati, ili simulira višestruke istovremene interakcije korisnika.

Ovakve kombinacije alata otvorene su za nadogradnju i adaptaciju te evoluiraju u skladu s potrebama.

Poglavlje 5

ODRŽAVANJE I EVOLUCIJA

5.1 Općenito o održavanju i evoluciji

Nakon puštanja u upotrebu, softverski sustav se i dalje mora mijenjati. Promjene softvera nužne su da bi se držao korak s novim korisničkim zahtjevima, promjenama poslovnog okruženja, napretkom hardvera itd.

Proces mijenjanja softvera nazivamo *održavanje* odnosno *evolucija*. Za neke autore ove dvije riječi su sinonimi, s time da je prva tradicionalna i posuđena iz tehnike, a druga je novija i točnija. Mi ćemo pod održavanjem podrazumijevati planirani rutinski dio procesa koji se sastoji od manje radikalnih promjena, dok će evolucija označavati ukupni proces s nepredvidivim elementima koji može dovesti do bitnih promjena u arhitekturi softvera i korištenoj tehnologiji.

5.1.1 Strategije mijenjanja softvera

Prema autoru Warren-u (1998), postoje sljedeće tri strategije koje se međusobno dopunjuju.

- *Održavanje softvera.* Promjene se rade zato da bi se držao korak s (promijenjenim) zahtjevima. Mijenja se funkcionalnost, no struktura softvera uglavnom ostaje ista.
- *Arhitekturna transformacija.* Značajna promjena arhitekture softverskog sustava. Na primjer, sustav se iz centralizirane mainframe arhitekture prebacuje u distribuiranu arhitekturu s klijentima i poslužiteljima. Funkcionalnost se može ili ne mora promijeniti.
- *Softversko re-inženjerstvo.* Ne dodaje se nikakva nova funkcionalnost. Također, nema velike promjene u arhitekturi. Umjesto toga, sustav se transformira u oblik koji se lakše može razumijeti i dokumentirati, te obrađivati raspoloživim alatima. Na primjer, programski kod se automatski prebacuje iz jednog (zastarjelog) programskog jezika u drugi (suvremeni).

Održavanje softvera je kontinuirani rutinski proces koji se svodi na niz uzastopnih promjena. Kao rezultat održavanja nastaje velik broj različitih verzija softvera, od kojih svaka predstavlja nešto drukčiju konfiguraciju sastavnih dijelova. Cijeli proces treba pažljivo planirati i kontrolirati. Također je neophodno imati točnu evidenciju svih verzija i njihovih konfiguracija. Skup potrebnih metoda, alata i organizacijskih zahvata naziva se *upravljanje konfiguracijom* i bit će opisan u Odjeljku 5.2.

Arhitekturna transformacija i softversko re-inženjerstvo su jednokratni i radikalni zahvati kojima se pribjegava onda kad održavanje više ne daje rezultata ili postane preskupo. Ti zahvati obično se primjenjuju na stari *"baštinjeni"* (engleski *legacy*) softver, zato da bi se takav softver osuvremenio te učinio pogodnijim za daljnje korištenje i mijenjanje. Nakon arhitekturne transformacije ili re-inženjerstva, transformirani softver dalje se podvrgava održavanju. Transformiranje baštinjenog softvera bit će obrađeno u Odjeljku 5.3.

5.1.2 Vrste održavanja softvera

Održavanje se sastoji od mijenjanja postojećih dijelova softvera, te po potrebi od dodavanja novih dijelova, bez bitnog narušavanja postojeće arhitekture. S obzirom na sadržaj i karakter promjena razlikujemo sljedeće vrste održavanja.

- *Korekcijsko održavanje.* Ispravljaju se uočene greške. Može se raditi o greškama u kodiranju (jeftino za ispraviti), u oblikovanju (skuplje), odnosno u specifikaciji (najskuplje).
- *Adaptacijsko održavanje.* Obavlja se prilagodba na novu platformu, na primjer drugi hardver, drugi operacijski sustav, ili drugu biblioteku potprograma. Funkcionalnost softvera nastoji se održati nepromijenjenom.
- *Perfekcijsko održavanje.* Implementiraju se novi funkcionalni ili ne-funkcionalni zahtjevi. Ti zahtjevi odražavaju povećane potrebe korisnika ili promjene u poslovnom okruženju.

Pojedina nova verzija softvera obično kombinira više vrsta promjena, no na managementu je da odluči kojoj vrsti održavanja će dati prednost.

5.1.3 Dinamika održavanja softvera

Autori Lehman i Belady (1985) objavili su empirijska istraživanja dinamike održavanja softvera. Istraživanja sugeriraju da vrijede sljedeći, takozvani Lehman-ovi "zakoni".

1. *Nužnost mijenjanja.* Softver koji se zaista koristi u stvarnom svijetu nužno se mora mijenjati jer u protivnom ubrzo postaje neupotrebljiv.
2. *Povećavanje složenosti.* Dok se softver mijenja, njegova struktura teži tome da postane sve složenija. Da bi se očuvala jednostavnost strukture, potrebno je uložiti dodatni trud i resurse.
3. *Ograničena brzina unapređivanja.* Količina "novosti" koju pojedino izdanje softvera može donijeti otprilike je konstantna i karakteristična je za taj softver.

Prvi i drugi Lehman-ov zakon dovoljno su razumljivi bez dodatnih komentara. Treći zakon znači da ako u jednom izdanju unesemo previše novosti, unijet ćemo i mnogo novih grešaka, pa će iduće izdanje morati biti posvećeno ispravljanju tih grešaka, a prosječna količina novosti po izdanju vratit će se na normalnu vrijednost. Konstanta iz trećeg zakona određena je originalnim procesom razvoja softvera i ne može se naknadno popraviti. Znači, ako je proces razvoja bio kvalitetniji, stvorit će se softver koji je pogodniji za održavanje, te će konstanta biti veća, to jest dopuštat će veću brzinu kasnijeg unapređivanja softvera.

5.1.4 Cijena održavanja softvera

Ukupna cijena održavanja softvera dostiže i prelazi cijenu njegovog originalnog razvoja. Cijenu održavanja možemo smanjiti opet tako da povećamo kvalitetu originalnog razvojnog procesa. Uz veću kvalitetu cijena razvoja će biti veća, ali će se stvoriti softver koji je jeftiniji za kasnije održavanje. Ova ideja ilustrirana je Slikom 5.1.

U nastavku nastojimo popisati osnovne faktore koji utječu na cijenu održavanja.

1. *Cjelovitost polazne specifikacije.* Ukoliko odmah uključimo sve zahtjeve, kasnije će biti manje perfekcijskog održavanja.
2. *Dobrota oblikovanja.* Dobar design je jeftiniji za održavanje. Smatra se da su sa stanovišta održavanja najbolji objektno-oblikovani sustavi, koji se sastoje od malih modula s jakom unutrašnjom kohezijom i labavim vezama prema van.

3. *Način implementacije.* Kod u "strožem" programskom jeziku poput Jave lakše se održava nego kod u jeziku poput C-a. Strukturirani kod (`if`, `while`) sa smisleno imenovanim varijablama razumljiviji je od kompaktnog koda s mnogo `goto` naredbi.

4. *Stupanj verificiranosti.* Dobro verificirani softver ima manje grešaka pa će zahtijevati manje korekcijskog održavanja.

5. *Stupanj dokumentiranosti.* Uredna, dobro strukturirana i cjelovita dokumentacija olakšava razumijevanje softvera, te na taj način pojeftinjuje održavanje.

6. *Način upravljanja konfiguracijom.* Ukoliko se primjenjunju metode, alati i organizacijska pravila upravljanja konfiguracijom, tada je održavanje na dugi rok jeftinije.

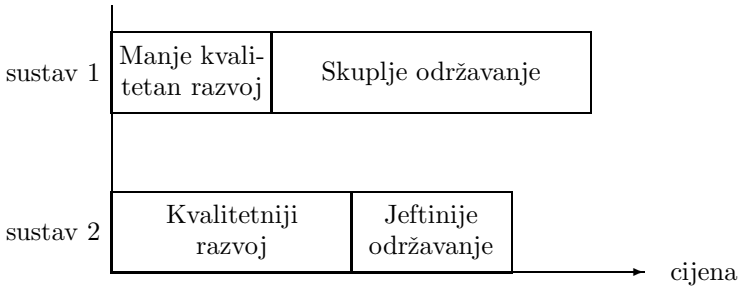
7. *Starost softvera.* Što je softver stariji, to je skuplji za održavanje, budući da mu se građa degradirala, ovisan je o zastarjelim razvojnim alatima, a dokumentacija mu je postala neažurna.

8. *Svojstva aplikacijske domene.* Ako je riječ o stabilnoj domeni gdje se poslovna pravila rijetko mijenjaju, tada će se rijetko pojavljivati potreba za perfekcijskim održavanjem u svrhu usklađivanja s novim pravilima.

9. *Stabilnost razvojnog tima.* Održavanje je jeftinije ako se njime bave originalni razvijači softvera, jer oni ne moraju trošiti vrijeme na upoznavanje sa softverom.

10. *Stabilnost platforme.* Ako smo softver implementirali na platformi koja će još dugo biti suvremena, tada neće trebati adaptacijsko održavanje.

Na prvih šest faktora moguće je utjecati tijekom originalnog razvojnog procesa.



Slika 5.1: utjecaj kvalitete razvojnog procesa na cijenu održavanja softvera.

5.2 Upravljanje konfiguracijom

Veliki softverski sustav može se promatrati kao konfiguracija svojih sastavnih dijelova. Za vrijeme svog života, sustav se mijenja. Nastaju različite verzije, od kojih svaka predstavlja nešto drukčiju konfiguraciju dijelova - vidi Prilog 5.1.

Upravljanje konfiguracijom (engleski *configuration management*) je organizirani proces koji kontrolira mijenjanje softvera i evidentira njegove različite verzije. Proces uključuje sljedeće četiri aktivnosti:

- izrada plana upravljanja konfiguracijom,
- upravljanje promjenama sustava,
- upravljanje verzijama sustava,
- gradnja sustava.

Svaku od ovih aktivnosti opisat ćemo opširnije u jednom od sljedećih pod-odjeljaka.

5.2.1 Izrada plana upravljanja konfiguracijom

Plan upravljanja konfiguracijom opisuje standarde i postupke koji se trebaju koristiti za upravljanje konfiguracijom. Riječ je o dokumentu koji sadrži sljedeća poglavlja.

- Popis entiteta koji će biti obuhvaćeni upravljanjem. Shema za identifikaciju tih entiteta.
- Odluka o tome tko je odgovoran za pojedine procese unutar upravljanja.
- Opis postupaka koji će se koristiti za upravljanje promjenama i upravljanje verzijama.
- Opis dokumenata koji će se generirati i pohranjivati tijekom upravljanja.
- Opis alata koji će se koristiti za upravljanje.
- Opis baze podataka za pohranjivanje informacija o konfiguracijama.

Shema za identifikaciju entiteta daje jedinstveno ime svakom dokumentu, modulu, ..., itd. koji je obuhvaćen upravljanjem konfiguracijom. Obično se koristi hijerarhijska shema kao u Prilogu 5.2. U tom prilogu riječ je o paketu `PCL-TOOLS` koji se sastoji od četiri programa: `COMPILE`, `BIND`, `EDIT` i `MAKE_GEN`. Svaki program dalje se sastoji od raznih modula, a za svaki modul na najnižoj razini pamtimo opis (OBJECT), programski kod (CODE) i skup testova (TESTS). Primjeri imena su:

```
PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE ,
PCL-TOOLS/EDIT/HELP/QUERY/HELPFRAMES/FR-1 .
```

Problem s ovakvom shemom je da isti entitet dobiva sasvim drugo ime ako se upotrijebi unutar drugog sustava. Također, otežano je pretraživanje po kriterijima koji su drukčiji od onih primijenjenih za klasifikaciju.

Baza podataka o konfiguracijama pohranjuje informacije koje su potrebne da bi se odredile konfiguracije pojedinih vezija softvera, te također i druge relevantne informacije. Baza mora biti u stanju dati odgovore na primjer na sljedeće upite.

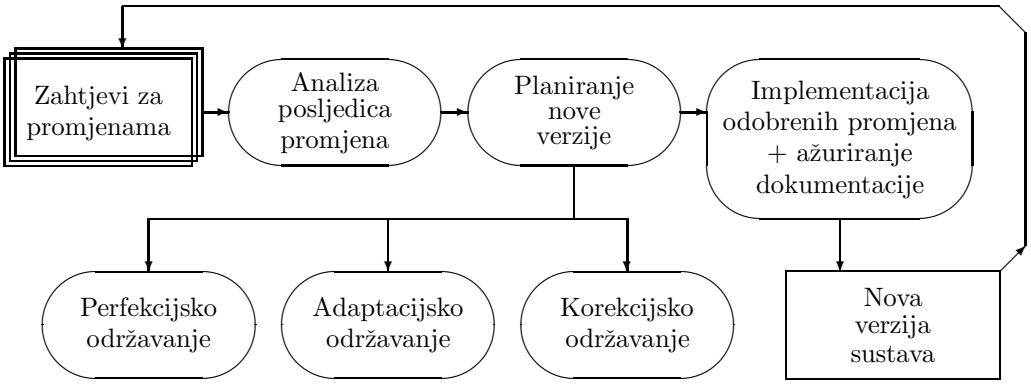
- Koliko verzija sustava postoji i koji su datumi njihovog nastanka?
- Od kojih verzija kojih dijelova se sastoji zadana vezija sustava?
- Koji kupci su instalirali zadanu verziju sustava?
- Koji hardver, operacijski sustav i biblioteke su potrebni za pokretanje zadane verzije sustava?
- Koji su neriješeni zahtjevi za promjenom zadane verzije sustava?
- Koje su greške prijavljene za zadanu verziju sustava?

Baza služi kao podrška za upravljanje promjenama, upravljanje verzijama, te gradnju sustava.

5.2.2 Upravljanje promjenama sustava

Upravljanje promjenama treba osigurati da se proces mijenjanja (održavanja) softvera drži pod kontrolom, da se promjene odvijaju na racionalan način, te da se sve promjene evidentiraju i dokumentiraju. Jedna nova verzija sustava obično sadrži veći broj promjena i nastaje u skladu sa Slikom 5.2.

Zahtjev za promjenom upisuje se u propisani obrazac - na primjer onaj iz Priloga 5.3. Analizu posljedica promjene obavlja imenovani analitičar. Odluku o tome koje promjene će se usvojiti i uključiti u novu verziju sustava donosi "change control board". Implementiranje skupa odobrenih promjena te ponovno testiranje sustava obavlja programerski tim za održavanje. Za svaki modul ili funkciju evidentira se "povijest" promjena - obično preko uvodnog komentara kao u Prilogu 5.4.



Slika 5.2: proces mijenjanja (održavanja) softvera.

5.2.3 Upravljanje verzijama sustava

Upravljanje verzijama omogućuje identificiranje i evidentiranje različitih verzija i izdanja istog sustava. *Verzija* je primjerak sustava koji se po nečemu razlikuje od ostalih primjeraka. *Izdanje* je verzija koja se isporučuje kupcima. Osim samih izvršivih programa, izdanje također sadrži konfiguracijske datoteke, datoteke s podacima, program za instalaciju, elektronički ili papirnati priručnik za korisnike, itd. Izdanje se distribuira na odgovarajućim medijima (CD, DVD, download s Interneta, ...).

Prvo što upravljanje verzijama mora osigurati je jednoznačna identifikacija svih verzija i njihovih sastavnih dijelova. Uobičajeni način identificiranja je pomoću brojeva, na primjer:

verzija 1.0, 1.1, 1.2, ..., 2.0, 2.1, ...

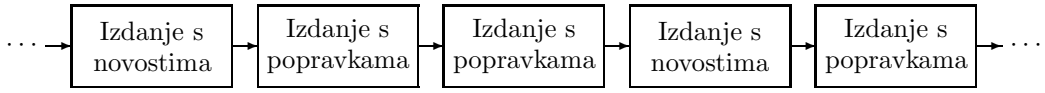
Druga mogućnost je pomoću atributa, na primjer:

program AC3D (jezik=Java, platforma=WinXP, datum=Jan2003).

Brojčani način identificiranja je češći, no on u sebi krije probleme jer implicira linearni redoslijed stvaranja verzija. Pravi redoslijed može biti kompliciraniji, kao na primjer onaj u Prilogu 5.5. Atributni način identificiranja je pogodniji za pretraživanje i on je obično implementiran "iznad" brojčane identifikacije, to jest baza podataka o konfiguracijama čuva veze između atributa koji opisuju verzije i brojeva odgovarajućih verzija sustava i njihovih dijelova.

Nakon identifikacije, upravljanje verzijama treba osigurati da se različite verzije sustava mogu reproducirati kad je to potrebno, te da se te verzije neće nepažnjom promijeniti. To ne mora značiti da svaka verzija mora biti eksplicitno pohranjena, već na primjer da postoji postupak kojim se izvorni kod tražene verzije može proizvesti iz neke pohranjene verzije.

Upravljanje verzijama također uključuje donošenje odluke koju od verzija treba pretvoriti u izdanje. Riječ je o osjetljivoj poslovnoj odluci. Naime, ako su izdanja prečesta, korisnici ih neće prihvatiti jer im instalacija iziskuje troškove. Ako su izdanja prerijetka, korisnici bi mogli odustati od našeg softvera i prijeći na alternativna rješenja. Lehman-ov treći zakon sugerira da nakon izdanja s novim mogućnostima (perfekcijsko održavanje) obično treba slijediti jedno ili dva izdanja kojima se popravljaju greške (korekcijsko održavanje). Takav ritam izdavanja ilustriran je Slikom 5.3.



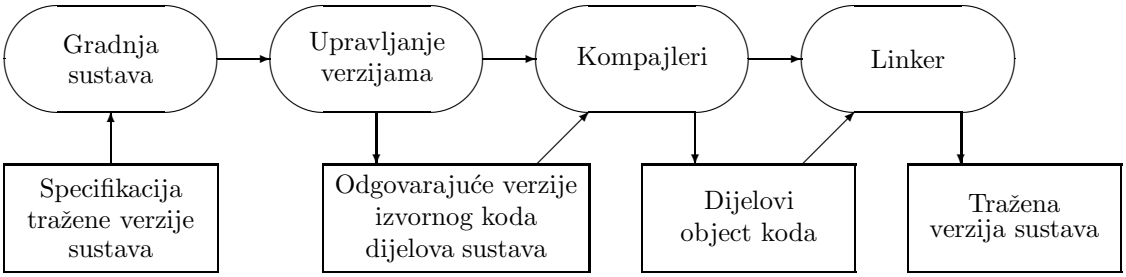
Slika 5.3: Uobičajeni ritam izdavanja softvera.

5.2.4 Gradnja sustava

Riječ je o postupku prevođenja i povezivanja dijelova sustava u sustav koji će se izvršavati na određenoj ciljnoj platformi. Stvari na koje treba obratiti pažnju kod gradnje su sljedeće.

- Da li su u instrukcije za gradnju uključeni svi potrebni dijelovi softvera (moduli, headeri, biblioteke, ...)?
- Da li instrukcije za gradnju uključuju pravu verziju svakog dijela softvera?
- Da li su dostupne sve potrebne datoteke s podacima?
- Ukoliko modul referencira datoteku, da li je upotrebljeno ime datoteke unutar modula isto kao ime na ciljnom stroju?
- Da li je još uvijek dostupna prava verzija compilera i ostalih alata?

Postupak gradnje ilustriran je Slikom 5.4.



Slika 5.4: postupak gradnje sustava.

5.2.5 CASE-alati za upravljanje konfiguracijom

Prva generacija alata davala je podršku samo za pojedine aktivnosti unutar upravljanja konfiguracijom. Primjeri alata prve generacije su: `sccs` (Rochkind, 1975) i `rcs` (Tichy, 1985) za upravljanje verzijama, te `make` (Feldman, 1979) za gradnju sustava.

Alati druge generacije kao na primjer `Lifespan` (Whitgift, 1991) i `DSEE` (Leblang and Chase, 1987) daju podršku za više aktivnosti, no još uvijek ne obuhvaćaju cijeli proces upravljanja konfiguracijom. Postoje i sasvim integrirani alati (Leblang, 1994), no oni su obično složeni i skupi, tako da održavatelji softvera ipak još uvijek koriste prvu i drugu generaciju. Microsoft-ov Visual Studio također daje prilično sveobuhvatnu podršku za upravljanje konfiguracijom, no isključivo za softver pisan za Windows platformu.

U nastavku ovog odjeljka opisat ćemo detaljnije kakvu vrstu podrške za pojedine aktivnosti možemo očekivati od današnjih alata.

Podrška za upravljanje promjenama. Alati za upravljanje promjenama obično su integrirani sa sustavom za upravljanje verzijama i sadrže sljedeće elemente.

- *Editor formulara* kojim se može mijenjati izgled zahtjeva za promjenama.
- *Workflow sustav* koji omogućuje da se definiraju ljudi koji će obraditi zahtjev za promjenu, te omogućuje da se definira redoslijed obrade - sustav šalje formulare i obavijesti pravim ljudima u pravo vrijeme elektroničkom poštom;
- *Baza podataka* koja čuva sve zahtjeve za promjenama i koja se može povezati sa sustavom za upravljanje verzijama.

Podrška za upravljanje verzijama. Alati za upravljanje verzijama pohranjuju razne verzije sustava i njihovih sastavnih dijelova. Uobičajene mogućnosti alata su sljedeće.

- *Identifikacija verzija (i izdanja).* Svakoj novoj verziji automatski se pridjeljuje identifikator, te eventualno dodatni atributi za pretraživanje.
- *Kontrolirano mijenjanje.* Da bi bio mijenjan, dio softvera mora eksplicitno biti "izvađen" iz repozitorija te premješten u neki radni direktorij. Kad se dio vrati u repozitorij, stvara se njegova nova verzija, a stara verzija i dalje postoji.
- *Efikasno pohranjivanje.* Budući da se verzije velikim dijelom podudaraju, pohranjuje se zapravo samo jedna "master" verzija, a ostale se opisuju tako da se zapišu razlike u odnosu na master.
- *Pamćenje povijesti promjena.* Sve promjene za zadani sustav ili njegov dio se pamte, te se mogu izlistati.
- *Nezavisan razvoj.* Različite verzije sustava mogu se razvijati paralelno, te se svaka verzija može mijenjati neovisno o drugim verzijama. Alat rješava konflikte koji mogu nastati kad više ljudi pokuša mijenjati isti dio.

Jednostavan primjer alata prve generacije za upravljanje verzijama na UNIX platformama je već spomenuti `rcs` (Revision Control System). Unutar `rcs`-a, izvorni kod zadnje verzije sprema se kao master, a ostale verzije definiraju se kao razlike (takozvane "delte") u odnosu na zadnju verziju, kao što se vidi u Prilogu 5.6. Podržani su samo ASCII masteri, a delte se opisuju kao skupovi komandi za editiranje na primjer u `vi` editoru. Dopušteni su i složeni redoslijedi verzija kao na primjer onaj u Prilogu 5.5.

Podrška za gradnju sustava. Alat za gradnju sustava automatizira postupak gradnje da bi smanjio mogućnost ljudske greške. Također, alat nastoji smanjiti posao tako da, na primjer, izbjegne nepotrebna kompiliranja ili povezivanja. Alat može biti samostalan ili može biti integriran s alatom za upravljanje verzijama. Poželjne su sljedeće mogućnosti.

- *Jezik za definiranje ovisnosti i pripadni interpreter.* Jezik omogućuje definiranje ovisnosti između različitih dijelova softvera, ili bolje rečeno ovisnosti između različitih datoteki od kojih se gradi sustav.
- *Podrška za izbor i pokretanje drugih alata.* Omogućuje se specificiranje compilera, linkera i drugih alata koji se koriste za obradu datoteki. Također se omogućuje pokretanje izabranih compilera ili linkera uz zadavanje opcija.
- *Distribuirano kompiliranje.* Neki alati za gradnju sustava u stanju su istovremeno pokrenuti kompiliranje različitih dijelova softvera na različitim umreženim računalima. Time se drastično smanjuje vrijeme potrebno za grednju sustava.
- *Upravljanje s izvedenim datotekama.* Izvedene datoteke su one koje su stvorene iz drugih (izvornih) datoteki primjenom odgovarajućih alata. Na primjer, object-datoteka se izvodi iz datoteke s izvornim kodom primjenom compilera. Upravljanje izvedenim datotekama omogućuje da se izvedena datoteka ponovo stvara jedino ako je to zaista potrebno zbog promjena u izvornoj datoteki.

Jednostavni primjer alata prve generacije za gradnju sustava na UNIX platformama je `make`. On povezuje program koji se sastoji od više modula, s time da prethodno re-kompilira samo one izvorne datoteke koje su bile mijenjane nakon svog zadnjeg kompiliranja. Korisnik piše skriptu (takozvani "makefile") koji opisuje veze između datoteki. Ažurnost pojedine datoteke utvrđuje se usporedbom datuma i vremena zadnje promjene. Prilog 5.7 sadrži dijagram građe nekog C programa, a u Prilogu 5.8 je pripadni makefile.

Glavna manjkavost `make`-a je ta što on nije povezan s odgovarajućim alatom za upravljanje verzijama, na primjer s `rcs`. Drugim riječima, `make` nije svjestan postojanja različitih verzija istog programa, već se korisnik sam mora brinuti da mu "podmetne" pravu verziju.

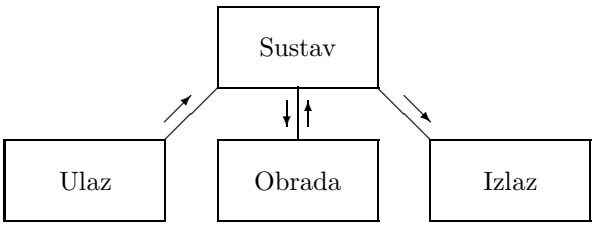
5.3 Baštinjeni softver i njegovo mijenjanje

U velikim organizacijama postoje takozvani *baštinjeni* (engleski *legacy*) sustavi. Riječ je o softveru razvijenom tijekom 70-tih ili 80-tih godina 20. stoljeća koji je u tehnološkom smislu zastario. Održavanje baštinjenih sustava izuzetno je skupo, koji put i nemoguće. S druge strane, organizacija se ne može tek tako odreći tog softvera, budući da je on neophodan za njeno svakodnevno funkcioniranje. Izlaz iz ove situacije nastoji se naći u jednokratnim radikalnim zahvatima kao što su softversko re-inženjerstvo odnosno arhitekturna transformacija. Tim zahvatima baštinjeni softver se naprije mijenja u oblik koji se lakše može razumjeti, dokumentirati i obrađivati suvremenim alatima. Nakon toga, softver se nastoji osuvremeniti i učiniti pogodnijim za daljnje korištenje i održavanje.

5.3.1 Građa i tehnološke osobine baštinjenog softvera

Baštinjeni sustav obično je originalno bio razvijen prije 20-30 godina, te je kasnije bio mijenjan. U originalnom razvoju koristila se neka od funkcionalno-orijentiranih metoda za razvoj softvera. Zato imamo sljedeće osobine.

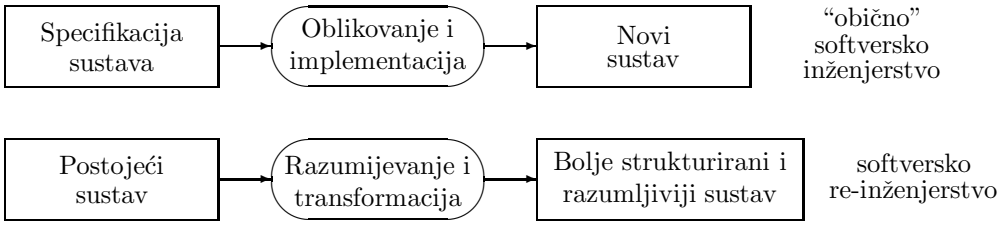
- Sustav se sastoji od više programa i zajedničkih podataka koje koriste ti programi. Podaci se čuvaju u datotekama ili pomoću (zastarjelog) sustava za upravljanje bazom podataka.
- Pojedini program unutar sustava oblikovan je funkcionalnim pristupom i sastoji se od funkcija koje komuniciraju preko parametara i globalnih varijabli.
- U slučaju poslovnih primjena, sustav se obično svodi na "batch" obradu ili obradu transakcija. U oba slučaja, opća organizacija je s skladu s modelom "ulaz-obrada-izlaz" koji je ilustriran Slikom 5.5.
- Sustav je implementiran u zastarjelom programskom jeziku kojeg današnji programeri ne razumiju, te za kojeg više nema compilera.
- Dokumentacija sustava je dijelom izgubljena ili neažurna. U nekim slučajevima jedina dokumentacija je izvorni programski kod. Ima slučajeva kad je čak i izvorni kod izgubljen te jedino postoji izvršiva verzija programa.
- Dugogodišnje održavanje pokvarilo je strukturu sustava, tako da ju je vrlo teško razumjeti.
- Izvorni kod pisan je s namjerom da se optimiziraju performanse ili smanje zahtjevi za memorijom, a ne s namjerom da kod bude razumljiv. To stvara poteškoće današnjim programerima koji su učili suvremene tehnike softverskog inženjerstva i nikad nisu vidjeli takav stil programiranja.



Slika 5.5: organizacija sustava u skladu s modelom "ulaz-obrada-izlaz".

5.3.2 Softversko re-inženjerstvo

Cilj softverskog re-inženjerstva je da se poboljša struktura sustava, te da se ona dovede u oblik koji se može lakše razumjeti. Očekuje se da će se na taj način smanjiti cijena daljnjeg održavanja sustava. Razlika između "običnog" softverskog inženjerstva i re-inženjerstva vidi se na Slici 5.6.

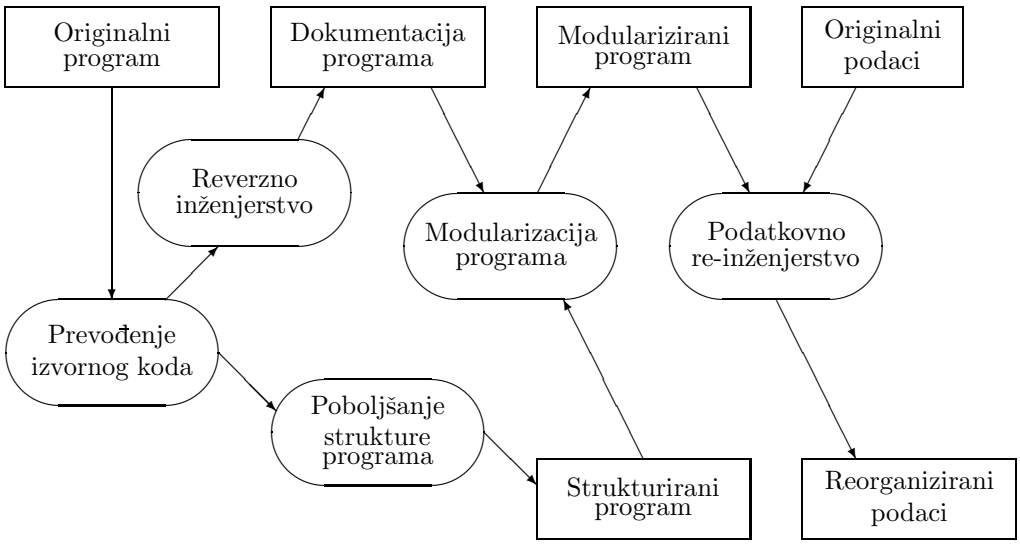


Slika 5.6: razlika između softverskog inženjerstva i re-inženjerstva.

Re-inženjerstvo uključuje sljedeće pod-aktivnosti.

- *Prevođenje izvornog koda.* Automatska (uglavnom) konverzija programa pisanog u jednom (zastarjelom) programskom jeziku u drugi (suvremeni) jezik. Ovaj zahvat je nužan ukoliko za zastarjeli jezik više nema compilera ili ako ga mladi programeri ne razumiju.
- *Reverzno inženjerstvo.* To je proces reproduciranja designa i specifikacije sustava na osnovu njegovog programskog koda. Potrebno je ukoliko je originalna dokumentacija izgubljena. Koriste se pomoćni alati, takozvani "program-browseri" kojima se obavlja "navigacija" kroz kod, na primjer pronalaženje mjesta gdje se koristi određena varijabla, i slično.
- *Poboljšanje strukture programa.* Uključuje na primjer zamjenu nestrukturiranih kontrolnih konstrukcija poput `goto` s ekvivalentnim `while` petljama i `if` naredbama. Postupak se može automatizirati. Nakon ovog zahvata, kod postaje uredniji i pogodniji za daljnje održavanje.
- *Modularizacija programa.* Svodi se na reorganizaciju izvornog koda, tako da se dijelovi koji su u međusobnoj vezi grupiraju zajedno. Nakon toga ti dijelovi postaju razumljiviji, pa ih je lakše optimizirati, ili je lakše ukloniti redundanciju.
- *Podatkovno re-inženjerstvo.* Potrebno je ukoliko programi na nekonzistentan ili zastarjeli način upravljaju podacima. Uključuje na primjer modifikaciju svih programa tako da umjesto vlastitih datoteki koriste zajedničku bazu podataka, ili na primjer prelazak sa zastarjele (hijerarhijske) baze na suvremenu (relacijsku) bazu.

Mogući slijed pod-aktivnosti unutar re-inženjerstva vidi se na Slici 5.7.



Slika 5.7: pod-aktivnosti unutar softverskog re-inženjerstva.

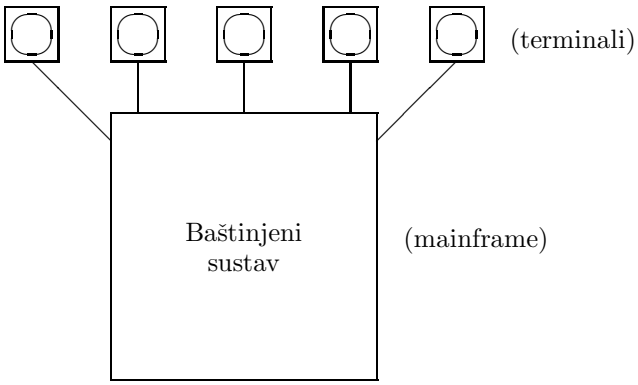
Kao primjer poboljšanja strukture programa pogledajmo Prilog 5.9. Riječ je o ne-strukturiranoj verziji kontrolera sustava za grijanje. Kod je gotovo nemoguće razumjeti zbog brojnih `goto` naredbi - takozvana špageti logika. Nakon re-inženjerskog zahvata, dobili smo znatno razumljiviju verziju u Prilogu 5.10. Ovakav zahvat moguće je obaviti uz pomoć automatskog alata, koji najprije konstruira dijagram toka kontrole originalne "`goto`" verzije, a zatim realizira taj dijagram uz pomoć `if` i `while` naredbi.

Prilog 5.11 ilustrira ideju o zamjeni datoteki sa zajedničkom bazom podataka. Programi u polaznom sustavu koristili su klasične datoteke u kojima je postojala redundancija i nekonzistentnost, i od kojih je svaka bila optimizirana za jednu aplikaciju a nepogodna za druge. Nakon re-inženjerskog zahvata imamo integriranu i konzistentnu bazu podataka koja osim postojećih aplikacija može podjednako dobro podržati i razne buduće aplikacije. Za ovakav zahvat potrebno je stvoriti logički model podataka, prepraviti sve programe tako da pristupaju bazi umjesto datotekama, te prebaciti podatke iz datoteki u bazu.

Softversko re-inženjerstvo ima svoja ograničenja. Na primjer, sustav razvijen funkcionalnim pristupom nije moguće pretvoriti u objektni sustav. Nakon re-inženjerstva softver je pogodniji za održavanje, no ipak se ne može tako dobro održavati kao novi softver. Podatkovno re-inženjerstvo može biti izuzetno skupo. Management treba procijeniti da li se re-inženjerstvo uopće isplati, ili je bolje krenuti u razvoj sasvim novog sustava.

5.3.3 Arhitekturna transformacija

Nakon re-inženjerstva, baštinjeni softver obično se podvrgava arhitekturnoj transformaciji, te se na taj način nastoji prilagoditi suvremenim tehnologijama. Tipični baštinjeni sustav je centralizirani sustav za rad na mainframe računalu i tekstualnim terminalima, prikazan na Slici 5.8.

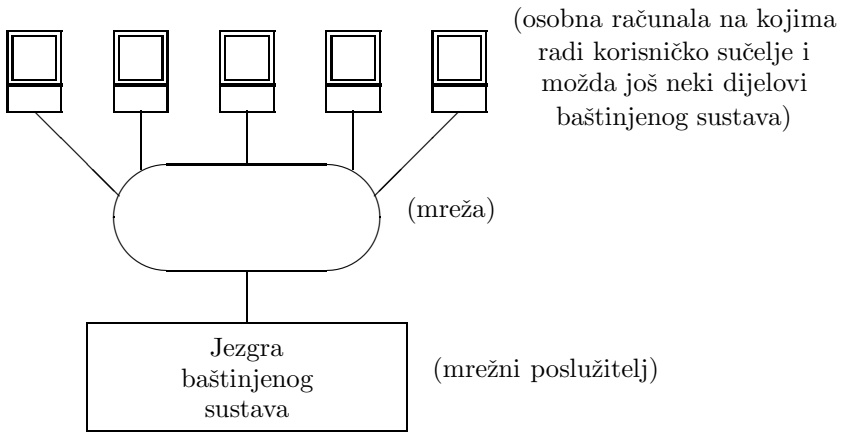


Slika 5.8: polazna arhitektura baštinjenog sustava.

Najčešći cilj arhitekturne transformacije je prelazak na arhitekturu klijent-poslužitelj, tako da se transformirani sustav može instalirati na mrežu sastavljenu od osobnih i poslužiteljskih računala u skladu sa Slikom 5.9.

Prelazak na arhitekturu klijent-poslužitelj je poželjan iz sljedećih razloga.

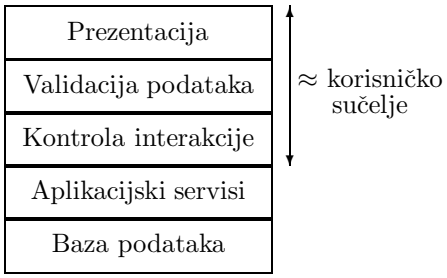
- *Cijena kupovine i održavanja hardvera.* Distribuirana računalna mreža znatno je jeftinija od mainframe računala ekvivalentne snage.
- *Očekivanja u pogledu korisničkog sučelja.* Tradicionalni mainframe sustavi daju sučelje s tekstualnim formularima. No korisnici danas očekuju grafičko sučelje i laganu interakciju sa sustavom. Grafičko sučelje zahtijeva znatno više lokalnog procesiranja i može se efikasno implementirati jedino na klijentima kao što su osobna računala ili grafičke radne stanice.
- *Distribuirani pristup sustavu.* Organizacije su sve više fizički distribuirane, pa računalni sustavi moraju biti dostupni s različitih vrsta opreme.



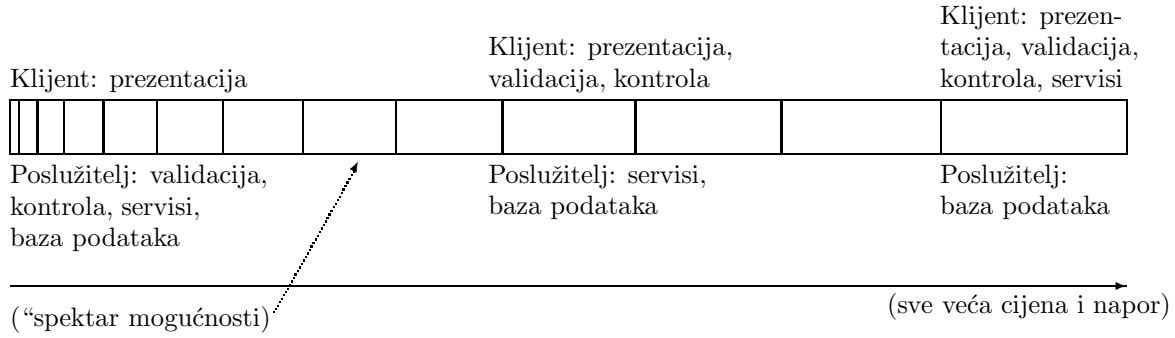
Slika 5.9: transformirana arhitektura baštinjenog sustava.

Kod planiranja arhitekturne transformacije, korisno je zamišljati da se naš polazni baštinjeni sustav sastoji od slojeva, kao što je prikazano na Slici 5.10. Pojedini slojevi obavljaju sljedeće zadaće.

- *Prezentacijski sloj* bavi se prikazom i organizacijom formulara za unos ili prikaz podataka.
- *Sloj za validaciju podataka* provjerava ispravnost podataka koje korisnik unosi u formular.
- *Sloj za kontrolu interakcije* upravlja redoslijedom korisnikovih operacija, te redoslijedom formulara koji se prikazuju korisniku.
- *Sloj s aplikacijskim servisima* obavlja obradu podataka i računanje.
- *Sloj s bazom podataka* omogućuje pristup podacima koji su pohranjeni u bazi.



Slika 5.10: slojevi unutar baštinjenog sustava.

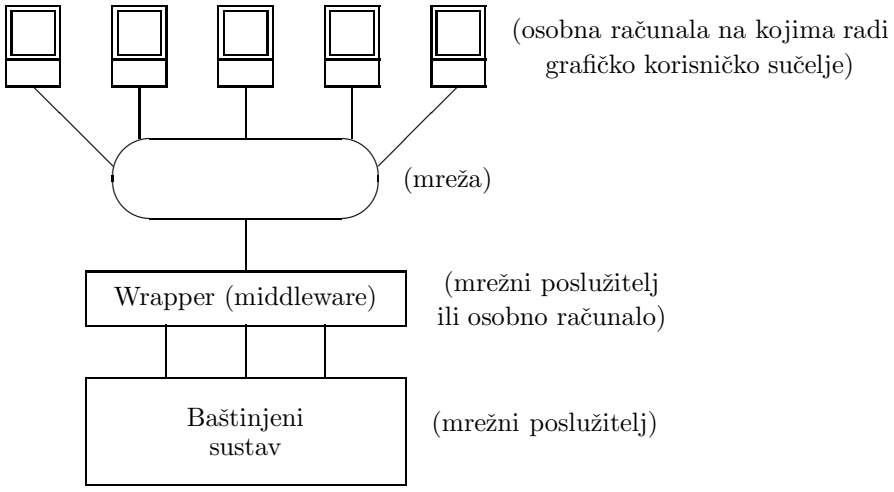


Slika 5.11: načini raspoređivanja slojeva baštinjenog sustava na klijente i poslužitelje.

Postoje razni načini disribuiranja sustava, koji se razlikuju po načinu kako su slojevi raspoređeni između poslužitelja i klijenata. Spektar mogućnosti ilustriran je Slikom 5.11. Slojevi koji su prebačeni na klijenta u pravilu se moraju mijenjati jer u protivnom nećemo moći iskoristiti prednosti grafičkog sučelja. Zato distribucije gdje je sve veći broj slojeva prebačen na klijenta postaju sve skuplje. Najjeftinije rješenje prebacuje samo prezentaciju na klijenta, no time ne dobivamo mnogo jer se klijent tada ponaša kao emulator terminala, pa sučelje i dalje slijedi logiku tekstualnih formulara. Najskuplje rješenje je da se sve osim baze prebaci na klijenta - tada se poslužitelj svodi na poslužitelja baze podataka i možda se može zamijeniti komercijalnim DBMS-om, čime bi se cijeli baštinjeni sustav izbacio iz upotrebe. Srednje rješenje je da na klijentu budu prezentacija, validacija i kontrola interakcije - ta tri sloja zajedno mogu se na najbolji način zamijeniti s novim grafičkim sučeljem koje više neće slijediti logiku tekstualnih formulara.

Opisani načini distribuiranja sustava pretpostavljali su da polazni sustav ima jasno odijeljene slojeve. U praksi to često nije slučaj, to jest slojeve nije moguće izdvojiti budući da su njihove funkcije pomiješane u raznim modulima. Tada se pribjegava nešto drukčijem vidu arhitekturne transformacije, koji je opisan Slikom 5.12. Transformirana arhitektura tada se sastoji od sljedeća tri dijela.

- Baštinjeni sustav, "zamrznut" u svom originalnom obliku, instaliran na poslužiteljskom računalu.
- Iznova razvijeno (grafičko) korisničko sučelje, koje radi kao klijent na osobnom računalu.
- Posebno razvijeni "wrapper", koji prevodi zahtjeve klijenata u interakcije s nepromijenjenim baštinjenim sustavom. Iz perspektive klijenta, wrapper izgleda kao poslužitelj. Iz perspektive baštinjenog sustava, wrapper izgleda kao skup tekstualnih teminala.



Slika 5.12: transformirana arhitektura baštinjenog sustava s wrapperom.

Spomenuta arhitektura s wrapperom opet dozvoljava razne varijante, koje se razlikuju po tehnologiji upotrebljenoj na strani klijenta. Neke od varijanti su sljedeće.

- Klijent je off-the-shelf emulator terminala. Wrapper je tada nepotreban ili vrlo jednostavan. Ovo je svakako najjeftinije no i najmanje kvalitetno rješenje. Naime, sučelje je zadržalo tekstualni oblik, makar je uklopljeno u prozor unutar grafičkog sučelja.
- Klijent je standardni web browser poput Microsoft Internet Explorer. Wrapper je tada web server poput Apache, nadopunjen servletima ili CGI-bin skriptama koje komuniciraju s baštinjenim sustavom i dinamički generiraju web stranice za klijenta. Ovo je prilično jeftino rješenje gdje postoji velika neovisnost o klijentskoj platformi. Ipak, u skladu s mogućnostima web browsera, grafičko sučelje je relativno skromno.
- Klijent je program razvijen nekim od alata za razvoj PC aplikacija, koji koristi sve mogućnosti grafičkog sučelja na osobnim računalima. Ovo rješenje je najkvalitetnije i najskuplje, te ima dodatnu manu da je vezano uz određenu klijentsku platformu poput Microsoft Windows.

Dodatak A

POPIS PRILOGA

U tablicama koje slijede, stranice i oznake iz udžbenika [1] u pravilu se odnose na aktualno 7. izdanje udžbenika, osim ako nije drukčije navedeno. Adrese s web-site-a autora udžbenika imaju zajednički početak koji glasi ovako: http://www.comp.lancs.ac.uk/computing/resources/IanS . Zbog preglednosti, taj početak je na svim mjestima pojavljivanja zamijenjen s tri točkice.

Oznaka u skripti	Mjesto i oznaka u udžbeniku	Adresa na web-u i redni broj slajda
Prilog 1.1	Strana 101 Slika 5.5	.../SE7/Presentations/PPT/ch5.ppt Slajd 20
Prilog 1.2	Strana 102 Slika 5.6	.../SE7/Presentations/PPT/ch5.ppt Slajd 21
Prilog 1.3	Strana 103 Slika 5.7	.../SE7/Presentations/PPT/ch5.ppt Slajd 22
Prilog 1.4	Strana 104 Slika 5.8	.../SE7/Presentations/PPT/ch5.ppt Slajd 23

Tablica A.1: prilozi uz Poglavlje 1.

Oznaka u skripti	Mjesto i oznaka u udžbeniku	Adresa na web-u i redni broj slajda
Prilog 2.1	Strana 119 Slika 6.1	.../SE7/Presentations/PPT/ch6.ppt Slajd 8
Prilog 2.2	Strana 139 Slika 6.17	.../SE7/Presentations/PPT/ch6.ppt Slajd 52
Prilog 2.3	Strana 174 Slika 8.3	.../SE7/Presentations/PPT/ch8.ppt Slajd 12
Prilog 2.4	Strana 173 Slika 8.2	.../SE7/Presentations/PPT/ch8.ppt Slajd 9
Prilog 2.5	Strana 180 Slika 8.8	.../SE7/Presentations/PPT/ch8.ppt Slajd 22
Prilog 2.6	Strana 176 Slika 8.5	.../SE7/Presentations/PPT/ch8.ppt Slajd 17
Prilog 2.7	Strana 178 Slika 8.6	.../SE7/Presentations/PPT/ch8.ppt Slajd 18 i 19
Prilog 2.8	Strana 179 Slika 8.7	.../SE7/Presentations/PPT/ch8.ppt Slajd 20
Prilog 2.9	Strana 184 Slika 8.10	.../SE7/Presentations/PPT/ch8.ppt Slajd 29
Prilog 2.10	Strana 184 Slika 8.11	.../SE7/Presentations/PPT/ch8.ppt Slajd 30
Prilog 2.11	Strana 185 Slika 8.12	.../SE7/Presentations/PPT/ch8.ppt Slajd 32
Prilog 2.12	Strana 186 Slika 8.13	.../SE7/Presentations/PPT/ch8.ppt Slajd 33
Prilog 2.13	Strana 187 Slika 8.14	.../SE7/Presentations/PPT/ch8.ppt Slajd 36
Prilog 2.14	Strana 225 Slika 10.7	.../SE7/Presentations/PPT/ch10.ppt Slajd 22
Prilog 2.15	Strana 228 Slika 10.8	.../SE7/Presentations/PPT/ch10.ppt Slajd 27 i 28
Prilog 2.16	Strana 46 Slika 3.1	.../SE7/Presentations/PPT/ch3.ppt Slajd 10
Prilog 2.17	6. izdanje Strana 206 Slika 9.11	.../SE6/Slides/PPT/ch9.ppt Slajd 33
Prilog 2.18	6. izdanje Strana 208 Slika 9.12	.../SE6/Slides/PPT/ch9.ppt Slajd 35
Prilog 2.19	6. izdanje Strana 208 Slika 9.13	.../SE6/Slides/PPT/ch9.ppt Slajd 37

Tablica A.2: prilozi uz Poglavlje 2.

Oznaka u skripti	Mjesto i oznaka u udžbeniku	Adresa na web-u i redni broj slajda
Prilog 3.1	Strana 253 Slika 11.5	.../SE7/Presentations/PPT/ch11.ppt Slajd 30
Prilog 3.2	Strana 255 Slika 11.6	.../SE7/Presentations/PPT/ch11.ppt Slajd 33
Prilog 3.3	5. izdanje Strana ? Slika 15.6	.../SE6/IG/functional-design.pdf Strana 8 Slika 15.6
Prilog 3.4	Strana 244 Slika 11.1	.../SE7/Presentations/PPT/ch11.ppt Slajd 10
Prilog 3.5	Strana 30 Slika 2.6	.../SE7/Presentations/PPT/ch2.ppt Slajd 27
Prilog 3.6	Strana 32 Slika 2.8	.../SE7/Presentations/PPT/ch2.ppt Slajd 29
Prilog 3.7	Strana 248 Slika 11.2	.../SE7/Presentations/PPT/ch11.ppt Slajd 19
Prilog 3.8	Strana 250 Slika 11.3	.../SE7/Presentations/PPT/ch11.ppt Slajd 22
Prilog 3.9	Strana 251 Slika 11.4	.../SE7/Presentations/PPT/ch11.ppt Slajd 25
Prilog 3.10	Strana 271 Slika 12.2	.../SE7/Presentations/PPT/ch12.ppt Slajd 13
Prilog 3.11	Strana 271 Slika 12.3	.../SE7/Presentations/PPT/ch12.ppt Slajd 14
Prilog 3.12	Strana 273 Slika 12.6	.../SE7/Presentations/PPT/ch12.ppt Slajd 21
Prilog 3.13	Strana 274 Slika 12.8	.../SE7/Presentations/PPT/ch12.ppt Slajd 24
Prilog 3.14	Strana 277 Slika 12.11	.../SE7/Presentations/PPT/ch12.ppt Slajd 30
Prilog 3.15	Strana 317 Slika 14.2	.../SE7/Presentations/PPT/ch14.ppt Slajd 11
Prilog 3.16	Strana 323 Slika 14.7	.../SE7/Presentations/PPT/ch14.ppt Slajd 30
Prilog 3.17	Strana 324 Slika 14.8	.../SE7/Presentations/PPT/ch14.ppt Slajd 32
Prilog 3.18	Strana 325 Slika 14.9	.../SE7/Presentations/PPT/ch14.ppt Slajd 33
Prilog 3.19	Strana 326 Slika 14.10	.../SE7/Presentations/PPT/ch14.ppt Slajd 35

Tablica A.3: prilozi uz Poglavlje 3 (prvi dio).

Oznaka u skripti	Mjesto i oznaka u udžbeniku	Adresa na web-u i redni broj slajda
Prilog 3.20	Strana 328 Slika 14.11	.../SE7/Presentations/PPT/ch14.ppt Slajd 40
Prilog 3.21	Strana 330 Slika 14.12	.../SE7/Presentations/PPT/ch14.ppt Slajd 45
Prilog 3.22	Strana 331 Slika 14.13	.../SE7/Presentations/PPT/ch14.ppt Slajd 47
Prilog 3.23	Strana 333 Slika 14.14	.../SE7/Presentations/PPT/ch14.ppt Slajd 49
Prilog 3.24	Strana 334 Slika 14.15	.../SE7/Presentations/PPT/ch14.ppt Slajd 51
Prilog 3.25	Strana 335 Slika 14.16	.../SE7/Presentations/PPT/ch14.ppt Slajd 54
Prilog 3.26	Strana 372 Slika 16.7	.../SE7/Presentations/PPT/ch16.ppt Sljad 22
Prilog 3.27	Strana 373 Slika 16.8	.../SE7/Presentations/PPT/ch16.ppt Slajd 24
Prilog 3.28	Strana 373 Slika 16.9	.../SE7/Presentations/PPT/ch16.ppt Slajd 25
Prilog 3.29	6. izdanje Strana 338 Slika 15.11	.../SE6/Slides/PPT/ch15.ppt Slajd 38
Prilog 3.30	Strana 376 Slika 16.11	.../SE7/Presentations/PPT/ch16.ppt Slajd 31
Prilog 3.31	Strana 376 Slika 16.12	.../SE7/Presentations/PPT/ch16.ppt Slajd 32
Prilog 3.32	6. izdanje Strana 343 Slika 15.16	.../SE6/Slides/PPT/ch15.ppt Slajd 52
Prilog 3.33	6. izdanje Strana 312 Slika 14.5	.../SE6/Slides/PPT/ch14.ppt Slajd 17
Prilog 3.34	Strana 434 Slika 18.13	.../SE7/Presentations/PPT/ch18.ppt Slajd 44
Prilog 3.35	Strana 435 Slika 18.14	.../SE7/Presentations/PPT/ch18.ppt Slajd 46
Prilog 3.36	Strana 423 Slika 18.5	.../SE7/Presentations/PPT/ch18.ppt Slajd 19
Prilog 3.37	Strana 424 Slika 18.6	.../SE7/Presentations/PPT/ch18.ppt Slajd 18
Prilog 3.38	Strana 424 Slika 18.7	.../SE7/Presentations/PPT/ch18.ppt Slajd 20

Tablica A.4: prilozi uz Poglavlje 3 (drugi dio).

Oznaka u skripti	Mjesto i oznaka u udžbeniku	Adresa na web-u i redni broj slajda
Prilog 4.1	Strana 526 Slika 22.7	.../SE7/Presentations/PPT/ch22.ppt Slajd 27, 28
Prilog 4.2	Strana 528 Slika 22.8	.../SE7/Presentations/PPT/ch22.ppt Slajd 31
Prilog 4.3	Strana 530 Slika 22.9	.../SE7/Presentations/PPT/ch22.ppt Slajd 34
Prilog 4.4	Strana 555 Slika 23.10	.../SE7/Presentations/PPT/ch23.ppt Slajd 39
Prilog 4.5	Strana 556 Slika 23.11	.../SE7/Presentations/PPT/ch23.ppt Slajd 42
Prilog 4.6	Strana 558 Slika 23.14	.../SE6/Slides/PPT/ch20.ppt Slajd 22
Prilog 4.7	Strana 557 Slika 23.13	.../SE7/Presentations/PPT/ch23.ppt Slajd 46
Prilog 4.8	Strana 559 Slika 23.15	.../SE7/Presentations/PPT/ch23.ppt Slajd 47
Prilog 4.9	Strana 560 Slika 23.16	.../SE7/Presentations/PPT/ch23.ppt Slajd 49
Prilog 4.10	6. izdanje Strana 453 Slika 20.14	.../SE6/Slides/PPT/ch20.ppt Slajd 34 i 35

Tablica A.5: prilozi uz Poglavlje 4.

Oznaka u skripti	Mjesto i oznaka u udžbeniku	Adresa na web-u i redni broj slajda
Prilog 5.1	Strana 691 Slika 29.1	.../SE7/Presentations/PPT/ch29.ppt Slajd 6
Prilog 5.2	Strana 694 Slika 29.2	.../SE7/Presentations/PPT/ch29.ppt Slajd 14
Prilog 5.3	Strana 697 Slika 29.4	.../SE7/Presentations/PPT/ch29.ppt Slajd 20
Prilog 5.4	Strana 699 Slika 29.5	.../SE7/Presentations/PPT/ch29.ppt Slajd 24
Prilog 5.5	Strana 701 Slika 29.6	.../SE7/Presentations/PPT/ch29.ppt Slajd 29
Prilog 5.6	Strana 709 Slika 29.9	.../SE7/Presentations/PPT/ch29.ppt Slajd 47
Prilog 5.7	Strana 710 Slika 29.10	.../SE7/Presentations/PPT/ch29.ppt Slajd 49
Prilog 5.8	-	http://www.math.hr/∼manger/si/Prilog_5.8.txt
Prilog 5.9	6. izdanje Strana 630 Slika 28.6	.../SE6/Slides/PPT/ch28.ppt Slajd 18
Prilog 5.10	6. izdanje Strana 630 Slika 28.7	.../SE6/Slides/PPT/ch28.ppt Slajd 19
Prilog 5.11	6. izdanje Strana 636 Slika 28.11	.../SE6/Slides/PPT/ch28.ppt Slajd 30

Tablica A.6: prilozi uz Poglavlje 5.

Literatura

[1] Sommerville I: *Software Engineering*, 7-th Edition. Addison-Wesley, Harlow, England, 2005. ISBN 0-321-21026-3. http://www.software-engin.com

[2] Van Vliet H.: *Software Engineering - Principles and Practice*, 2-nd Edition. John Wiley and Sons, Chichester, England, 2000. ISBN 0-471-97508-7. http://www.wiley.co.uk/vanvliet

[3] Pressman R.S.: *Software Engineering - A Practicioner's Approach*, 6-th Edition. McGraw Hill, New York, 2005. ISBN 0-07-285318-2.

[4] Schach S.R.: *Object Oriented & Classical Software Engineering*, 6-th Edition. McGraw Hill, New York, 2005. ISBN 0-07-286551-2.

[5] Pont M.J.: *Software Engineering with C++ and CASE Tools*. Addison-Wesley, Harlow, England, 1996. ISBN 0-201-87718-X.

[6] Grupa autora: *Argo UML - an UML Tool with Cognitive Support*. Open Source Software Engineering Tools. http://argouml.tigris.org/