Prof. dr Dragica Radosav

SOFTVERSKO INŽENJERSTVO I

Univerzitet u Novom Sadu
Tehnički fakultet "Mihajlo Pupin"
Zrenjanin, 2005.

0.UVOD

Pojam softversko inženjerstvo prvi put se pominje krajem šezdesetih godina XX veka, na konferenciji o krizi softvera, prouzrokovanoj trećom generacijom računara (prije svega IBM S/360), koja je svojom snagom inicirala razvoj velikih softverskih sistema. Pošto za ove sisteme nisu postojali odgovarajući teorijski modeli razvoja, realizacija projekata pojedinih informacijskih sistema je trajala izuzetno dugo, kasnila po više godina, što je višestruko uvećavalo planirane troškove razvoja, komplikovalo održavanje, a sama realizacija je bila na vrlo niskom nivou. Tako je počeo rad na teorijskim aspektima metodologija razvoja softvera, kasnije na njihovoj primjeni u praksi, da bi se danas jasno formirala disciplina koja proučava razvoj informacijskih tehnologija, poznata pod nazivom softversko inženjerstvo.

Pojmovno softversko inženjertsvo se može definisati kao stroga primena inženjeringa, naučnih i matematičkih principa i metoda u ekonomičnoj proizvodnji kvalitetnog softvera.

Softverski inžinjering je prvi definisao Fritz Bauer još 1968. godine na konferenciji koju je organizovao Naučni komitet NATO. Upotrijebio ga je kao termin koji označava primjenu principa inženjeringa u cilju ostvarivanja ekonomičnog softvera, efikasnog i pouzdanog u realnosti na računarskim mašinama. Nakon toga su nastale brojne definicije, ali su sve one posebno naglašavale potrebu primjene inženjeringa u razvoju softvera. Najpreciziju definiciju softverskog inženjeringa nalazimo u standardnom rečniku termina softverskog inženjeringa koji je 1990. godine objavio IEEE (Institute for Electronics, Energetics and Engineering), a prema kojem ***softverski inženjering podrazumjeva primjenu sistematičnog, disciplinovanog i merljivog pristupa razvoju, uvođenju i održavanju softvera***, *tj. primjeni inženjeringa na softver.*

Softverski inženjering je posledica hardverskog i sistemskog inženjeringa. Strukturu softverskog inženjeringa sačinjavaju tri ključne komponente: ***metode***, ***alati*** i ***postupci*** (procedure). Njihovo jedinstvo opredjeljuje kvalitet razvoja, pa je zbog toga zna-

čajno da se odaberu one komponente koje se postavljenim zadacima i problemima u razvoju najlakše prilagođavaju. Donošenje prave odluke nije jednostavan zadatak zbog:

- individualnog i kreativnog karaktera postupka projektovanja,
- različitosti pojedinih sistema za koji se razvija softver i
- različitog okruženja sistema.

Značajno je istaći da se za rešavanje konkretnog zadatka mogu izabrati najpogodnije komponente samo ukoliko projektanti poznaju njihove mogućnosti, alternative i znaju ih uspešno primeniti.

1. TIPOVI I KARAKTERISTIKE SOFTVERSKIH PROIZVODA (Types and characteristics of software products)

Softverski proizvod (engleski: consumer software package) predstavlja softversku podršku računarskim sistemima, a čine ga skup računarskih programa, datoteke i odgovarajuća dokumntacija, namenjeni realizaciji određenih zadataka (definicija preuzeta iz standarda JUS ISO 9127/94). Za razliku od softverskog proizvoda, softver se sastoji iz skupa računarskih programa i datoteka sa istom namenom koju ima softverski proizvod. Kako su, prema usvojenim standardima, softver i odgovarajuća prateća dokumentacija nerazdvojivi, upakovani za prodaju kao jedinstvena celina i prodaju se zajedno, u nastavku se ova dva pojma poistovećuju.

Iz izloženog se vidi da softverski proizvod predstavlja jedinstvo računarskih programa, struktura podataka sadržanih u datotekama i dokumentacije koja opisuje način funkcionisanja i upotrebe programa. Pri tome, posebno se ističe značaj dokumentacije, koju projektanti i programeri često zanemaruju, ali bez koje se softver ne može smatrati kompletnim. Informacije sadržane u pratećoj dokumentaciji su često jedino sredstvo putem koga proizvođač softvera i/ili distributer mogu komunicirati sa kupcem ili korisnikom. Samo ukoliko dokumentacija sadrži dovoljno informacija, korisniku se omogućuje uspešno korišćenje softvera.

Za pojam softvera vezan je i pojam ***softverska podrška***. Ona predstavlja rad na održavanju softvera i prateće dokumentacije u funkcionalnom stanju. Mogu je pružati: proizvođač (organizacija koja je razvila softver), predstavnik (organizacija koja plasira softver na tržištu), distributer (organizacija koja neposredno prodaje softver korisniku) ili neka druga organizacija.

Trend projektovanja softverskih proizvoda na osnovu elemenata ranijih aplikacija nezavisno razvijenih, sa različitim alatima, u različitim jezicima i na različitim platformama, postaje sve izraženiji. Ove, prethodno testirane komponente, pružaju mogu-

ćnosti jednostavnijeg i pouzdanijeg sklapanja u softverski proizvod, zatim njihovo višestruko korištenje i nezavisno poboljšanje mogućnosti i performansi.

Pojam ***softverskih komponenti*** podrazumeva sistemski ili aplikativni softver pomoću kojeg se upravlja fizičkim ili logičkim resursima sistema. Da bi se softverske komponente mogle koristiti pri navedenom načinu projektovanja potrebno je da zadovoljavaju jedinstven model razmene podataka međusobno, da postoji mogućnost neposrednog programiranja aktivnosti koje se odvijaju između komponenti, zatim da se u istoj datoteci mogu naći različiti podaci i da se mogu izvršavati na različitim platformama.

Skladišta softverskih komponenti koje zadovoljavaju opisane uslove i mogu se višestruko koristiti u različitim programima nazivaju se repozitorijumi. Pri formiranju, izmenama i upotrebi komponenti iz repozitorijuma treba voditi računa o tome da li nova verzija utiče i kako utiče na izvršenje pojedinih programa koji je koriste. Takođe je potrebno pratiti proces zamene komponenata novim verzijama u pojedinim programima i pri tome voditi računa o prilagođavanju komponente softverskom proizvodu.

Softverske komponente mogu se klasifikovati u sledeće funkcionalne grupe:

- seniorske (skupljaju informacije iz okoline sistema),
- pokretačke (prouzrokuju izmene u okolini sistema),
- računske (na osnovu određenih ulaza proračunavaju izlaze),
- komunikacijske (omogućavaju komunikaciju ostalih softverskih komponenti),
- koordinacijske (koordiniraju operacije ostalih komponenti) i
- interfejsi (transformišu prezentaciju izlaza jedne komponente u oblik razumljiv drugoj).

Tehnologije grupisane oko servisa WWW (World Wide Web) na Internetu svojim snažnim razvojem nametnutim opštom primenljivošću, a zasnovane na specifičnim funkcijama WWW interfejsa, otvorile su prvo pravo tržište gotovih softverskih komponenti razli-

čitih proizvođača. Ove komponente karakteriše standardizacija okruženja koja je nezavisna od platforme i operativnog sistema na kojima se one koriste. Primenom ovih komponenti ubrzava se razvoj novih, čime učešće gotovih komponenti postaje dominantno u realizaciji distribuiranih informacijskih sistema.

Drugi poznati primer softverskih komponenti predstavljaju takozvani šabloni (design patterns) koji se primenjuju u objektno orjentisanom dizajnu i programiranju. Pod ovim terminom podrazumevaju se tipske uloge, strukture, odnosi i mehanizmi obuhvaćeni jedinstvenim rešenjem koje je zajedničko za više softverskih proizvoda. Vrlo je popularna literatura u kojoj su dati i detaljno opisani brojni šabloni koji se mogu efikasno koristiti u razvoju objektno orjentisanih programa. Za njihov opis, pored imena, potrebno je navesti i niz drugih specifikacija kao što su *namena*, *primenljivost*, *struktura*, *učesnici* i *njihove veze*, *implementacija*, *primeri korištenja* i drugo.

Razlikujemo dve klase softverskih proizvoda:

- ***generički proizvodi*** su samostalni sistemi koje je izradila razvojna organizacija i prodaje ih na otvorenom tržištu zainteresovanim kupcima,
- ***proizvodi po meri*** su sistemi specijalno formirani za određenog kupca.

Do 1980.godine, dominirali su proizvodi po meri, koji su se izvodili na velikim računarima i bili su skupi, jer je kompletan razvoj plaćao jedan kupac. Tržište personalnih računara donelo je tržišnu prevagu generičkih proizvoda, koji se prodaju u hiljadama primeraka i zbog toga su znatno jeftiniji. Ipak, još uvek se više napora ulaže u izradu proizvoda po meri. Integrisana aplikativna rešenja sa značajnim referencama i znatno nižim cenama utiču na to da se danas tržište sve više orjentiše na generičke proizvode.

Danas se razlikuje više različitih klasifikacija softverskih proizvoda po raznim kriterijumima. U ovom tekstu biće data njihova klasifikacija prema funkcijama. Svrha ove klasifikacije je grubi prikaz najvećeg dela onoga što se na tržištu podrazumeva pod

pojmom softverski proizvodi. Prema ovoj klasifikaciji, na prvom nivou razlikujemo:

- softverske proizvode opšte namene i
- softverske proizvode posebne namene.

Daljom podelom, kod *softverskih proizvoda opšte namene* razlikujemo sledeće grupe:

- operativne sisteme,
- sistemske programe i
- kontrolne i dijagnostičke programe.

Softverski proizvodi posebne namene dele se na:

- aplikativne programe i
- korisničke softverske proizvode.

Tabela 1.1.: Klasifikacija softverskih proizvoda prema funkcijama, [2,16]

Softverski proizvodi (SP)				
SP opšte namene		SP posebne namene		
Operativni sistemi	Sistemski programi	Kontrolni i dijagnostički programi	Aplikativni programi	Korisnički softverski proizvodi
-opšte namene -real-time -transakcioni -multi procesorski	-asembleri -revodioci -punjači -vezni -spuleri -translatori	- debageri - drajveri - pristupne rutine	- SRBP - CASE - GUI - Internet alati - OLAP	- CAD - CAP - CAM - CAQ - PPS - MIS

Operativni sistemi se mogu podelititi na više načina. Tako razlikujemo *operativne sisteme sa prividnom memorijom* (virtual

storage) i *operativne sisteme bez prividne memorije*, zatim *operativne sisteme vezane za određeni tip računara* i *relativno nezavisne operativne sisteme*. Može se izvršiti dalja podela na sledeće klase operativnih sistema: operativni sistemi opšte namene, operativni sistemi za rad u realnom vremenu, transakcioni operativni sistemi, multiprocesorski operativni sistemi i mrežni operativni sistemi.

Sistemski programi predstavljaju prvi nivo korisničkih alata, neophodnih za razvoj i korišćenje drugih softverskih proizvoda. Mogu se podeliti na sledeće klase: asembleri, softverski prevodioci, punjači, vezni programi, spuleri i translatori.

Kontrolni i dijagnostički programi predstavljaju grupu pomoćnih programa koja sadrži sledeće klase: debagere, drajvere i pristupne rutine.

Aplikativni programi predstavljaju skup programskih biblioteka, alata i razvojnih produkata koji se koriste pri razvoju softverskog proizvoda u njegovoj redovnoj produkciji kao pomoćno sredstvo. Njih čine sledeće grupe: komunikacijski programi, stono izdavaštvo, sistemi za rukovanje bazama podataka, generatori programa, CASE alati, hipertekst, sistemi veštačke inteligencije, sistemi za obradu teksta, sistemi spregnutih tabela, grafički interfejsi, translacijski softver, alati za testiranje, menadžerski softverski alati, alati za definisanje zahteva, internet serveri, internet čitači, multimedijalni sistemi, OLAP sistemi, integrisani softverski proizvodi i drugo.

Korisnički softverski proizvodi razvijaju se za tačno određenog korisnika ili za grupu određenih, odnosno potencijalnih korisnika. Ovim softverskim proizvodima automatizuju se procesi iz pojedinih funkcija sistema konkretnog korisnika, kao što su knjigovodstvo, finansiranje, skladišno poslovanje, proizvodnja, pružanje usluga, nabavka i prodaja, upravljanje ljudskim resursima i drugo.

Bitna strategija projektovanja korisničkih softverskih proizvoda naziva se CIM (Computer Integrated Manufacturing), koja se pojavljuje prvi put 1985. godine na sajmu informatike u Hanoveru. Predstavlja integralni pristup projektovanju softvera jedne organizacije, koji se sastoji iz sljdećih modula: CAD (Computer Aided Design), CAP (Computer Aided Planning), CAM (Computer Aided Manufacturing), CAQ (Computer Aided

Quality) i PPS (Pro-duction Planning System), dok se, kao prateći, skoro uvek poja-vljuje MIS (Management Information System), uz niz drugih opcionih modula.

1.1. Paradigme arhitekture softvera (Programming paradigms)

Arhitektura softvera se menja tokom vremena. Za promene postoji veći broj razloga, a osnovne treba tražiti među onima koji bitno utiču na troškove razvoja i eksploatacije sistema. Promene su evulucionog karaktera sa nekoliko karakterističnih paradigmi koje se međusobno nadovezuju. Analiza koja sledi može pomoći otkrivanju trenda na osnovu kojeg možemo sa više izvesnosti govoriti o budućoj evoluciji arhitekture softvera.

Ako se pođe od stava da osobine programske podrške objedinjuju arhitekturu sistema, onda se može konstatovati da je u drugoj polovini XX veka evolucioni put arhitektura određen sa svega nekoliko vladajućih paradigmi primene računara. Pojava novih paradigmi dovodila je do skoka ne samo u pogledu kvaliteta, nego i poboljšanja performansi, kako u fazi razvoja, tako i u fazi eksploatacije informacionih sistema. Imajući u vidu rezultate date u [4], u ovom poglavlju će se izložiti jedan od mogućih pristupa u prikazu osnovnih karakteristika evolucije informacionih sistema.

Monolitna programska podrška

Ova faza evolucije arhitekture se može locirati na period od 1948. do 1965.godine, a dominantne karakteristike informacionih sistema (IS) zasnovanog na primeni računara bile su pre svega određene tadašnjim računarskim hardverom koji je bio izuzetno skup. Cena personala koji je koristio računar po jedinici vremena u odnosu na ekvivalentne troškove hardvera, skoro da se mogla zanemariti. Softverska podrška, kao interfejs između korisnika i mašine, bila je pre svega prilagođena zahtevima hardvera. Korisnik je morao da "misli poput mašine". Bila je to faza monolitne programske podrške u kojoj je izvršni kod koji proizilazi iz algoritma rešenja problema "srastao" sa pogonskim rutinama ulazno-izlaznih uređaja i monitorskim programom.

Rešavanje nekog problema se tretiralo kao kreativan, unikatan rad programera. Podrška je pisana na mašinskom ili asemblerskom jeziku, sa osnovnim zadatkom da se program izvrši u što kraćem vremenu. Korisnik je bio osoba školovana za rad sa konkretnim sistemom. Imajući u vidu činjenicu da korisnik sa računarskim sistemom, namenjenim da vrši masovna izračunavanja, može da komunicira isključivo posredstvom korisničkog interfejsa, u prvoj fazi evolucije interfejs se dizajnirao u stilu "spartanskog" komfora. Cena korisničkog interfejsa, tipične električne pisaće mašine u funkciji konzole, bila je zanemarljiva u odnosu na cenu hardvera.

Dihotomija aplikacija- Sistemski softver

Proizvođači hardvera da bi pospešili prodaju svojih mašina počeli su potencijalnim korisnicima da nude ne samo hardver već i deo programske podrške koji se pokazao kao obavezan pri eksploataciji računarskog sistema. Ovaj deo gotove programske podrške poznat je pod imanom sistemski softver i lociran je kao interfejsni sloj između aplikacije koju razvija korisnik i hardvera za izračunavanje.

Sistemski softver se može nadalje podeliti na operativni sistem, bliži hardveru, i na pomoćne programe, deo bliži aplikaciji. Operativni sistem objedinjuje funkcije ulazno-izlaznih aktivnosti, mehanizme obrade hardverskih i softverskih prekida, mehanizme za planiranje, izvršavanje i praćenje izvršavanje zadataka.

Promena paradigme korišćenja računarskog sistema nije samo podigla produktivnost programera pri razvoju aplikacija, već je smanjila i fiksne troškove njihove obuke. Oni nisu više morali da budu upoznati sa detaljima rada i karakteristikama hardvera, već pre svega sa načinom rešavanja problema uz pomoć višeg programskog jezika. Programerima je bilo omogućeno da u realizaciji koriste ranije napisane i proverene delove programskog koda.

Jednom napisana aplikacija na nekom višem programskom jeziku mogla se prenositi i izvršavati na drugim računarskim siste-

mima istog ili različitih proizvođača. Portabilna aplikacija je time stekla svojstvo gotove robe.

Ako se ima u vidu da je većina gotovih aplikacija nastajala u okviru firmi koje su proizvodile pre svega hardver, odnosno sopstveni sistemski softver, onda je razumljivo da su takve aplikacije projektovane pre svega sa svojstvom tzv. vertikalne kompatibilnosti, tj. mogućnosti instalacije na sistemima istog proizvođača različite obradne snage.

Računari su se projektovali da omoguće simultani rad većeg broja korisnika, koji se dele u dve kategorije, kategoriju profesionalnih korisnika - programera i kategoriju pravih korisnika.

Za ovu fazu evolucije IS-a karakteristično je i poboljšanje performansi harverskog korisničkog interfejsa. Vizuelno alfanumeričko komuniciranje posredstvom ekranskog terminala postaje opšte prihvaćen standard. Ova faza se vezuje za period od 1965. do 1985.godine.

Paradigma korisniku bliskog sistema

Osamdesetih godina prošlog veka, odigrao se veliki skok u razvoju IS, pojavom jeftinog i svima dostupnog personalnog računara (PC). Pojava personalnog računara dovela je do nove paradigme u korišćenju računarskih sistema.

Tipičan korisnik PC-ja je vrlo malo upoznat sa tehničkim karakteristikama i mogućnostima računara. On kupuje računar da bi rešio probleme u svom okruženju i podigao sopstvenu produktivnost. Zapravo korisnik kupuje rešenje svog problema, a rešenje čini mikroračunar sa skupom gotovih aplikacija. Sada se veći deo hardvera i softvera javlja u ulozi "prilagodnog" sloja, interfejsa između korisnika i aplikacije. Kako je vizuelno simboličko komuniciranje zasnovano na korišćenju "ikona", jasna je važnost grafičkog korisničkog interfejsa, i to ne samo u hardverskom delu sa ekranom u boji visoke rezolucije i odgovarajućom grafičkom karticom, već i softverskim grafičkim interfejsom kakav je, na primer, WINDOWS.

PC omogućava multimedijsku prezentaciju, a skup gotovih programa omogućava potpunu automatizaciju kancelarijskog poslovanja. Kao posledica pojave PC-ja može se navesti raspad tradicionalno jedinstvenih proizvođača hardvera, operativnih sistema i velikih programskih paketa. U ovoj fazi razvoja IS na tržištu se jasno razlikuju proizvođači standardnih hardverskih komponenti, proizvođači sistemskog softvera i specijalizovane kuće za određenu kategoriju aplikacija.

Računarska mreža

Iako nije zamišljen kao sredstvo za realizaciju složenih IS, PC kao radna stanica u lokalnoj računarskoj mreži je ubrzo nakon pojave postao moćno sredstvo za realizaciju IS firmi srednje veličine. Razmena podataka sa drugim entitetima u okruženju se obavlja pomoću telekomunikacione računarske mreže, koja može biti sastavljena od računara sa različitim hardverom i operativnim sistemima. Problem konektivnosti, odnosno međusobnog povezivanja heterogenih računara, rešen je sedamdesetih godina razvojem posebnih komunikacionih protokola, od kojih je najpoznatiji TCP/IP. Aplikacija je ponovo podeljena na specijalizovani sistemski deo - ljusku distribuiranog operativnog sistema i pravu aplikaciju. Danas najveća svetska mreža INTERNET koristi TCP/IP protokol. Računarske mreže poput Interneta su omogućile realizaciju globalnih IS, u kojem svaka informacija u sistemu, bez obzira na njen trenutan položaj, postaje potencijalno dostupna svim korisnicima sistema. Usavršena je tehnologija za efikasno skladištenje, organizovanje i međusobno povezivanje različitih tipova informacija u računarskim mrežama, bilo da su tekstualne, numeričke, tabelarne, grafičke, slikovne, zvučne ili video zapisi.Takve pogodnosti pruža World Wide Web (WWW). Korisnik mreže ima mogućnost da kompletnu svetsku mrežu vidi kao jedinstven gigantski disk sa svim raspoloživim multimedijskim informacijama.

Globalni IS

Sadašnje stanje u oblasti komercijalnih tehnologija za međusobno povezivanje računarskih sistema omogućava kreiranje efikasne infrastrukture za trenutnu isporuku uskladištenih informacija na proizvoljno mesto. Stvoreni su preduslovi za prevazilaženje

ograničenja koja nameću heterogeni distribuirani računarski sistemi u odnosu na koncept dinamičke raspodele procesorskih opterećenja. U osnovi je jednostavna ideja: "Ako se već postojeći mehanizam na kojem leži WWW koristi za prenos informacija, zašto se na isti način ne bi preneli i programi koji rade sa tim informacijama." Dakle, isti programi za sve računare u mreži. To je relativno jednostavno u slučaju homogene mreže, ali to je moguće i u heterogenoj mreži, ukoliko se svi programi pišu za unapred dogovorenu apstraktnu mašinu, a potom se za računare različitih proizvođača napiše kratak program - simulator, orijentisan na rad sa WWW serverom. Takve apstraktne mašine su raspoložive od 1995.godine. Ova softverska paradigma je evolutivna, a u osnovi nasleđuje svojstva pethodnih paradigmi.

Tabela 1.2: Pregled paradigmi arhitekture softvera i njihovih glavnih
osobina, [2,3]:

Naziv paradigme	*Glavne osobine*
faza monolitne programske podrške	• proces obrade vezan za uređaje, • programi u mašinskom jeziku ih asembleru, • interfejs nekomforan i malih mogućnosti
dihotomija aplikacija - sistemski softver	• pojava multiprogramskog rada, • mogućnost prenosa programa, • interaktivno komuniciranje i razmena podataka pomoću telekomunikacija
okruženje blisko korisniku	• kupovina gotovih rešenja, • pojava grafičkog korisničkog interfejsa, • aplikacije kao konfekcijski proizvodi
računarske mreže	• povezivanje računara sa različitim hardverom i operativnim sistemima, • korišćenje Interneta, • globalizacija informacijskih sistema
globalni multimedijalni informacijski sistem	• upotreba različitih tipova informacija, • interaktivna isporuka informacija i pratećih programa, • jedinstveno praćenje informacija iz svetskog okruženja

Literatura:

1. Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin, 2001.
2. B.Jošanov,P.Tumbas, *Softverski inženjering*, VPŠ, Novi Sad, 2002.
3. D.Starčević, *Evolucija arhitekture softvera*, INFOTEH 96,Donji Milanovac,str.34-40
4. D.Verne Morland, *The Evolution of Goftware Architecture*, Datamation, February 1.,1985.
5. Brad J. Cox, *Object-Oriented Programming - An Evolutionary Approach*, Addison-Wesley,1987.

2. ALATI ZA RAZVOJ SOFTVERA (Software development tools)

Danas se na tržištu može naći veliki broj alata za razvoj softvera. Među njima se mogu naći i alati specijalizirani za izvršavanje samo jedne funkcije kao i kompletni paketi razvojnih alata namenjenih svim životnim ciklusima softvera, tzv. IDE (*integrated development environment*). Iako je broj alata i priručnika za njihovu upotrebu vrlo velik, mali je broj članaka o softverskim alatima napisan sa nekog opšteg tehničkog aspekta. Dodatni problem su česte i brojne promene pojedinih alata. Iako svaki od alata ima svoja ograničenja, dobar IDE ili neki drugi razvojni alat, upotrebljen na odgovarajući način, značajno će doprineti uspehu projekta.

Alati za razvoj softvera su alati čija je svrha podrška procesima životnog ciklusa softvera. Alati omogućavaju automatizovanje repetitivnih, precizno definisanih aktivnosti, čime se reduciraju napori softver inženjera, te im se time pruža mogućnost da se skoncentrišu na kreativne aspekte projekta. Alati su često dizajnirani kao podrška određenim softver inženjering metodama čime se značajno reduciraju administrativni poslovi. Svrha alata, kao i metoda uostalom, je sistematizirati pristup softver inženjeringu.

SWEBOK[1] deli softver inženjering alate u deset kategorija:

1. Software Requirements Tools,
2. Software Design Tools,
3. Software Construction Tools,
4. Software Testing Tools,
5. Software Maintenance Tools,
6. Software Configuration Management Tools,
7. Software Engineering Management Tools,
8. Software Engineering Process Tools,
9. Software Quality Tools,

[1] *Guide to the Software Engineering Body of Knowledge; standard IEEE Computer Society-a, objavljen u februaru 2004. godine*

10. Miscellaneous Tools.

1. Software Requirements Tools
Ovi alati namenjeni su specificiranju softverskih zahteva, a dele se u dve kategorije: alate za modeliranje i *traceability* alate.

1.1.Alati za modeliranje zahteva. Ovi alati namenjeni su otkrivanju, analiziranju, specificiranju i validaciji softverskih zahteva. Primeri alata za modeliranje zahteva (pri čemu neki od alata namenjeni su samo za modeliranje zahteva, ali ih se većina može koristiti i za druge aktivnosti u svim fazama životnog ciklusa softvera):

- AnalystPro od Goda Software
- Caliber RM™ od Starbase®
- C.A.R.E. od Sophist Group
- IRqA od TCP Sistemas e Ingeniería
- OnYourMark Pro od Omni Vista
- Requirements Design & Traceability (RDT©) od IGATECH Systems Pty Ltd.
- RequireIT™ od Telelogic
- Requisite Pro® od Rational®
- ScenarioPlus for Use Cases
- Slate Require™ od EDS
- Vital-Link od Compliance Automation Inc.
- Volere od The Atlantic Systems Guild
- Cross-Tie Requirements Tracer Software (XTie-RT®) od Teledyne Brown Engineering (TBE)

1.2. Traceability alati. Važnost ovih alata se neprestano povećava s kompleksnošću softvera. Budući da su ovi alati relevantni i za ostale procese životnog ciklusa, često se u klasifikacijama definišu kao zasebna kategorija. Traceability alati omogućavaju inženjerima da povežu zahteve s njihovim izvorima, s promenama zahteva kao i modeliranje elemenata koji će zadovoljiti zahtieve.Ovi alati omogućavaju praćenje zahteva kroz niz dokumenata koji su nastali kao popratna dokumentacija u toku razvoja sistema.

Primeri *traceabilitiy* alata:

- AnalystStudio; proizvođač:Rational Software; Opis: Tool Suite. uključuje RequisitePro, Rose, SoDA and ClearCase
- Caliber-RM; proizvođač: Technology Builders, Inc (TBI); Opis: Requirements traceability tool; Opis: CASE alat namenjen inženjeringu celog životnog ciklusa softvera (requirements management, behavior modeling, system design,verification);
- CORE; proizvođač: Vitech Corporation; Opis: omogućava analizu zahteva, behavorial analiza, definisanje arhitekture i verifikacija, validacija dizajna, business process modeling project.
- DOORSrequirelT; proizvođač: Telelogic; Opis: Requirements trace alat integrisan s Microsoft Wordom.

2. *Software Design Tools*

Alati ove skupine omogućavaju kreiranje i proveru dizajna softvera. Kao posledica velikog broja različitih metoda i notacija u dizajniranju softvera postoji i mnoštvo različitih alata, ali niti jedna opšte prihvaćena podela.

2.1.Jedan od danas najčešće korištenih jezika za modeliranje je UML *(Unified Modeling Language)*. UML je industrijski, standardni jezik za specificiranje, vizualizaciju, konstrukciju i dokumentiranje softverskog sistema. Neophodno je imati model softverskog sistema pre nego se uopšte počne pisati kod. Dobro opisan model olakšava komunikaciju među članovima projektnog tima kao i dekomopoziciju kompleksnog softverskog sistema na manje i jednostavnije zadatke. UML se sastoji od:

- elemenata modela - osnovni koncepti i semantika,
- notacije - vizualizacija elemenata modela,
- smernica - načini upotrebe.

UML je i programski jezik i razvojni alat. Mnogi proizvođači (npr. Rational Software -Rational Rose) nude UML alate. UML uključuje skup razvojnih koncepata visokog nivoa kako što su *collaboration, framework i pattern*. Iako je UML dizajniran za objektno-

orjenitisani pristup, moguće ga je koristiti i za modele koji nisu objektno orjentisani.
UML-om su definisani sledeći dijagrami:

- *use case* - dijagram slučajeva upotrebe,
- *class* - dijagram klasa,
- *statechart* - dijagram stanja,
- *activity* - dijagram aktivnosti,
- *sequence* - sekvnecijalni dijagram,
- *collaboration* - kolaboracijski dijagram,
- *component* - dijagram komponenti.

Pomenuti dijagrami omogućavaju veći broj različitih perspektiva na probleme analize i razvoja softvera.

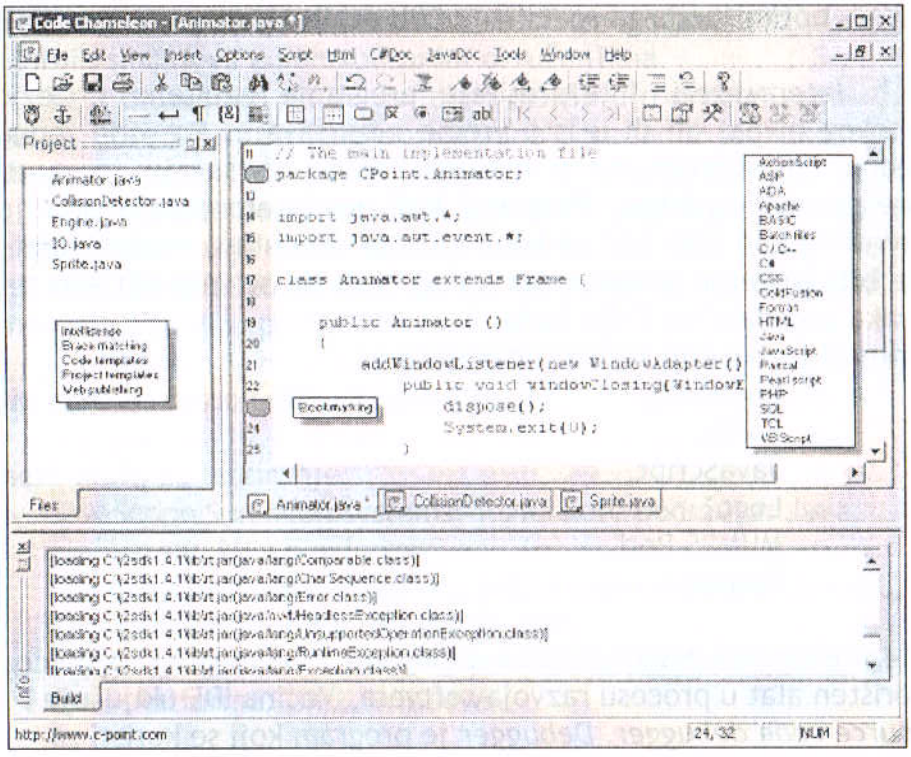
2.2.Alati za testiranje dizajna
Ova grupa alata omogućava softver inženjerima da odluče koje je testove neophodno uraditi, te da generišu testne podatke.

- ALLPAIRS; Test Case Generation Tool; konstruiše mali set testnih slučajeva koji uključuju uparivanje svake vrednosti sa setom parametara;
- AllPairs.java; All-pairs test case generator;
- Assertion Definition Language (ADL); Automatic test generation tool;Assertion;
- DARTT Automated testing over subprogram parameter range; alat za verifikaciju softvera i kvalitativnu analizu koda;
- Datagen2000; Test Data Generation Tool; ovaj alat generirše testne podatke za Oracle baze podataka;
- Datatect Test Data; Generator Test data;
- McCabe Test; Test Design Tool ; vizuelno okruženje za planiranje softverskih testnih resursa;
- Orchid; test case design tool ; omogućava dizajniranje efikasnih testnih slučajeva;
- Panorama C/C++; alat za planiranje i testiranje dizajna.

3. Software Construction Tools
Ovi alati se koriste za kreiranje i prevođenje programske reprezentacije-*source* koda u detaljnu i eksplicitnu formu koju je moguće izvršiti. *Software construction* alate možemo podeliti na:

3.1. Program editori. Ovi alati se koriste za kreiranje i modificiranje programa i dokumenata koji su povezani s tim programima. Danas su editori obično deo IDE.
Primeri program editora:

- CriSP; proizvođač: Vital, Inc ;Opis: file editor za Unix i Windows okruženja;
- EditPad Lite; proizvođač: JGsoft - Just Great Software; Opis: basic text editor, zamena za Notepad;
- VbsEdit;proizvođač: ADERSOFT; Opis: služi za editiranje VBS datoteka;
- Code Chameleon, proizviđač: C Point;
- Shusheng SQL Tool; Proizvođač: CELC ; Opis:web-based SQL programski alat;
- HexCmp; proizvođač: Fairdell Software; Opis: convenient visual binary file comparison application.



Slika 2. 1: Izgled program editora Cameleon

3.2 *Kompajleri i generatori koda.* Tradicionalno shvatanje kompajlera podrazumeva *non-interactive* prevodioce *source* koda, ali danas postoji trend da se kompajleri i tekst editori integrišu u jedinstveno programsko okruženje (IDE). U ovu grupu alata spadaju i pre-procesori, linker/loader i generatori koda. Jedna od ključnih karakteristika kompajlera je brzina. Dobar kompajler omogućava tri nivoa kompajliranja. Osnovni mod ne nudi informacije za ispravljanje grešaka (debug information). Prilikom kompajliranja velikih programa u procesu razvoja, ova opcije može bit vrlo korisna jer je najbrža. No, ukoliko želite dodatne informacije o greškama biće potrebno više vremena za kompajliranje. Najsporije kompajliranje ćete dobiti u slučaju kada isključite opciju za optimizaciju. Nivo optimizacije se razlikuje od kompajlera do kompajlera i u većini slučajeva biste trebali proveriti što vaš kompajler nudi:

- nekoliko dodatnih optimizacija opšte namene,
- optimizaciju za specifični CPU instrukcijski set,
- optimizaciju za specifičnu CPU arhitekturu.

3.3. Interpreteri. Interpretiraju programe napisane u jezicima visokog nivoa, ali ih u isto vreme i izvršavaju. Prevode jednu po jednu liniju programa u mašinski jezik, izvršavaju je, a zatim prelaze na narednu. Programi koji se interpretiraju sporije se izvršavaju od onih koji se kompajliraju. Međutim, ovakakv alat može biti izuzetno koristan jer omogućava da se interaktivno testira svaka pojedinačna linija koda.
Primeri jezika koji koriste interpretere:

- Euphoria,
- Forth,
- JavaScript,
- Logo,
- MUMPS Perl,
- Python.

3.4. Debuggeri. Nakon editora *debuggeri* su verovatno najčešće korišten alat u procesu razvoja softvera. Većina IDE uključuje i *source code debugger. Debugger* je program koji se koristi za ispravljanje grešaka u drugim programima.
Primeri *debuggera*:

- GNU Debugger (GDB) radi pod Unix sistemima i na mnogim programskim jezicima uključujući i C, C++ i FORTRAN;

- CodeView i Visual Studio Debugger; Microsoft debugger za programe pisane u Microsoft C i CodeView;
- T-Bug, the integrated debugger pod Perl 5;
- Broadway;
- DAEDALUS
- Java Platform Debugger Architecture;
- Adb;
- Sdb;
- Dbx;
- Dynamic debugging technique (DDT);
- Purify;
- Ladebug.

4. Software Testing Tools

U ovu kategoriju alata spadaju:

4.1. Test generatori. Ovi alati omogućavaju razvoj testnih slučajeva.

4.2 Test execution frameworks. Ovi alati omogućavaju izvršavanje testnih slučajeva u kontrolisanim okruženjima.

4.3. Test evaluation alati. Ovi alati olakšavaju procenu rezultata testa, određujući da li je posmatrano ponašanje jednako očekivanom.

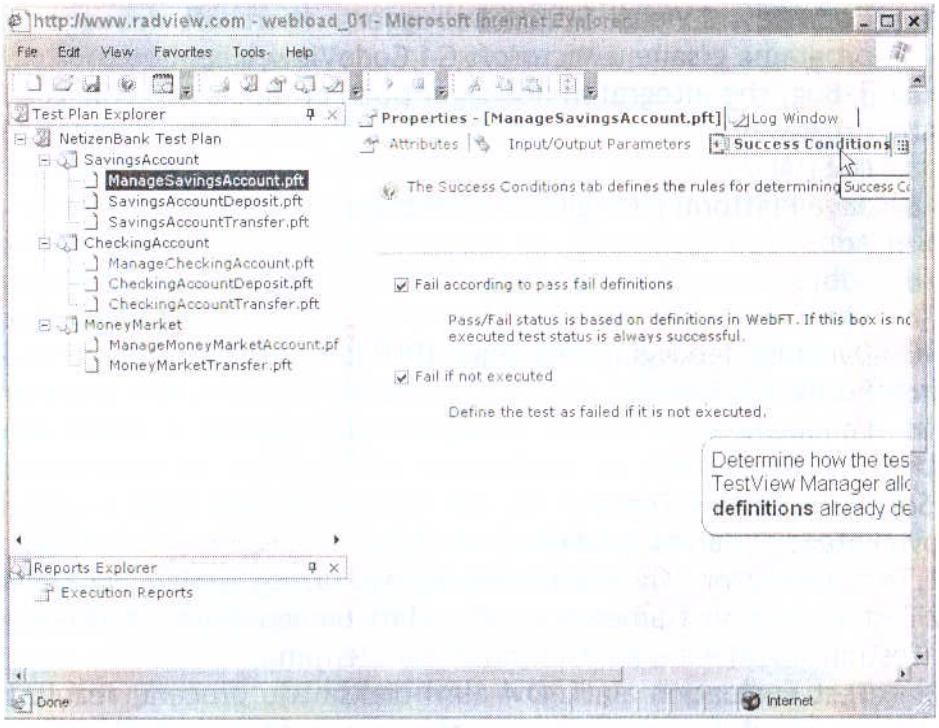
4.4. Test management alati. Ovi alati nude podršku za sve apkete procesa testiranja softvera.

4.5. Alati za analizu performansi. Ova grupa alata se koristi za merenja i analizu performansi softvera.

Testni alati su često i sastavni dio IDE.

Primeri alata za testiranje:

- JavaScope Sun Microsystems; jezici koje podržava: Java; platforma: Java platform;
- Pegasus Ganymede Software; jezici koje podržava: Java, C, C++; platforma: Unix, Windows, Mac;
- WebLoad Radview Software; jezici koje podržava: Java, C, C++; platforma: Unix, Windows.



Slika 2. 2: Izgled alata za testiranje, WebLoad Radview Software

5. Software Maintenance Tools

Ova kategorija obuhvaza alate neophodne za održavanje postojećeg softvera. Razlikujemo dve kateorije:

5.1. *Comprehension* alati. Ovi alata olakšavaju ljudima razumevanje programa. Obično uključuju alate za vizuelizaciju.

5.2. Reinženjering alati. Reinženjering se definiše kao preispitivanje i preoblikovanje softvera u novu formu kao i implementacija tog novog oblika softvera. Reinženjering alati podržavaju takve aktivnosti. Postoje i tzv. *reverse engineering* alati koji podržavaju proces radeći obrnutim redosledom - od postojećeg produkta kreiraju specifikaciju i opis dizajna koji se onda može upotrebiti u transformaciji i generisanju novog produkta iz postojećeg..

Primeri *maintenance* alata:

- BPR Toolkit; Business Process Reengineering Toolkit;
- Ensemble ; Automated maintenance and testing for existing C code;

- 4D ; C/C++ Reverse Engineering and Documentation Tool;
- Vantage Team; Integrated development environment for client-server maintenace and development.

6. *Software Configuration Management Tools*
Ovaj skup alata se može podeliti u tri kategorije:

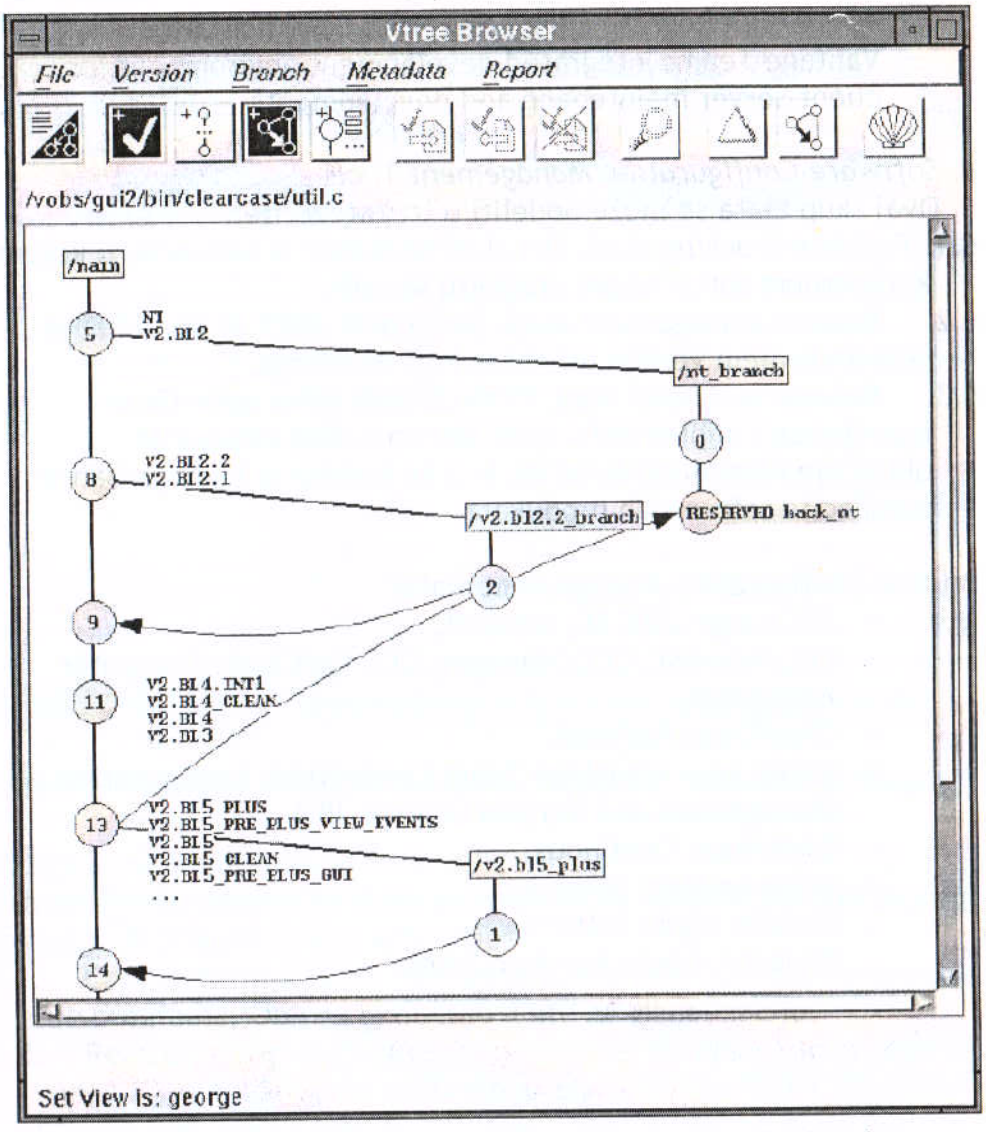
6.1. *Problem-tracking* alati. Ovi alati se koriste u zavisnosti o kojem konkretnom softverskom produktu se radi.

6.2. *Version management* alati. Svrha ovih alata je upravljanje velikim brojem verzija produkta koji se razvija.

6.3. *Release and build* alati. Pomoću ovih alata upravlja se izgradnjom i implementacijom softvera. Ova kategorija uključuje i instalacijske alate koji se koriste za konfigurisanje instalacije softverskih produkata.

Primeri Configuration Management alata:

- AllChange 2000 SE; IntaSoft,
- CCC/Harvest, CCC/Manager, CCC QuikTrak; Computer Associates,
- ClearCase; Rational,
- CMVC, now VisualAge Team Connection; Configuration Management and Version Control, IBM,
- Continuus; Continuus,
- eChange Man; Serena,
- Enabler aqua; Softlab,
- Endevor; Computer Associates.



Slika 2.3: Izgled alata ClearCase kompanije Rational

7. Software Engineering Management Tools
Engineering management alati se dele na tri kategorije:
7.1. Alati za planiranje i prećenje projekta (*Project planning and tracking tools*). Ovi alati se koriste za procenu i merenja troškova i pravljenje rasporeda toka projekta.
7.2. *Risk management* alati. Ovi alati se koriste za identifikaciju, procenu i nadgledanje rizika.

7.3. Alati za merenja. Ovaj skup alata omogućava sprovođenje merenja softverskih produkata.
Primeri *engeneering managemnet* alata:

- DecisionPro; Vanguard Software Corp,
- PRA - Project Risk and Contingency Analysis;Katmar Software,
- Software Risk Evaluation; Software Engineering Institute,
- C-Cover: test coverage, measurement,
- ActionPlan; Netmosphere Inc.,
- AIO WIN; Knowledge Based Systems, Inc.

8. Software Engineering Process Tools
Softver *engineering* proces alati se dele na:
8.1. Alate za modeliranje procesa (Process modeling tools). Ovi alati se koriste za modeliranje i istraživanje softver inžinjering procesa.
8.2. Alate za upravljanje procesima (Process management tools). Zadatak ovih alata je osigurati podršku *management*-u softver inženjeringa.
8.3. Integrisana CASE okruženja. Ovaj skup alata je podrška za više faza životnog ciklusa softvera.
8.4. Process-centered software engineering environments. Ova okruženja uključuju informacije o procesima životnog ciklusa softvera, te vode i omogućavaju nadzor nad definisanim procesima.

Primeri *Software Engineering Process* alata:

- Computer Associates BPWin,
- First Place Group Proprietary Tools,
- Microsoft Visio.

9. Software Quality Tools
Quality alati se mogu podeliti u dve kategorije:
9.1. *Review and audit tools*. Glavni zadatak ovih alata je nadzor, (*These tools are used to support reviews and audits*).
9.2. *Static analysis tools*. Ovi alati se koriste za analizu softvera, a uključuju sintaksne i semantičke analizatore, analizatore zavisnosti kao i kontrolu toka podataka.

Primeri *quality* alata:

- AceProject, web-bazirani projekt *management* softver i *bug tracking* alat,
- AdventNet QEngine; *comprehensive cross-platform tool with a compelling business value proposition for automating large-scale java and J2EE application testing,*
- Aimware; pomoć pri test *management*-u.

10. Miscellaneous tools (Ostali alati)
Ovu kategoriju sačinjavaju alati koji na neki način podržavaju razvoj softvera, ali se s obzirom na navedenu podelu ne uklapaju niti u jednu kategoriju. Kategoriju alata ostali možemo podeliti na:

10.1. Integrisani alati (IDE - interactive development environment)
Integracija alata u jedinstveno okruženje je izuzetno važna zbog omogućavanja individualnim alatima da rade zajedno. Ova kategorija alata se preklapa sa integrisanim CASE okruženjima, međutim ova vrsta okruženja danas sve više dobija na značaju, te je stoga potrebno posebno da se naglasi. Kvalitetan IDE sadrži širok raspon alata kao što su editori, debugger i alati za testiranje koji podržavaju razvoj softvera. IDE omogućava integraciju sastavnih komponenti tako da olakšava programeru rad u *edit-compile-debug* ciklusu.

Primeri IDE:

- Bean Machine IBM Java; Windows, OS2, Unix; Builder Xcessory Pro Integrated Computer Solutions Java, C, C++ Unix, Windows
- CodeWarrior Professional Metrowerks; Java, C, C++, Pascal;Unix, Windows, Mac
- Java Workshop Sun Microsystems Java; Solaris, Windows
- JBuilder Imprise; Java; Windows, AS400
- SuperCede for Java Supercede;Java; Windows
- UIM/X VisualEdge Software; Java, C, C++; Unix, Windows, Mac
- Visual Cafe for JavaSymantec; Java; Windows
- VisualAge IBM; Java; Unix, Windows
- Visual J++ Microsoft; Java; Windows

10.2. Meta alati. Meta-alati generišu druge alate;
10.3. Evaluacijski alati.

Još neki primeri alata kategorije *ostali*:

- Aivosto VB Watch; VB coverage, performance, debug and error detection tool.
- AQtime; Integrated profiling and leak-detection tool
- ASSIST (Asynchronous/Synchronous Software Inspection Support Tool); Enforcement and support of the inspection process
- Camtasia Video Screen Recorder Software
- Phoenix ImageCast MFG Disk imaging tool
- IPCheck Server Monitor Uptime/Downtime Monitoring Tool
- Norton Ghost Disk imaging tool
- NULLSTONE Automated Compiler Performance Analysis Tool Automated Compiler Performance Analysis Tool
- Q-CHESS Quality managament, web-based checklist support system
- Rational Quantify Performance Profiling Tool
- ReviewPro ReviewPro is a proven web-based, collaborative Technical Review and Inspections solution.
- SSW SQL Total Compare Database Tool
- Virtual PC Virtualization software
- VMware Virtualization software

Literatura:

1. Marc Hamilton; *Software Development: Building Reliable Systems*, Publisher : Prentice Hall PTR; Pub Date : March 22, 1999; ISBN : 0-13-081246-3
2. http://www.rational.com/products/rose/features.html.
3. http://www.ganymedesoftware.com.
4. http://www.radview.com.
5. http://sun.com.
6. http://www.computer.org/certification/Swebok_2004.pdf
7. http://www.site.uottawa.ca:4321/oose/index.html#softwarecrisis
8. http://epic.onion.it/workshops/w09/slides01/
9. http://www.itmweb.com/essay544.htm#f5

3. DIZAJN SOFTVERSKOG PROIZVODA (Software products design)

Polazeći od izvršene analize sistema, faza projektovanja, tj. dizajna treba da predloži kako realizovati softverski proizvod. Osnovni cilj ove faze je kreiranje jednostavnih, jasnih i preciznih specifikacija za efektivnu i efikasnu izradu i implementaciju softverskog proizvoda.

Procedura projektovanja može da teče na sledeći način:

1. Na osnovu identifikovanih i specifikovanih informacionih zahteva formulišu se neophodni ulazi koje treba obezbediti da bi softverski proizvod mogao formirati tražene izlaze.
2. Vrši se dekompozicija funkcija i pri tome se dekomponuju i predviđeni ulazi i izlazi.
3. Formira se predlog organizacije podataka.
4. Definišu se funkcije sa kojima će se obezbediti unos ulaznih podataka i realizovati informacioni zahtevi, u skladu sa predviđenom organizacijom podataka.
5. Daju se detaljni opisi funkcija iz br.4, sa dokumentovanim ulazima, izlazima, rečnikom i organizacijom podataka, koji će poslužiti kao baza za specificiranje programskih zahteva, na osnovu kojih se sprovodi kodiranje.

Uloga projektovanja je da navede alternative rešenja datog problema. Ovaj proces predstavlja alternative rešenja, modelira ih i razvija u skladu sa specificiranim zahtevima.

Proces projektovanja ima za cilj da definiše kako izgraditi sistem da bi se ponašao na način opisan zahtevima. U ovom procesu se izrađuje niz dokumenata koji treba da obezbede ulaze u proces implementacije. Kao forma procesa rešavanja problema, ovaj proces uključuje aspekt fizičkog i logičkog projektovanja.

Dva osnovna pristupa u današnjoj praksi su:

1. strukturni dizajn i
2. objektno-orijentisani (OO) dizajn.

Osnovne karakteristike strukturnog dizajna su:

- modeluje rešenja problema, a ne sam problem,
- rešenje se hijerarhijski razlaža na jednostavnije funkcionalne celine,
- problemi se rešavaju u određenim algoritamskim koracima, na višem ili nižem nivou hijerarhije,
- kada je potrebno izvršiti promene u programu, najčešće je potrebno menjati i algoritme,
- nakon dodavanja i izmena u programu, potrebno je ponovo proveriti širi kontekst projektovanog rešenja.

OO dizajn karakteriše da se:

- ovim pristupom modeluju problemi, a ne rešenja,
- problemi se razlažu na objekte, za koje se određuje šta, a ne kako rade,
- objekti se u dizajnu sistema tretiraju kao crne kutije,
- nad objektima se izvršavaju spoljne akcije,
- izmene i dodavanja vrše se najčešće u određenom objektu,
- dodavanje novih objekata je fleksibilno,
- postoji mogućnost ponovnog korišćenja komponenti ili njihovih delova.

U dijzajnu softverskih proizvoda se koristimo modeliranjem i nastojimo iskoristiti važnost modeliranja za ciljeve koji želimo postići.

3.1. Modeliranje

Modeliranje je centralni deo svih aktivnosti koje vode do generisanja dobrog software-a. Jedna firma za proizvodnju softvera je uspešna u meri koliko proizvodi na konzistentan način kvalitetan softver koji zadovoljava potrebe korisnika.

Za proizvodnju kvalitetnog softvera je potrebno:

-povezati i uključiti sve korisnike u cilju prikupljanja svih realnih zahteva sistema,

- kreirati čvrstu arhitektonsku bazu koja će takođe biti fleksibilna na promene,

-koristiti čvrste razvojne procese koji su prilagodivi mogućim promenama problema koji rešavamo.

Ukoliko proizvodimo softver malih dimenzija, pogrešimo, a rezultati ovog softvera nemaju posledice za okruženje u kojem će delovati, ni po podatke kojima će manipulisati, ne moramo ulagati puno truda u planiranje i modeliranje.

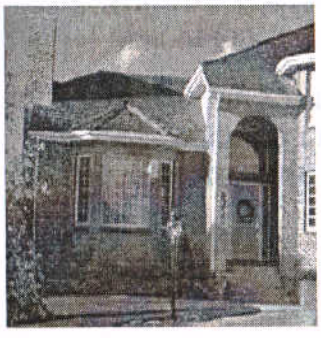
Ukoliko proizvodimo softver velikih dimenzija i čija efikasnost i rezultati u velikoj meri utiču na okruženje i najmanja greška može imati katastrofalne posledice. Zbog toga je potrebno veće projekte pažljivo planirati i uložiti trud u modeliranje.

Dobar primer je poređenje gradnje kućice za psa, kuće za jednu obitelj i poslovne zgrade.

Kuća za psa:

- Nije nam potrebno puno alata.
- Nije potrebno puno planirati.
- Ukoliko mu se ne svidi, nije problematično!

Kuća za jednu porodicu:



- Potrebno je planirati.
- Verovatno je potrebna ekipa (tim).
- Različiti planovi (spratovi, vodovod, ...).
- Poodica je zahtevna.

Poslovna zgrada:

- Biće potrebno planirati više mjeseci.
- Cena neuspeha je jako visoka!

Ali često se dogodi da:

- mnogi projekti imaju ambicije konstruisati neboder, ali prilaz problemu im je kao da konstruišu kućicu za psa,
- neki opet krenu sa ciljem da konstruišu kućicu za psa koja vremenom dostigne veličinu nebodera,
- dolazi se do momenta kad se kućica sruši na psa!

Model

Model je pojednostavljena realnost. Pruža nam planove sistema kao što su: opšti planovi (globalna vizija) i detaljni planovi pojedinih delova. Kreiramo modele da bismo bolje razumeli sistem koji želimo konstruisati. Modeli nam pomažu da vizueliziramo kakav je sistem ili kakav želimo da bude, daju nam šablon koji nas vodi u konstrukciji jednog sistema i dokumentuju odluke koje smo doneli. Modelom takođe specificiramo strukturu i ponašanje sistema.

Postoji ljudski limit da razume kompleksnost. Putem modeliranja reduciramo problem koji se izučava i centriramo se u svakom momentu samo na jedan aspekt. Modeliranjem se potencira ljudski um: model koji je adekvatno izabran dozvoljava da radimo na većem nivou apstrakcije.

Principi modeliranja

Upotreba modeliranja ima dugu istoriju u svim inženjerskim disciplinama. Ova istorija sugeriše četri osnovna principa:

1. Izbor koji model kreirati ima veliki uticaj u pristupu problemu i kako se daje forma rešenju.

2. Svi modeli mogu biti izraženi sa različitim nivom preciznosti (detaljnosti).
3. Najbolji modeli su usko vezani za realnost. Svi modeli su pojednostavljena realnost. Umetnost je u tome da pojednostavljenja koja su napravljena ne prikrivaju ni jedan važan detalj iz realnosti.
4. Jedan model nije dovoljan. Bilo kojem netrivijalnom sistemu se prilazi na bolji način putem manjih skupova modela, skoro nezavisnih.

Zašto orijentisanost ka objektima u modeliranju?

Orjentisanost ka objektima je u širokoj primeni zbog sličnosti koncepata modeliranja u odnosu na realne entitete, boljeg prikupljanja i validiranja zahteva i približavanja problema njegovom rešenju.

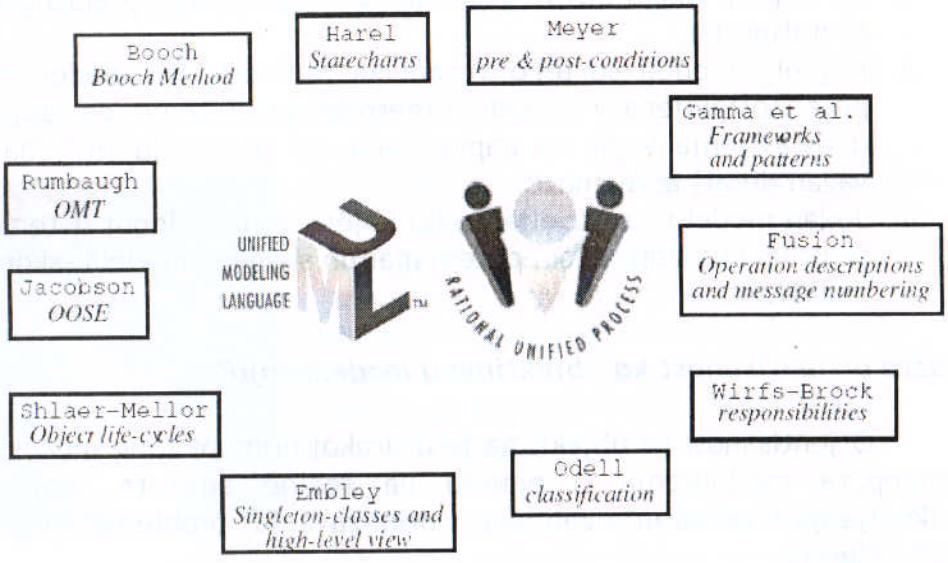
Zajednički koncepti modeliranja tokom analize, dizajna i implementacije: olakšavaju prelazak na sledeću fazu, poboljšavaju interaktivni razvoj i postavljaju granicu između «šta» i «kako».

3.2. UML - Unifed Modelig Language

Jedna od najpoznatijih OO metodologija jeste UML.

Prvi OO programski jezici su se pojavili 60-tih godina:
- SIMULA (1965. godine i zatim 1967. godine),
- SMALLTALK (70-tih i u potpunosti 80-tih XX veka).

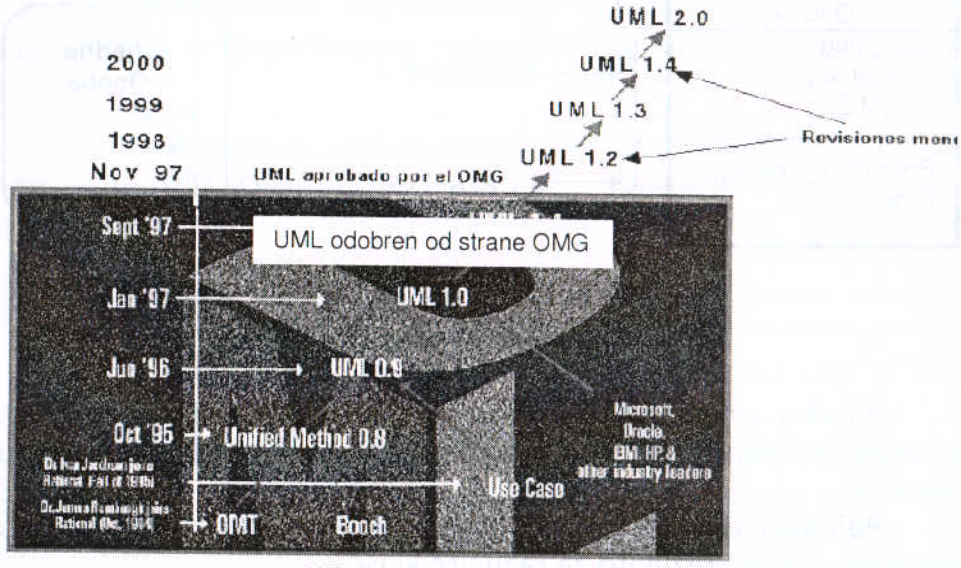
Tradicionalne metode analize i dizajna (strukturalna metodologija) nisu najbolje prilagođene ovim programskim jezicima



Slika 3.1.: *A ring to bind them all . . .*

Unified Modelling Language ili *UML*, je jezik za grafičko modeliranje, konstrukciju i dokumentaciju elemenata koji formiraju softverski sistem objektno orjentisan. Postao je standard u softverskoj industriji.

Notacija UML-a je opšte prihvaćena zahvaljujući prestižnosti njenih kreatora (Grady Booch, Ivar Jacobson i Jim Rumbaugh) i zbog toga što sadrži prednosti svih metoda na kojima se bazira (u osnovi Booch, OMT i OOSE). UML je stavio tačku na takozvani «rat metoda» koji smo imali tokom devedesetih godina. Sa UML-om se ujedinuju notacije navedenih metoda i napravljen je jedinstveni alat koji dele svi inženjeri softvera koji rade na objektno orjentisanom razvoju softvera.



Slika 3.2.: Istorija UML-a

3.3. Procesi objektno orjentianog razvoja informacijskih sistema

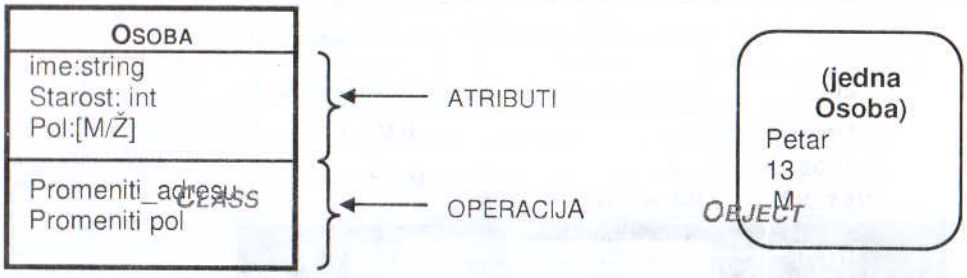
3.3.1. Osnovni koncepti objektno orjentisane paradigme

Objekti su osnovni elementi objektno orjentisane (OO) paradigme. Prezentiraju entitete iz realnog sveta (klijent, bankovni račun, student, itd.). Poseduju atribute (ime klijenta, adresa, i td.), operacije (ponašanje) i sopstveni identitet nezavisan od vrednosti atributa. Objekti su vlasnici svojih atributa i može im se prići samo putem operacija.

Klase su apstrakcije koje opisuju važna svojstva jedne aplikacije. Objekti se grupišu u klase. Klasa je najvažniji koncept OO. Apstraktna su dva tipa svojstava:

- *atributi*: podaci koji karakterišu objekte klase,
- *operacije(metode)*: ponašanje objekata klase. To su akcije ili transformacije koje objekti realizuju ili trpe.

Odabir klasa zavisi od potreba sistema.



Slika 3.3.: Klasa i objekat

Poruka (massage) je način na koji komuniciraju objekti. Porukom možemo reći jednom objektu da realizuje određenu operaciju.

Polimorfizam je koncept koji definiše slučaj kada se jedna operacija ponaša različito za različite klase.

Nasleđivanje je relacija specijalizacije između različitih klasa. Potklase specijaliziraju podatke i ponašanja superklasa.

Prednosti OO metoda:

1. Smanjuje kompleksnost.
2. Svi objekti su definisani, implementirani i testirani tako da se mogu ponovo upotrebiti u drugim sistemima.
3. Sistemi razvijeni ovom metodom su fleksibilniji, tako da se mogu modificirati ili dodati novi tipovi objekata.
4. Analitičari rade na nivou realnog sveta, kao i korisnici.
5. OO metode su idealne za razvoj Web aplikacija.
6. OO metode opisuju različite elemente IS-a u korisničkim terminima, tako da korisnik može lakše da shvati šta će raditi novi sistem i kako će postizati ciljeve.

3.3.2. Aktivnosti softverskog inženjeringa u procesu objektno orijentisanog razvoja informacionih sistema

Svaki složeni softver treba razvijati *iterativno i inkrementalno*. To znači da se postepeno, u svakom koraku, sprovodi celi ciklus: zahtevi, analiza, dizajn, implementacija za manji skup slučajeva upotrebe. U svakom sledećem koraku (iteraciji), dodaju se realizacije novih slučajeva upotrebe, tako da sistem postepeno dobija na funkcionalnosti i složenosti. Svaki sistem koji dobro funkcioniše po pravilu je nastao iz manjeg sistema koji je dobro funkcionisao.

Imajući u vidu postavke vezane za IDEF0[2], UML[3], IDEF1X[4], kao i potrebe za reinženjeringom poslovnih procesa, može se reći da se objektno orjentisani razvoj informacionih sistema izvodi kroz četiri procesa:

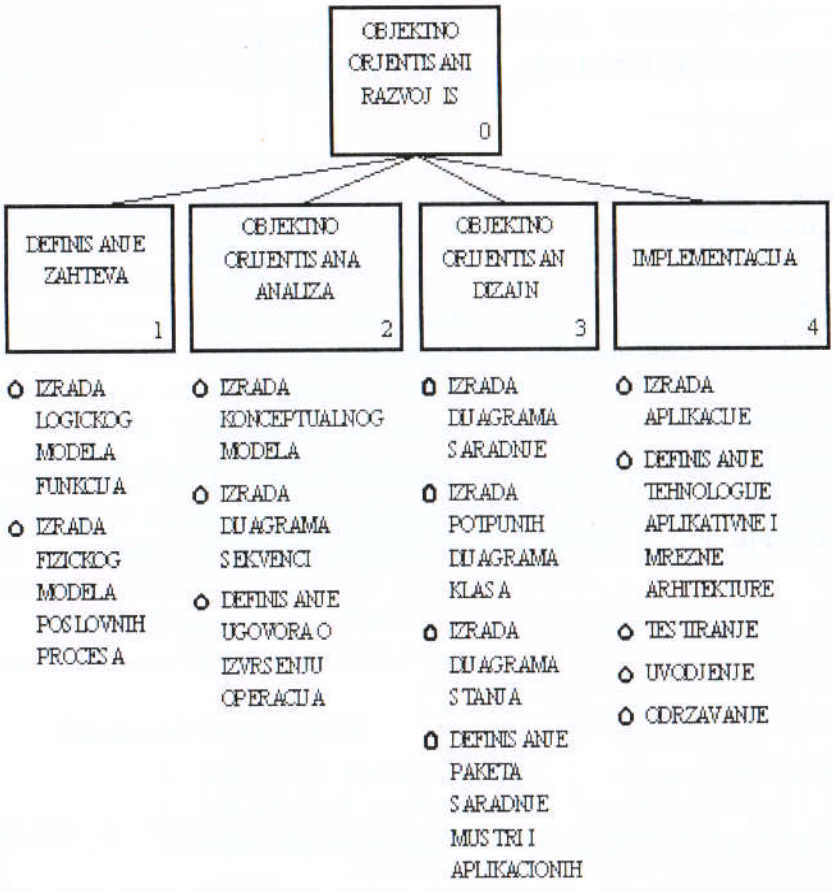
- Definisanje zahteva,
- Objektno orjentisana analiza,
- Objektno orjentisani dizajn,
- Implementacija.

Na slici 3.4. prikazano je stablo aktivnosti vezano za objektno orijentisani razvoj informacionih sistema.

[2] *IDEF0 je metoda dizajna za modeliranje odluka, akcija i aktivnosti jedne organizacije ili sistema.*

[3] *UML(Unified Modeling Language) je case alat za specifikaciju, konstrukciju i dokumentaciju sistema OO metodom*

[4] *IDEF1X je metoda dizajna relacionih baza podataka koja posjeduje sintaksu koja podržaje neophodnu semantiku za razvoj konceptualnih šema.*



Slika 3.4.: Stablo aktivnosti postupka objektno orijentisanog razvoja informacionog sistema

3.3.2.1. Definisanje zahteva

Prvi proces "Definisanje zahteva" se definiše kao:

- Izrada logičkog modela funkcija i
- Izrada fizičkog modela poslovnih procesa.

U okviru aktivnosti "**Izrada logičkog modela funkcija**" treba izvršiti funkcionalnu specifikaciju informacionog sistema. Logički model funkcija definisan je nezavisno od organizacionog ili tehnološkog okruženja u kome će posao biti implementiran. Logički

model funkcija je stabilniji, sporije se menja i može se potpuno ili delimično ponoviti i u drugim organizacijama.

Ovom aktivnošću identifikuju se granice posmatranog sistema, vertikalno povezivanje funkcija kroz definisanje stabla logičkih funkcija, horizontalno povezivanje kroz izradu dekompozicionog dijagrama i definisanje logike primitivnih funkcija.

Aktivnost "Izrada fizičkog modela poslovnih procesa" detaljno opisuje poslovne procese (koristeći UML dijagram aktivnosti) kao sekvence aktivnosti koje se obavljaju u konkretnom organizacionom i tehnološkom okruženju (organizaciona struktura, sistematizacija radnih mesta, tehnologija obavljanja posla). U ovoj aktivnosti se razmatraju poslovi (poslovni procesi) koji se moraju uraditi po određenom redu izvršavanja. Fizički model poslovnih procesa podložan je čestim izmenama zbog promene organizacije i tehnologije obavljanja posla. Ovom aktivnošću definiše se dinamika, odnosno način odvijanja posla gde se poslovni procesi dekomponuju do nivoa primitivnih procesa.

Poslovni procesi se opisuju slučajevima upotrebe. Slučajevi upotrebe opisuju funkcionalnost sistema iz korisničke perspektive i polazni su korak za prikaz upotrebe sistema od strane budućih korisnika u raznim karakterističnim situacijama. Opisivanje dinamike slučajeva upotrebe kao i logike jednog slučaja upotrebe izvodi se *sistemskim dijagramom sekvenci* (sekvencijalnim dijagramom) čime se definiše redosled poruka ili dijagramom aktivnosti za opis onih slučajeva upotrebe gde se opisuje paralelizam u procesima. Implementacija slučajeva upotrebe izvodi se preko saradnje (kolaboracije).

Definisanjem zahteva izvršena je identifikacija sistema nalaženjem funkcionalnog modela sistema. U sledećem koraku potrebno je izvršiti realizaciju (objektno orijentisanu analizu i objektno orjentisani dizajn) sistema nalaženjem modela sistema u izabranom prostoru stanja.

UML-ov dijagram stanja

Automat stanja (*state machine*) je ponašanje koje specificira sekvence stanja kroz koje prolazi neki objekat i modelira istoriju života nekog objekta.

Objekat može biti instanca klase, slučaja korišćenja ili čak sistem u celini. Objekat reaguje na događaje promenom stanja koje takođe izaziva nove događaje.

Dijagrami stanja prikazuju automate stanja fokusirajući se na događajima vođeno ponašanje. Dijagrami aktivnosti takođe prikazuju automate stanja, ali se fokusiraju na tok aktivnosti. Dijagrami stanja se kreiraju za apstrakcije čiji objekti pokazuju bitno dinamičko ponašanje.

Automat stanja se primenjuje da specificira ponašanje:
- objekata koji moraju odgovarati na asinhrone događaje,
- objekata čije tekuće ponašanje zavisi od istorije,

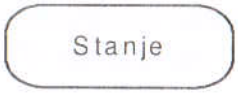
Uspešno se koristi za modeliranje ponašanja reaktivnih sistema (onaj koji odgovara na signale koje daju akteri iz spoljašnjeg sveta).

Elementi dijagrama stanja su:
- stanja,
- događaji koji uzrokuju promenu (tranziciju) stanja,
- akcije koje su rezultat promene stanja.

Stanje objekta je uslov ili situacija u kojoj taj objekat može da postoji. U jednom stanju objekat zadovoljava neki uslov, obavlja neku aktivnost ili čeka događaj.

Primeri:
- uslov - student je u stanju GLADAN ili SIT
- aktivnost - student je u stanju VEČERA
- čekanje - student je počeo sa jelom, i čeka neki prekidni signal koji će ga obavestiti da je SIT

Grafička notacija stanja je pravougaonik sa zaobljenim uglovima:



Početno i završno stanje su dva specijalna stanja:

početno  završno: 

Početno i završno stanje su pseudostanja. Tranzicija iz početnog u završno stanje može imati sve elemente osim okidajućeg događaja.

Elementi stanja su:

- Ime - tekst koji razlikuje jedno od drugih stanja; stanje može biti i anonimno (bez imena)

-Ulazna akcija - atomska radnja koja se obavi pri ulasku u stanje

- Izlazna akcija - atomska radnja koja se obavi pri izlasku iz stanja

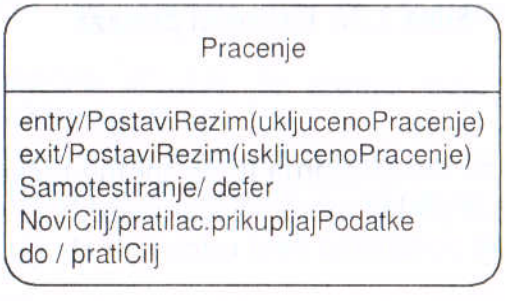
- Aktivnost - neatomska radnja koja se izvršava dok je objekat u datom stanju

- Podstanja - stanja koja postoje unutar datog stanja, sekvencijalno ili konkurentno aktivna

- Odloženi događaji - lista događaja koji se ne obrađuju u datom stanju nego se smeštaju u red

- Unutrašnje tranzicije - tranzicije koje obrađuju događaj i zadržavaju objekat u istom stanju; različite su od samotranzicije: ne izazivaju izlaznu pa ulaznu akciju

Grafički prikaz:



Slika 3.5.: Unutrašnje tranzicije stanja

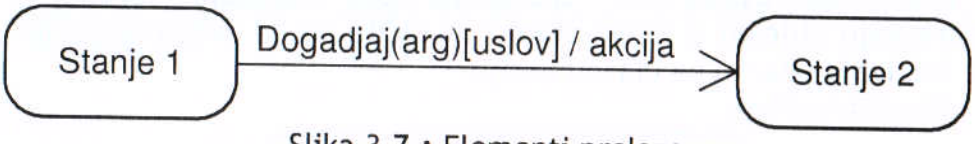
Prelaz (tranzicija) je relacija između dva stanja. Ukazuje da objekat napušta jedno stanje, obavlja akciju i ulazi u drugo stanje kada se dogodi specificirani događaj i kada je ispunjen specificirani uslov.

Grafička notacija - strelica:

Slika 3.6.: Prelaz iz stanja u stanje

Elementi prelaza:

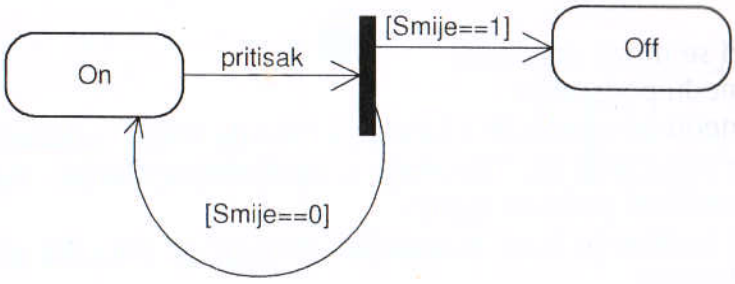
- Događaj je zbivanje koje nema trajanje i može prouzrokovati prelaz
- Zaštitni uslov je Boole-ov izraz koji čini prelaz mogućim kada je uslov ispunjen
- Akcija je atomska radnja koja je pridružena prelazu i može biti:
  - poziv operacije objekta vlasnika automata stanja ili drugog objekta koji je vidljiv datom objektu
  - kreiranje ili uništavanje drugog objekta
  - slanje signala nekom objektu (ključna riječ *send*)



Slika 3.7.: Elementi prelaza

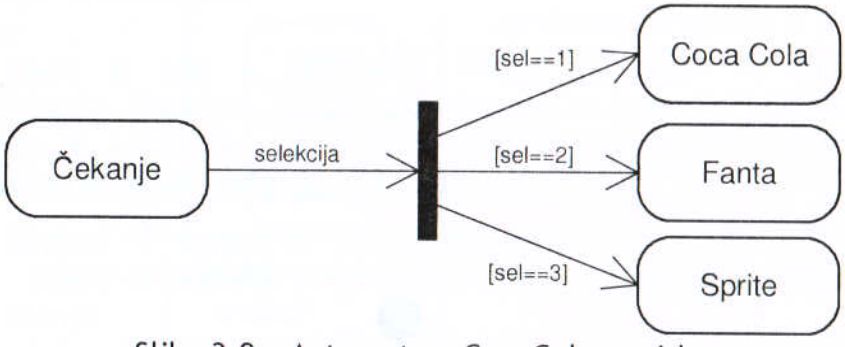
Grananje prelaza

Svaki prelaz može da ide do stanja ili do spojne tačke. Ako ide do spojne tačke onda se prelaz dijeli na segmente, i za svaki segment posebno možemo da postavimo neki uslov i/ili akciju.



Slika 3.8.: Grananje prelaza

Primer dijagrama stanja:



Slika 3.9.: Automat za CocaCola napitke

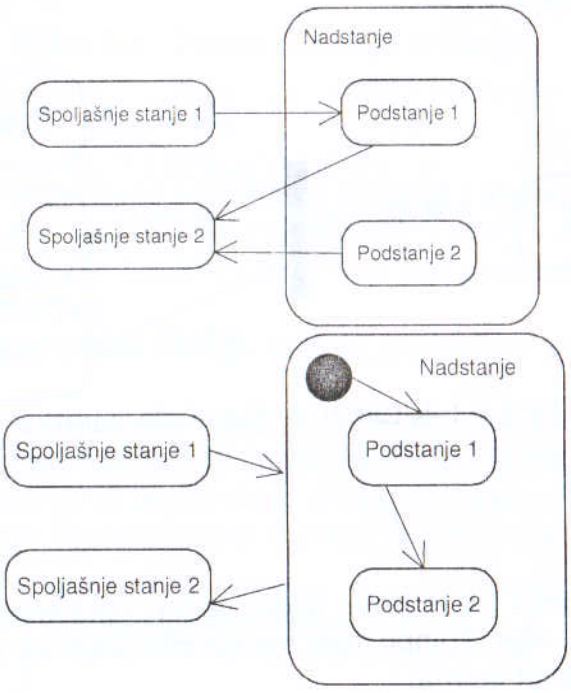
Vrste stanja

Jednostavno stanje je stanje koje nema unutrašnju strukturu automata stanja, dok kompozitno stanje ima unutrašnja stanja, tj. predstavlja automat stanja.
Ugnježđena stanja se koriste da bi se smanjila grafička kompleksnost.
Nadstanje (kompozitno stanje) je stanje koje obuhvata više unutrašnjih (ugniježđenih) stanja.
Podstanje je unutrašnje (ugnježđeno) stanje.
Kada se objekat nalazi u podstanju - istovremeno se nalazi i u nadstanju.
Postanja mogu biti:

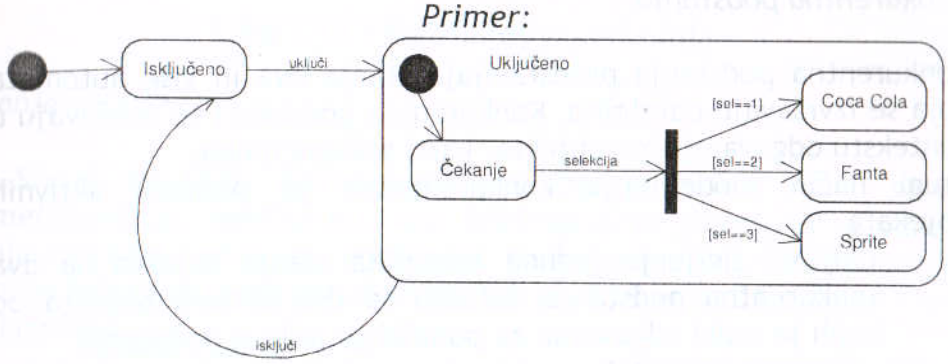
- sekvencijalna,
- konkurentna (paralelna).

Sekvencijalna podstanja

- Prelazi se mogu događati:
  - između podstanja
  - između podstanja ili nadstanja i stanja izvan nadstanja
- Ako je nadstanje cilj tranzicije iz spoljašnjeg stanja - nadstanje mora sadržati početno stanje
- Ako je nadstanje izvor tranzicije - najprje se napušta podstanje pa nadstanje
- Pri tranziciji u/iz nadstanja izvršavaju se ulazne/izlazne akcije i nadstanja i podstanja



Slika 3.10.: Podstanja

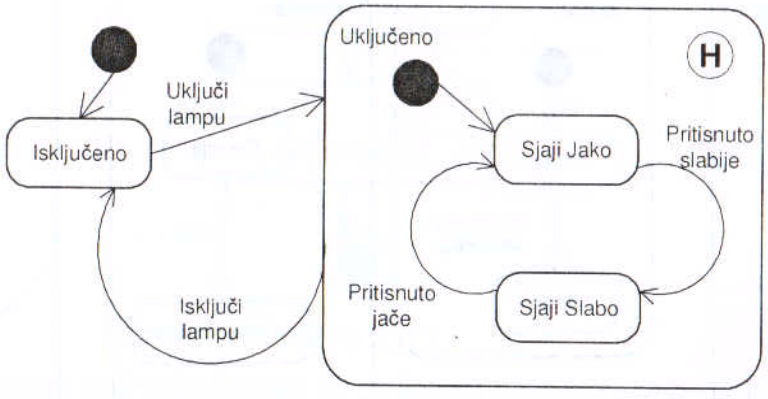


Slika 3.11.: Prošireni automat za CocaCola Napitke

Stanje sa historijom

Kada se uđe u nadstanje obično se kreće od inicijalnog podstanja. Ponekad postoji potreba da se krene od podstanja iz kojeg je nadstanje napušteno. Simbol H u kružiću ukazuje da nadstanje pamti historiju.

- Simbol (H) označava "plitku" historiju:
  - pamti se historija samo neposredno ugnježđenog automata stanja
- Simbol (H*) označava "duboku" historiju
  - pamti se historija do najugnježđenijeg automata stanja proizvoljne dubine



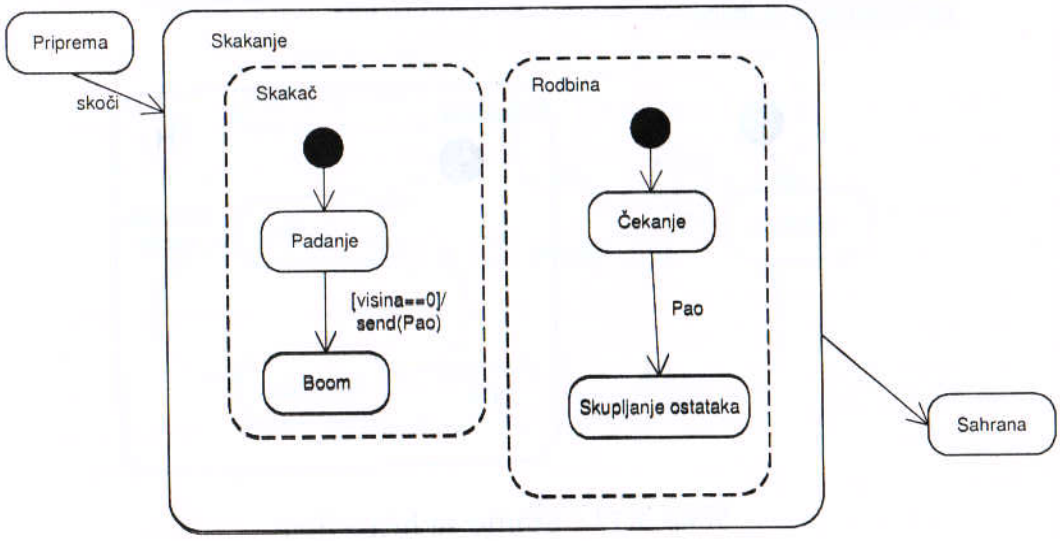
Slika 3.12.: Stanje sa historijom

Konkurentna podstanja

Konkurentna podstanja predstavljaju stanja dva ili više automata koja se izvršavaju paralelno. Konkurentna podstanja se izvršavaju u kontekstu odgovarajućeg objekta, kao i sekvencijalna.
Drugi način modeliranja konkurentnosti je pomoću aktivnih objekata

- umjesto deljenja jednog automata stanja objekta na dva konkurentna podstanja definišu se dva aktivna objekta od kojih je svaki odgovoran za ponašanje jednog podstanja

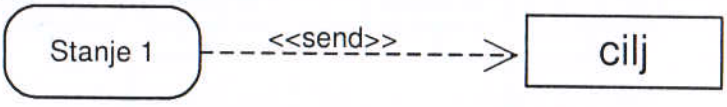
- Od više sekvencijalnih podstanja na jednom nivou - objekat može biti samo u jednom
- Od više konkurentnih podstanja na jednom nivou - objekat je u svakom od njih
- Prelaz u stanje sa konkurentnim podstanjima predstavlja *fork* grananje
- Ako jedno konkurentno podstanje stigne do završnog stanja pre drugog - čeka na drugo
- Sva konkurentna podstanja moraju biti završena da bi se izvršio prelaz *join* iz nadstanja
- Ugnežđeni konkurentni automati stanja ne mogu imati početno, završno i stanje historije
- **Sekvencijalna podstanja koja obrazuju svako od konkurentnih podstanja imaju ova stanja**



Slika 3.13.: Konkurentna podstanja

Slanje signala

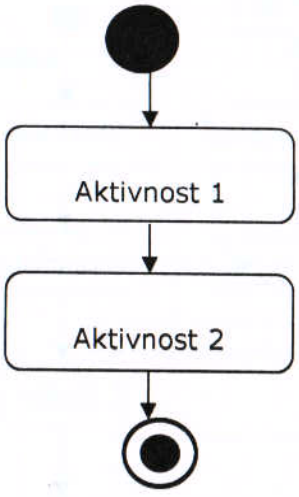
Signal se šalje komandom *send* čiji je opšti oblik: *send (ImeDogadjaja, imeStanja)*. Ako ImeDogadjaja jednoznačno određuje događaj, onda se imeStanja može zanemariti. Kada se iz akcije šalje signal nekom objektu, taj se objekat može prikazati na dijagramu.



Slika 3.14.: Slanje signala

UML-ov dijagram aktivnosti

Dijagrami aktivnosti služe za pojednostavljeni prikaz događanja tokom operacije ili procesa. Svaka aktivnost je prikazana sa elipsom (zaobljenim pravougaonikom). Strelica prikazuje prelaz s jedne aktivnosti na drugu. Početak je prikazan punim krugom, a kraj metom, Slika 3.15:

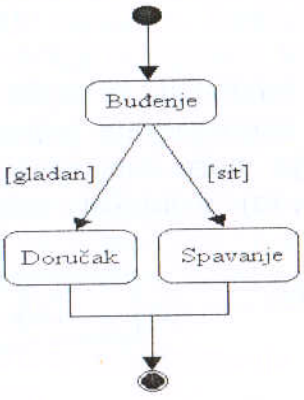


Slika 3.15

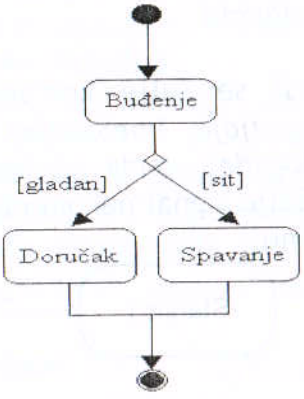
Tačke grananja aktivnosti mogu se prikazati na dva načina, Slika 3.16. a i b:

- staze izlaze (dolaze) direktno iz aktivnosti

-prelaz aktivnosti iz malog dijamanta i onda moguće staze iz tog simbola.

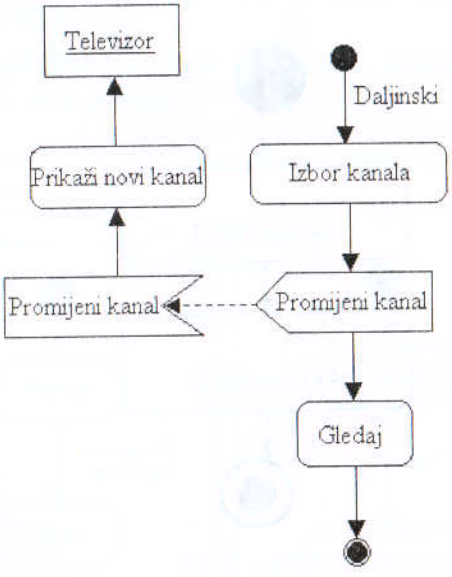


Slika 3.16.a



Slika 3.16.b

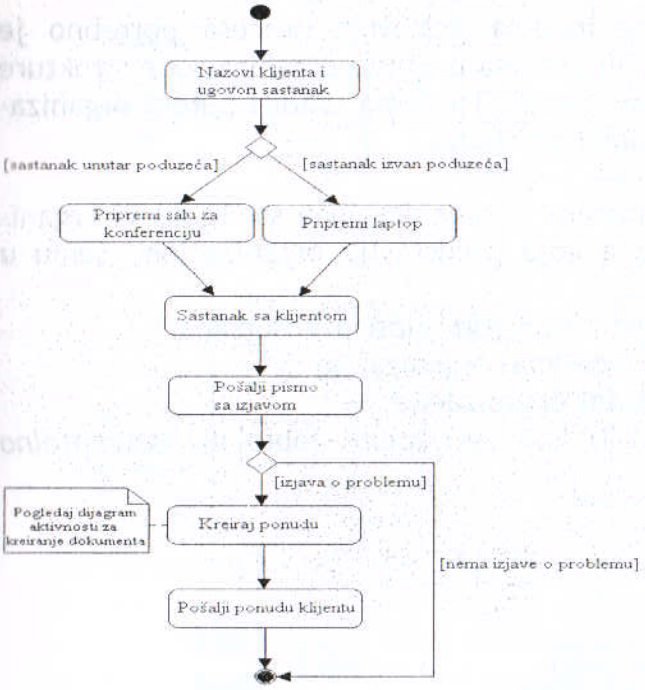
Tokom aktivnosti može se poslati ***signal***. Signal koji se šalje prikazan je konveksnim petouglom, a signal koji se prima, konkavnim petouglom, Slika 3.17.

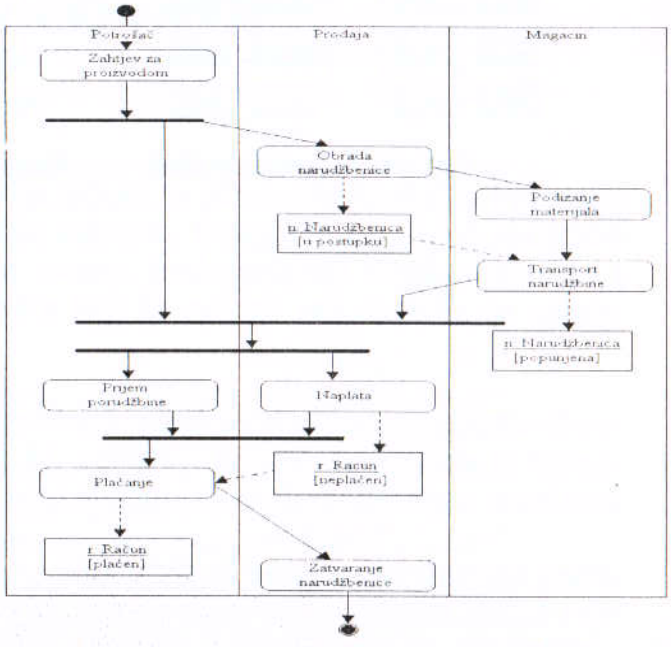


Slika 3.17.

Dijagram aktivnosti vizuelno prikazuje paralelne segmente pomoću *swimlanea*. Svaki *swimlane* pokazuje naziv (na vrhu) i ulogu aktivnosti,

Slika 3.18.:





Slika .19.Primer

Kupovina

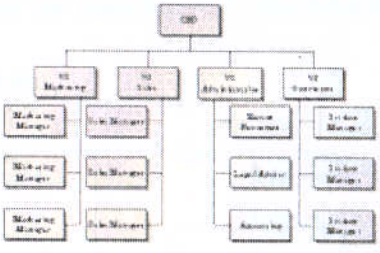
Organizacioni dijagram (Organization chart)

U okviru fizičkog modela poslovnih procesa potrebno je predstaviti okruženje ovih procesa u smislu organizacione strukture ili sistematizacije radnih mesta. To ćemo uraditi putem organizacionog dijagrama (*organization chart*).

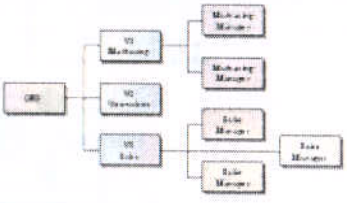
Organizacioni dijagram je organizaciona struktura ili organizaciona (ustrojena) šema koja predstavlja organizacionu šemu u obliku stabla i ilustruje:

- odnose (relacije) među ljudskim resursima,
- sektorima (odelima) organizacije ili
- podele unutar organizacije.

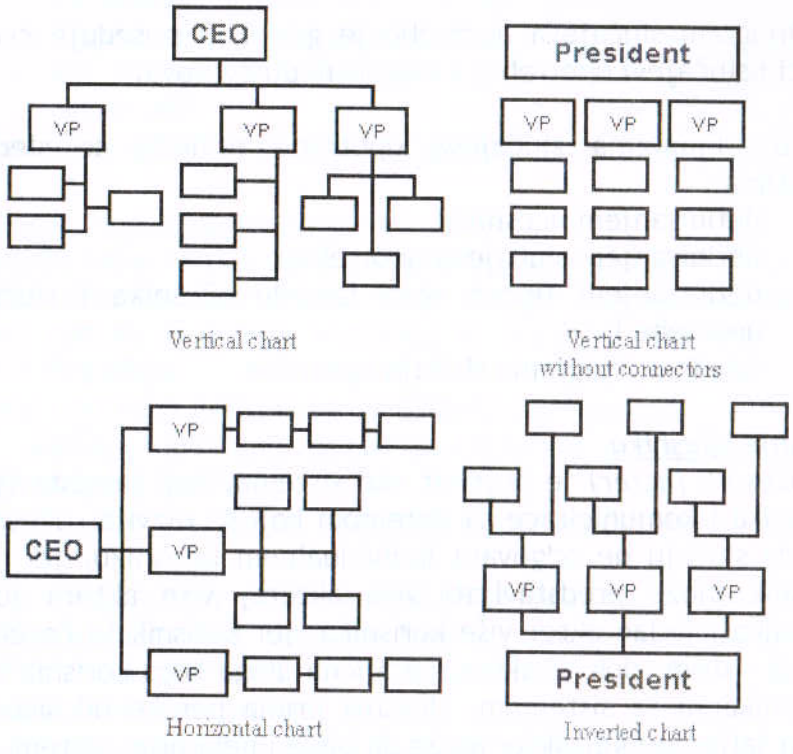
Ova šema se može crtati kao *vertikalno* stablo ili *horizontalno* stablo, Slika 3.20:



Vertikalni organization chart



Horizontalni organization chart



Slika 3.20

UML-ov Use case dijagram

Use case predstavlja sistem sa stanovišta korisnika i pomaže nam da shvatimo korisničke zahteve. Dijagrami slučajeva upotrebe *(use case)* se definišu na osnovu *funkcionalnog modela sistema* i opisuju interakciju objekata koji se nalaze van sistema sa samim informacionim sistemom.

Skup slučajeva upotrebe predstavlja sve pretpostavljene načine korištenja sistema, tj. određuje ponašanje sistema ili njegovog dela i opisuje akcije koje sistem izvodi da bi postigao rezultate koji su od koristi korisniku.

Slučajevi upotrebe se koriste da bi se ostvarilo željeno ponašanje sistema koji se razvija, bez obaveze da se odrede načini realizacije ponašanja. Slučajevi upotrebe koriste krajnjim korisnicima da razumeju sistem.

Dijagram slučajeva upotrebe je graf koji poseduje čvorove (učesnici i slučajevi upotrebe) i veze između čvorova.

Razvoj dijagrama slučajeva upotrebe definiše se sledećim aktivnostima:

- definisanjem učesnika,
- definisanjem slučajeva upotrebe,
- definisanjem tipova veza između učesnika i slučajeva upotrebe i
- izradom dijagrama slučaja upotrebe.

Definisanje učesnika

Učesnik (acter) je objekat van sistema, koji predstavlja tip korisnika i komuniciraće sa sistemom koji se razvija. Učesnik ni u kom slučaju ne odgovara individualnom korisniku, jer jedan korisnik može predstavljati više aktera, više aktera jednog korisnika i jedan akter više korisnika, jer korisnik je čovek koji koristi sistem, dok je akter specifična uloga koju korisnik ima u komunikaciji sa sistemom. Učesnik prima poruke od sistema i istom šalje poruke. Akter može da bude i neki drugi sistem.

Učesnik je jedna vrsta klase i može da bude u svim relacijama koje važe i za klase. Jedna važna relacija je generalizacija, gde izvedeni učesnik ima sve osobine osnovnog učesnika i može da učestvuje u svim slučajevima upotrebe kao i osnovni učesnik.

Učesnike je moguće identifikovati na osnovu odgovora na sledeća pitanja:

- Ko će koristiti osnovnu funkcionalnost sistema (primarni akteri)?
- Kome će biti potrebna podrška sistema u obavljanju dnevnih zadataka?
- Ko treba da upravlja, administrira i održava sistem(sekundarni akteri)?
- Kojim hardverskim uređajima treba da upravlja sistem?
- S kojim drugim sistemima dotični sistem treba da bude u vezi? Ti sistemi mogu da se podele na sisteme koji će da iniciraju komunikaciju sa našim sistemom i na one koje će naš sistem da kontaktira. Sistemi mogu da budu

računarski sistemi, kao i druge aplikacije na računaru u kojima se sistem nalazi.
- Ko ili šta je zainteresovan za rezultate koje sistem proizvodi?

Učesnici su u tesnoj vezi sa slučajevima upotrebe.

Definisanje slučajeva upotrebe

Opisuje karakteristične sekvence akcija u tipičnim situacijama upotrebe sistema, što znači da je to tehnika kojom se snima poslovni proces sa strane korisnika ili scenario koji opisuje skup sekvenci događaja.

Slučajevi upotrebe grafički se predstavljaju elipsom i imenom.

Naziv slučaja upotrebe treba da bude ne predugačak i jasan (tipično glagolski oblik).

Slučaj upotrebe opisuje se tekstualno gdje se detaljno opisuje *šta radi*.

- Svaki primitivni proces predstavlja jedan slučaj upotrebe.
- Slučaj upotrebe definiše funkcionalnost sistema sa strane korisnika. Korisnici ga iniciraju. Svako izvođenje slučajeva upotrebe = **instanca**.
- Opisuje normalan način rada ne uzimajući u obzir moguće greške ili anomalije → šablon ponašanja delova sistema.

Zaključak: logička jedinica posla ili sekvencija bliskih transakcija koje izvode korisnici u dijalogu sa sistemom.

Karakteristike slučajeva upotrebe:

- slučaj upotrebe uvek inicira učesnik, tj. izvršava se na zahtev učesnika koji mora direktno ili indirektno da «naredi» sistemu da izvrši neki slučaj upotrebe (ponekad učesnik ne mora da bude svestan da je on inicirao slučaj upotrebe);
- slučaj upotrebe pruža neku vrednost za učesnika, koja ne mora uvek da bude upadljiva, ali mora da bude prepoznatljiva;
- slučaj upotrebe mora da bude definisan kao potpun opis, a nije potpun sve dok se neka vrednost ne proizvede;
- česta greška je podeliti jedan slučaj na manje slučajeve upotrebe, koji se implementiraju jedni druge, slično funkcionalnim pozivima u programskim jezicima;
- slučaj upotrebe je jasno uočljivo ponašanje sistema koje se može testirati;

- može se shvatiti i kao odgovor sistema na neki događaj ili spoljnu akciju koju proizvede učesnik.
- slučaj upotrebe će biti implementiran pomoću scenarija koji ispunjava dato ponašanje sistema;
- slučaj upotrebe navodi samo šta sistem radi, ali ne i kako to radi;
- specifikacija slučaja upotrebe predstavlja skup opisa tokova događaja, uključujući i varijante (izuzetne situacije, situacije sa greškama i sl.);
- slučajeve upotrebe treba navoditi eksplicitno, u vidu tačaka nabrajanja;
- za svaki slučaj upotrebe treba navesti ko ga i kako pokreće (inicira), kao i precizan tok događaja, sa detaljima o tome koje informacije, i kada, učesnik i sistem razmenjuju;
- opis slučaja upotrebe može da ima navedene i ostale zahteve vezane za ponašanje: izgled korisničkog interfejsa, zahteve u pogledu performansi, zahteve u pogledu pouzdanosti i slično.

Za svakog učesnika mogu se definisati sledeća pitanja, kojima se identifikuju slučajevi upotrebe:

- Koje funkcije učesnik zahteva od sistema?
- Da li učesnik treba da čita, kreira, briše, izmeni, ili da unese neke informacije u sistem?
- Da li učesnik treba da bude obavješten o događajima u sistemu i da li učesnik treba da obavesti sistem o promenama u okruženju? Šta ovi događaji predstavljaju u odnosu na funkcionalnost?
- Da li svakodnevni rad učesnika može da se pojednostavi kroz nove funkcije sistema (obično funkcije koje trenutno nisu automatizovane kroz sistem).
- Kakav ulaz/izlaz zahteva sistem? Gdje ulaz/izlaz ide, odnosno odakle dolazi?
- Koji su glavni problemi trenutne implementacije sistema?

Na osnovu postavljenih pitanja odgovori se definišu u okviru scenarija, odnosno tekstualnog opisa slučajeva upotrebe.

Tekstualni opis slučajeva upotrebe je poseban dokument koji služi za dalju detaljizaciju informacija definisanih u modelu slučaja upotrebe.

Svrha opisa slučaja upotrebe je definisanje prioriteta za izvršavanje grupa logičkih zahteva i određivanje nivoa prioriteta baziranih na važnosti identifikovanog opsega projekta.

Definisanje veza između korisnika i slučajeva upotrebe

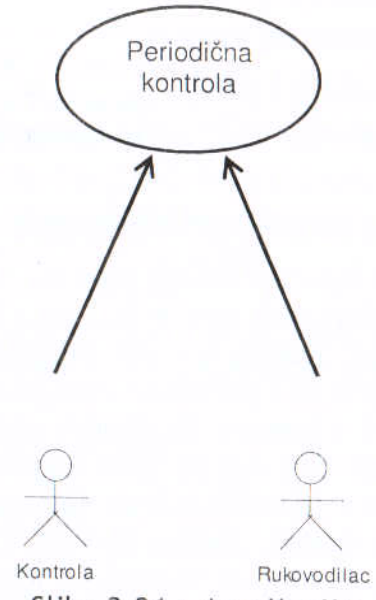
Slučajevi upotrebe po pravilu nisu nezavisni. Analizom uočavamo relacije između slučajeva upotrebe, jer se time složena funkcionalnost dekomponuje i uočavaju se zajednički delovi. ***Jedan čvor učesnika uvijek je povezan barem sa jednim čvorom slučaja upotrebe i obrnuto - jedan čvor slučaja upotrebe povezan je sa barem jednim čvorom učesnika.***

Direktna interna komunikacija između konkretnih slučajeva upotrebe nije dozvoljena. Međutim, mogu se definisati asocijacije između slučajeva upotrebe i između učesnika, da bi se jednostavnije prikazao neki složeni model.

U okviru UML standarda definiršu se sledeće veze:

- asocijacija (association)
- asocijacija između slučajeva upotebe
  <<include>> ili <<extends>>
- generalizacija (generalization-inheritance)
- zavisnost (depedency)

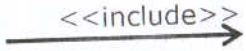
Asocijacija je bidirekcionalna veza koja spaja učesnike i slučajeve upotrebe. Asocijacija između samih učesnika, ili slučajeva upotrebe, definiše logičku povezanost tih elementata.



Slika 3.21.: Asocijacije

Asocijacije između slučajeva upotrebe su: *include* i *extend*.

<<include>>



U složenim sistemima se pojavljuju slučajevi upotrebe sa sličnim ponašanjem. Ovo slično ponašanje se izdvaja i dele ga slični slučajevi upotrebe.

3.3.2.2.Objektno orjentisana analiza

Proces "**Objektno orjentisana analiza**" razmatra objekte sadržane u realnom sistemi kao i njihove međusobne odnose. Objektno orijentisana analiza (OOA) ističe u prvi plan istraživanje problema tj. pronalaženje i opisivanje objekata ili koncepta u domenu problema, ne dajući odgovore na pitanje kako su rešenja definisana. OOA prestavlja najkritičniju fazu jer je potrebno uočiti koji se sve objekti pojavljuju u realnom sistemu. Zatim se vrši specifikacija najvažnijih atributa unutar objekata i interakcija između objekata. Analiza je proces ispitivanja korisničkih zahteva u cilju njihovog definisanja na osnovu čega se pristupa objektno orijentisanom dizajnu (OOD).

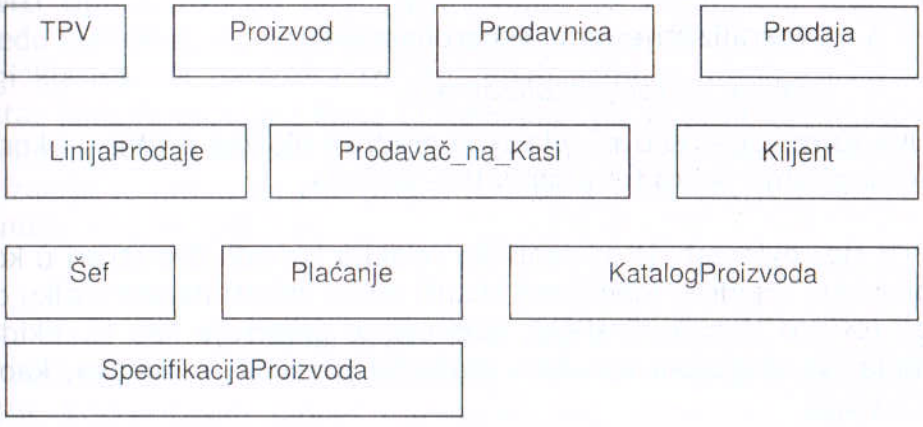
Ovaj proces se definiše kao:

- Izrada konceptualnog modela
- Izrada dijagrama sekvenci i
- Definisanje ugovora o izvršenju operacije

3.3.2.2.1.Izrada konceptualnog modela

Aktivnost "Izrada konceptualnog modela" treba da definiše dijagram klasa bez definisanih operacija za ciljne koncepte unutar domena posmatranja. Konceptualnim modelom definišu se koncepti, odgovarajući atributi i potrebne asocijacije između koncepata. Konceptualni model odgovara klasičnom modelu objekti veze. U okviru ove aktivnosti definiše se lista kandidata korištenjem liste kategorije koncepta i identifikacijom imenica iz fraza, potom se kreira konceptualni model, dodaju atributi i odgovarajuće asocijacije. Da zaključimo u ovoj aktivnosti se identifikuju uloge ljudi i stvari od interesa koji će biti angažovani u procesima.

U UML-u konceptualni model odgovara **diagramu klasa** sa: konceptima, relacijama među konceptima i atribitima koncepata. Prilikom identifikacije koncepte prepoznajemo u relevantnim imenicama. Bolje je preterati u specifikaciji konceptualnog modela sa mnogo koncepata (fine - grained) nego da bude nedovoljan broj koncepata.



Slika 3.22.: Identifikovani koncepti konceptualnog modela

Nakon idetifikacije koncepata potrebno ih je klasificirati po kategorijama koristeći **popis kategorija** koncepata.

Popis kategorija I
- Fizički objekti *(TPV, Avion)*,
- Opisi *(SpecifikacijaProizvoda, OpisLeta)*,
- Mesta *(Prodavnica, Aerodrom)*,
- Transakcije *(Prodaja, Plaćanje, Rezervisanje)*,
- Delovi transakcija *(ProdajaLinijaProizvoda)*,
- Uloge osoba *(Prodavač na kasi, Pilot)*,
- Kontejneri *(Prodavnica, Avion)*,
- Stvari unutar jednog kontejnera *(Proizvodi, Putnici)*,
- Drugi računarski sistemi *(SistemAutorizacijeKredita)*,

Popis kategorija II
- Organizacije *(Odsek za prodaju)*,
- Događaji *(Prodaja, Let)*,
- Procesi *(Rezervacijajedišta)*,
- Pravila *(Pravila plačanja)*,
- Katalozi *(KatalogProizvoda)*,
- Registri, ugovori *(Vračanje, Ugovor_o_Zapošljavanju),itd*

Identifikirane koncepte je potrebno povezati relacijama (vezama među konceptima). **Relacija** je odnos između dva ili više koncepata koja ukazuje na vezu između njih.
Postoje četiri tipa relacija unutar UML modela:

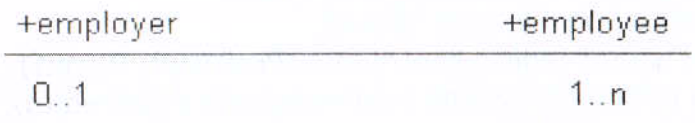
1. Ovisnosti *(eng. dependency)*
2. Asocijacije *(eng. association)*
3. Generalizacije *(eng. generalization)*
4. Realizacije *(eng. realization)*

Ove su relacije osnovni relacijski gradivni blokovi u UML-u i koriste se za pisanje dobro formisanih UML modela.

Prvi tip, *ovisnost*, je semantička relacija između dve stvari u kojoj promena u jednoj (neovisnoj stvari) može uticati na semantiku druge (ovisne stvari). Grafički, ovisnost se prikazuje kao isprekidana linija, sa mogućom oznakom direkcije (strelica) i imenom, kao na Slici 3.23.

Slika 3.23: Ovisnost

Drugi tip, *asocijacija*, je strukturalna relacija koja opisuje skup veza između objekata. *Kombinacija (eng. aggregation)* je specijalan oblik asocijacije i reprezentuje strukturalnu relaciju između celine i njenih delova. Grafički, asocijacija se predstavlja kao puna linija, uz mogućnost određivanja smera, i ponekad sadrži i dodatne stvari kao kardinalitet (n-arnost) i ime. Prikazana je na Slici 3.24.



Slika 3.24.: Asocijacija

Treći tip, *generalizacija (eng. generalization)*, je relacija specijalizacije/generalizacije u kojoj objekti specijaliziranih elemenata (deca) se mogu zameniti objektima generaliziranih elemenata (roditelja). Na ovaj način, deca dele strukturu i ponašanje roditelja. Grafički, relacija generalizacije se prikazuje kao puna

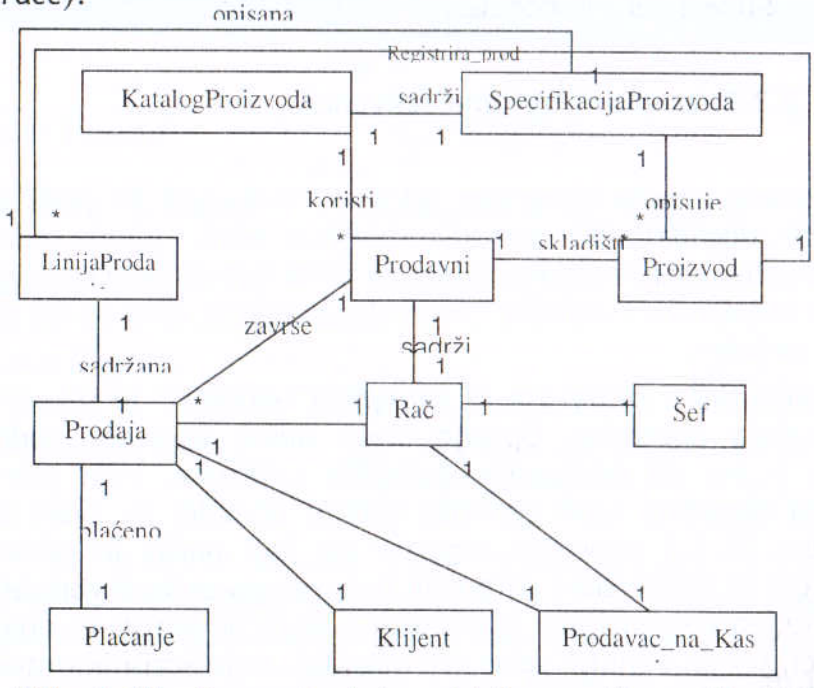
linija sa šupljom strelicom koja pokazuje prema roditelju, kao na Slici 3.25.

Slika 3.25.: Generalizacija

Četvrti tip, *realizacija (eng. realization)*, je semantička relacija između klasifikatora, gdje jedan klasifikator specificira ugovor koji drugi klasifikator garantira izvršiti. Realizacija se susreće na dva mesta: između sučelja i klasa ili komponenata koje ih realizuju, i između slučajeva korištenja i saradnji koje ih realizuju. Grafički, realizacija se prikazuje kao prelaz između generalizacije i relacije ovisnosti, kao na Slici 3.26.

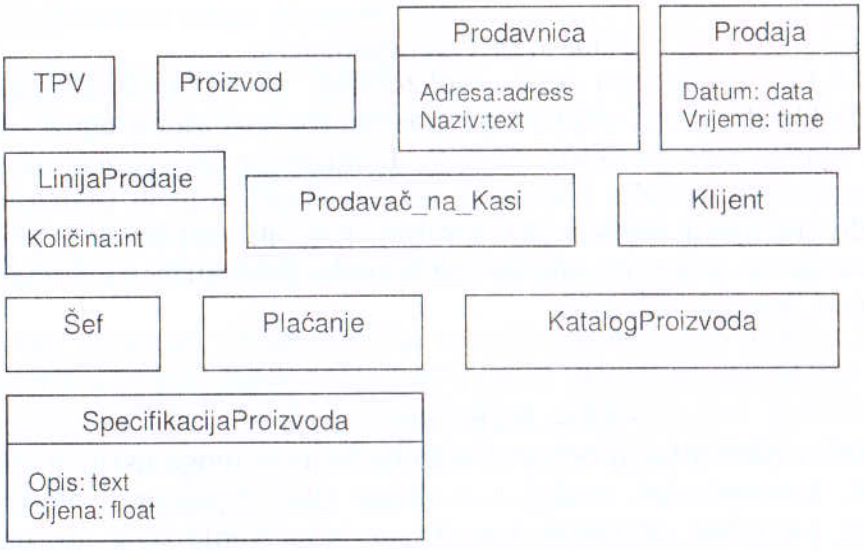
Slika 3.26.: Realizacija

Ova četiri elementa su osnovne relacije koje se mogu uključiti osim u UML konceptualni model i u druge UML dijagrame. Postoje, takođe, i njihove varijacije, kao što su uključivanje *(eng. include)*, proširenje *(eng. extend)*, rafiniranje *(eng. refinement)*, i praćenje *(eng. trace)*.



Slika 3.27.: Konceptualni model (koncepti, asocijacije i kardinaliteti)

Konceptima je potrebno dodeliti sopstvene atribute predviđene zahtevima. Atributi su važne karakteristike koncepata za domenu problema.



Slika 3.28.: Konceptualni model (koncepti i atributi)

3.3.2.2.2.Izrada dijagrama sekvenci

Aktivnost "Izrada dijagrama sekvenci" pokazuje za svaki slučaj upotrebe, događaje koje generiše spoljni učesnik i njihov redosled. Drugim rečima sekvencijalni dijagram prikazuje dinamičku saradnju između objekata u vremenu. Ovom aktivnošću se opisuje šta sistem radi, a ne kako.

Sekvencijalni dijagram ili dijagram sekvenci: je sekvencijali opis slučaja upotrebe. Opisuje svaki slučaj upotrebe, odnosno opisuje jednu od realizacija slučajeva upotrebe, koja pokazuje redosled dagađaja koje generišu spoljni učesnici za svaki slučaj upotrebe. Slučaj upotrebe sugeriše na koji način je učesnik u interakciji sa softverskim sistemom (*učesnik generiše događaje*).

Dakle, za sekvencijalni opis slučaja upotrebe koristi se dijagram sekvenci, jer on definiše sekvencu događaja koje korisnik (interfejs) prosleđuje sistemu u jednom slučaju upotrebe. Sistem je «crna kutija» koja pokazuje komunikaciju korisnika sa sistemom.

Dijagram sekvenci se koristi za specifikaciju vremenskih zahteva u opisu složenih scenarija, tj. za opis toka poruka između objekata kojima se realizuje odgovarajuća operacija u sistemu. Na dijagramu sekvenci se ne prikazuju veze između objekata, već se prikazuje šta sistem radi, tj. identifikuju se sistemski događaji i sistemske operacije za odgovarajući slučaj upotrebe.

Vreme u sekvencijalnom dijagramu se prikazuje u vertikalnoj, a objekti u horizontalnoj dimenziji.

Dijagram sekvenci je jedan od dijagrama interakcije. Interakcija je ponašanje koje obuhvata skup poruka koje se razmenjuju između skupa objekata u nekom kontekstu sa nekom namenom. Poruka je specifikacija komunikacije između objekata koja prenosi informaciju. Nakon poruke očekuje se da usledi aktivnost. Prijem jedne poruke se smatra instancom jednog događaja. Kada se pošalje poruka sledi akcija, odnosno izvršenje naredbe koja predstavlja apstrakciju metode.

Vrste akcija UML-a na osnovu poslatih poruka:

Tabela 3.1.

Akcija	Opis
Poziv (call)	Pokreće operaciju objekta primaoca (može pozivati i sam sebe)
Povratak (return)	Vraća vrednost primaocu
Slanje (send) operacija	Asinhrono se šalje signal primaocu
Kreiranje (create) <<create>>	Kreira se objekat
Uništavanje(destory) <<destory>>	Uništava se objekat (objekat može biti i suicidan)
Postajanje (become) <<become>>	Objekat menja prirodu (na obe strane veze je isti objekat)

Interakcija se koristi za modeliranje dinamičkih aspekata modela definisanih preko dijagrama interakcije koji čine dijagram sekvenci i dijagram kolaboracije. Dijagrami sekvenci naglašavaju vremensko uređenje interakcije, a kolaboracijski dijagram naglašava strukturnu vezu između učesnika u interakciji. Dijagram sekvenci je povoljniji za prikaz kompleksnih slučajeva upotrebe u interaktivnim

sistemima i sistemima koji rade u realnom vremenu, dok je kolaboracijski dijagram povoljniji za opis složenih algoritama.

Ove dve vrste dijagrama vizuelizuju na različite načine iste informacije. Semantički su potpuno ekivalentni i CASE alati ih automatski konvertuju jedan u drugi.

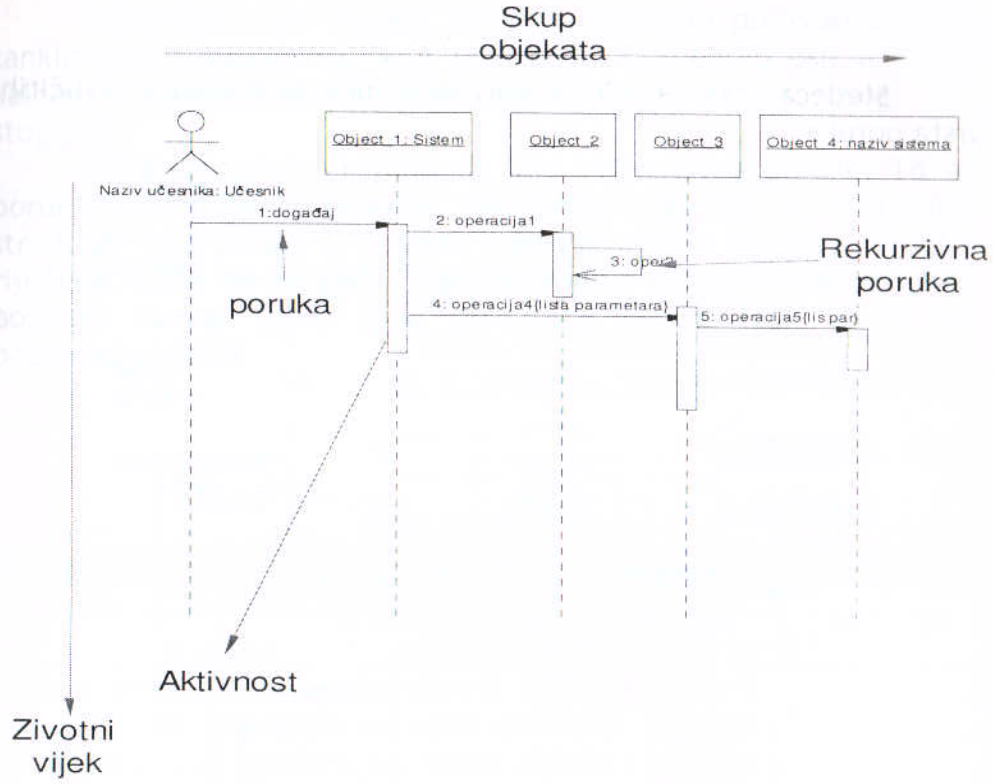
U okviru izrade dijagrama sekvenci definišu se sledeće aktivnosti:

1. definisanje objekata,
2. definisanje poruka,
3. definisanje indikatora sinhronizacije.

1.Definisanje objekata

Objekti u sekvencijalnim dijagramima su predstavljeni vertikalnim linijama. Na vrhu linije se navodi naziv objekta i/ili simbol objekta. Aktiviranje objekata se predstavlja *uskim pravougaonikom* na liniji objekta, a predstavlja operaciju (akciju) koju objekat, u periodu predstavljenog dužinom aktivacije, obavlja. Na vrh aktiviranog objekta se prikazuje događaj (poruka) koju je aktivirao objekat, a na dnu, povratna poruka objektu koji je aktivirao promatrani objekat. Povratna poruka često se ne prikazuje.

Sekvencijalni dijagram se formira tako što se prvo na vrh dijagrama duž ose X, postave objekti koji učestvuju u interakciji. Obično se objekti koji započinju interakciju stavljaju levo, a objekti koji slede redom desno. Nakon toga poruke koje ovi objekti šalju i primaju smeštaju se duž ose u rastućem nizu po vremenskom redosledu od vrha ka dnu. Ovo pruža jasnu asocijaciju na tok kontrole u protoku vremena.



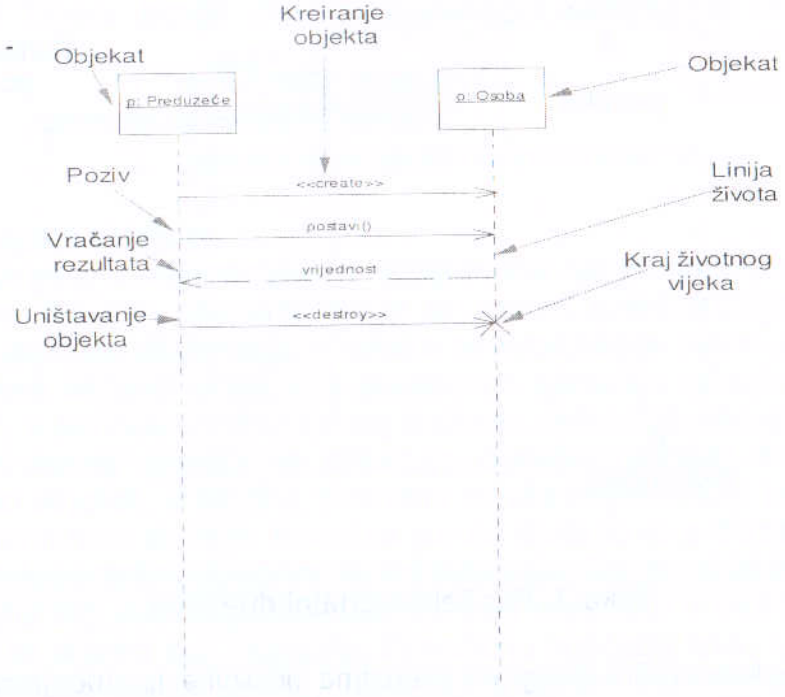
Slika 3.29.: Sekvencijalni dijagram

Sekvencijalni dijagram preuzima učesnike iz slučajeva upotrebe, a ako se prvo definiše konceptualni dijagram, preuzimaju se objekti. U sekvencijalnom dijagramu učesnici mogu biti korisnik ili veštački entitet (software ili entitet). Objekt je instanca koncepta koji šalje i prima poruke. Moguće je da objekat pošalje poruku sam sebi. Objekti se definišu nazivom objekta i nazivom koncepta, /koji su podvučeni/:

Naziv objekta: Naziv koncepta

Objekti se prikazuju u pravougaonicima i ime objekta se obavezno unosi. Iz svakog objekta polazi vertikalna isprekidana linija prema dole, koja predstavlja životni vek objekta *(lifeline)*. Životni vek objekta je vremenski period postojanja istog. Većina objekata postoji dok traje interakcija, ali objekti se mogu praviti i dok traje interakcija. Njihov životni vijek počinje nakon prijema poruke <<create>>. Objekti mogu biti uništeni u toku interakcije. Njihov životni vek se završava nakon prijema poruke *destroy*.

Sledeća slika je primer sekvencijalnog dijagrama i različitih vrsta poruka:



Slika 3.30.: Različite vrste poruka u sekvencijalnom dijagramu

<<create>> → kreira objekat za koncept osoba

- postaviti() → postavlja se osoba na radno mjesto
- akcija vrijednost → vraća se rezultat u objekat p koncepta preduzeće
- <<destroy>> → uništava se objekat koncepta osoba

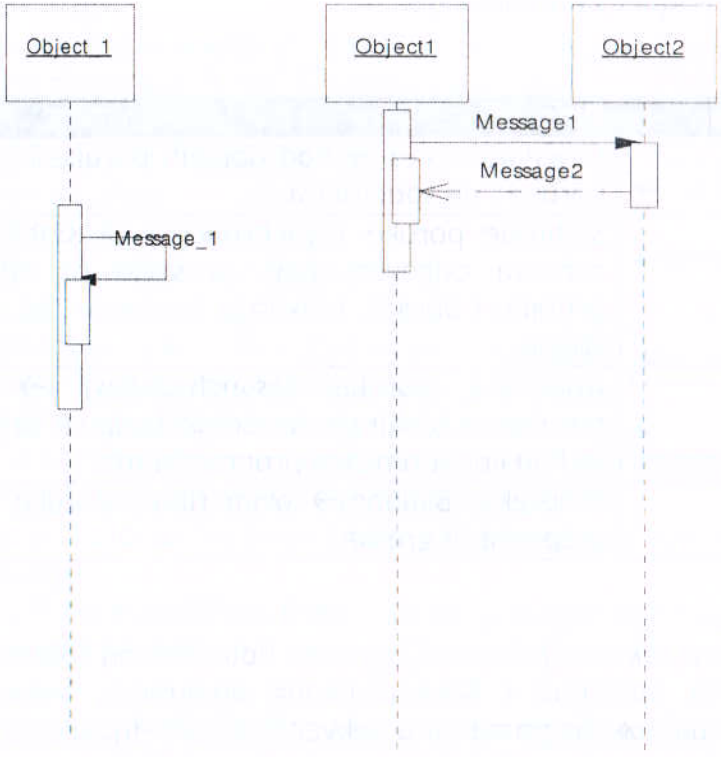
2.Definisanje poruka

Porukama (događajima) komuniciraju objekti. Poruke se prikazuju usmerenim linijama.

Argumenti se navode u običnim, a uslovi tranzicije u uglastim zagradama.

Aktivnosti koje objekat izvodi u vremenu predstavljaju se tankim pravougaonicima, koji prekrivaju vremensku osu u vertikalnom pravcu i u veličini koja odgovara vremenu njegovog postojanja.

Dijagram sekvenci može da ima i rekurzivne poruke. To su poruke koje objekat upućuje sam sebi. Rekurzija je prikazana strelicom koja polazi iz objekta i završava se u objektu. Drugim riječima može se prikazati ugnježđavanje (izazvano rekurzijom, pozivom samog sebe) zbog povratnog poziva (*call back*) od pozvanog objekta.



Slika 3.31.: Ugnježđene i rekurzivne poruke

Dakle, objekti međusobno komuniciraju slanjem *poruka*. Poruke se predstavljaju strelicama, koje polaze od jednog objekta, a završavaju na drugom. Poruka (Message) je definisana nazivom i

parametrima. Parametrima se definiše jedan ili više argumenata. Poruka je opisana na sledeći način:

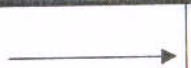
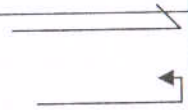
Naziv poruke (lista parametara)

Uz naziv poruke može se navesti i broj kojim se definiše redosled izvođenja poruka. Lista parametara je opciona i male zagrade se obavezno navode bez obzira na listu parametara.

3.Definisanje indikatora sinhronizacije
Indikatorom sinhronizacije definiše se tip poruke.

Različiti oblici sinhronizacije:

Tabela 3.2:

Oznaka	Opis
	Obična(Simple) → kod običnih poruka nije bitan tip poruke i ne specificira se.
	Sinhrone poruke (Synchronous) → sinhrona poruka zahteva odgovor, koji se šalje sa odredišta po prijemu i obradi, blokiraju izvršenje dok odgovor ne stigne.
	Asinhrona poruka (Asynchronous) → asinhrone poruke ne blokiraju izvršenje kada se šalju i koriste se kod konkurentnog programiranja.
	Prepreka (Balking)→ ovim tipom poruke se definiše prepreka ili greška.

Ukratko o sekvencijalnom dijagramu: Polazimo od tekstualong opisa slučajeva upotrebe i konceptualnog dijagrama, zatim koncepti konceptualnog diagrama su u sekvencijalnom dijagramu objekti. Iz tekstualnog opisa slučajeva upotrebe definišu se događaji i operacije za objekte. Dijagram sekvenci može poslužiti da se operacije pridruže klasama u dijagramu klasa.

3.3.2.2.3.Definisanje ugovora o izvršenju operacije

Aktivnost "Definisanje ugovora o izvršenju operacije" daje opis logike operacije koje predstavljaju odziv sistema na specificirane događaje.

Sekvencijalni dijagram sadrži operacije sistema, **gde se opisuje ponašanje operacija i rezultati njihove primene?**

U slučajevima upotrebe se opisuju događaji koji se pojavljuju u sistemu.
Ukoliko su operacije kompleksne ili slučajevi upotrebe nisu detaljni, opis ponašanja operacija može biti nezadovoljavajući (npr. može biti opširan i nekompletan).

Ugovor o izvršenju operacija sistema koje su komplikovane ili nedetaljne daćemo u funkciji sledećih koncepata:

- Zaglavlje operacije
- Slučajevi upotrebe gdje se javlja ova operacija (opcionalno)
- Preduslovi operacije
- Postconditions operacije

Postcondition jedne operacije opisuje promene prouzrokovane ovom operacijom na objektima iz modela domena.
Ove promene se mogu sintetizovati na sledeći način:

- Instance koje su kreirane i destruirane ovom operacijom
- Atributi koji su modificirani ovom operacijom
- Asocijacije koje su kreirane i koje su destruirane primenom ove operacije

Da bismo opisali *postconditions* koristimo se modelom domena koji smo prethodno konstruirali.

*Primer ugovora:*Ako imamo sledeći sekvencijalni dijagram sistema:

Prof. dr Dragica Radosav

SOFTVERSKO INŽENJERSTVO I

Univerzitet u Novom Sadu
Tehnički fakultet "Mihajlo Pupin"
Zrenjanin, 2005.

0.UVOD

Pojam softversko inženjerstvo prvi put se pominje krajem šezdesetih godina XX veka, na konferenciji o krizi softvera, prouzrokovanoj trećom generacijom računara (prije svega IBM S/360), koja je svojom snagom inicirala razvoj velikih softverskih sistema. Pošto za ove sisteme nisu postojali odgovarajući teorijski modeli razvoja, realizacija projekata pojedinih informacijskih sistema je trajala izuzetno dugo, kasnila po više godina, što je višestruko uvećavalo planirane troškove razvoja, komplikovalo održavanje, a sama realizacija je bila na vrlo niskom nivou. Tako je počeo rad na teorijskim aspektima metodologija razvoja softvera, kasnije na njihovoj primjeni u praksi, da bi se danas jasno formirala disciplina koja proučava razvoj informacijskih tehnologija, poznata pod nazivom softversko inženjerstvo.

Pojmovno softversko inženjertsvo se može definisati kao stroga primena inženjeringa, naučnih i matematičkih principa i metoda u ekonomičnoj proizvodnji kvalitetnog softvera.

Softverski inžinjering je prvi definisao Fritz Bauer još 1968. godine na konferenciji koju je organizovao Naučni komitet NATO. Upotrijebio ga je kao termin koji označava primjenu principa inženjeringa u cilju ostvarivanja ekonomičnog softvera, efikasnog i pouzdanog u realnosti na računarskim mašinama. Nakon toga su nastale brojne definicije, ali su sve one posebno naglašavale potrebu primjene inženjeringa u razvoju softvera. Najpreciziju definiciju softverskog inženjeringa nalazimo u standardnom rečniku termina softverskog inženjeringa koji je 1990. godine objavio IEEE (Institute for Electronics, Energetics and Engineering), a prema kojem ***softverski inženjering podrazumjeva primjenu sistematičnog, disciplinovanog i merljivog pristupa razvoju, uvođenju i održavanju softvera***, *tj. primjeni inženjeringa na softver.*

Softverski inženjering je posledica hardverskog i sistemskog inženjeringa. Strukturu softverskog inženjeringa sačinjavaju tri ključne komponente: ***metode***, ***alati*** i ***postupci*** (procedure). Njihovo jedinstvo opredjeljuje kvalitet razvoja, pa je zbog toga zna-

čajno da se odaberu one komponente koje se postavljenim zadacima i problemima u razvoju najlakše prilagođavaju. Donošenje prave odluke nije jednostavan zadatak zbog:

- individualnog i kreativnog karaktera postupka projektovanja,
- različitosti pojedinih sistema za koji se razvija softver i
- različitog okruženja sistema.

Značajno je istaći da se za rešavanje konkretnog zadatka mogu izabrati najpogodnije komponente samo ukoliko projektanti poznaju njihove mogućnosti, alternative i znaju ih uspešno primeniti.

1. TIPOVI I KARAKTERISTIKE SOFTVERSKIH PROIZVODA (Types and characteristics of software products)

Softverski proizvod (engleski: consumer software package) predstavlja softversku podršku računarskim sistemima, a čine ga skup računarskih programa, datoteke i odgovarajuća dokumntacija, namenjeni realizaciji određenih zadataka (definicija preuzeta iz standarda JUS ISO 9127/94). Za razliku od softverskog proizvoda, softver se sastoji iz skupa računarskih programa i datoteka sa istom namenom koju ima softverski proizvod. Kako su, prema usvojenim standardima, softver i odgovarajuća prateća dokumentacija nerazdvojivi, upakovani za prodaju kao jedinstvena celina i prodaju se zajedno, u nastavku se ova dva pojma poistovećuju.

Iz izloženog se vidi da softverski proizvod predstavlja jedinstvo računarskih programa, struktura podataka sadržanih u datotekama i dokumentacije koja opisuje način funkcionisanja i upotrebe programa. Pri tome, posebno se ističe značaj dokumentacije, koju projektanti i programeri često zanemaruju, ali bez koje se softver ne može smatrati kompletnim. Informacije sadržane u pratećoj dokumentaciji su često jedino sredstvo putem koga proizvođač softvera i/ili distributer mogu komunicirati sa kupcem ili korisnikom. Samo ukoliko dokumentacija sadrži dovoljno informacija, korisniku se omogućuje uspešno korišćenje softvera.

Za pojam softvera vezan je i pojam ***softverska podrška***. Ona predstavlja rad na održavanju softvera i prateće dokumentacije u funkcionalnom stanju. Mogu je pružati: proizvođač (organizacija koja je razvila softver), predstavnik (organizacija koja plasira softver na tržištu), distributer (organizacija koja neposredno prodaje softver korisniku) ili neka druga organizacija.

Trend projektovanja softverskih proizvoda na osnovu elemenata ranijih aplikacija nezavisno razvijenih, sa različitim alatima, u različitim jezicima i na različitim platformama, postaje sve izraženiji. Ove, prethodno testirane komponente, pružaju mogu-

ćnosti jednostavnijeg i pouzdanijeg sklapanja u softverski proizvod, zatim njihovo višestruko korištenje i nezavisno poboljšanje mogućnosti i performansi.

Pojam ***softverskih komponenti*** podrazumeva sistemski ili aplikativni softver pomoću kojeg se upravlja fizičkim ili logičkim resursima sistema. Da bi se softverske komponente mogle koristiti pri navedenom načinu projektovanja potrebno je da zadovoljavaju jedinstven model razmene podataka međusobno, da postoji mogućnost neposrednog programiranja aktivnosti koje se odvijaju između komponenti, zatim da se u istoj datoteci mogu naći različiti podaci i da se mogu izvršavati na različitim platformama.

Skladišta softverskih komponenti koje zadovoljavaju opisane uslove i mogu se višestruko koristiti u različitim programima nazivaju se repozitorijumi. Pri formiranju, izmenama i upotrebi komponenti iz repozitorijuma treba voditi računa o tome da li nova verzija utiče i kako utiče na izvršenje pojedinih programa koji je koriste. Takođe je potrebno pratiti proces zamene komponenata novim verzijama u pojedinim programima i pri tome voditi računa o prilagođavanju komponente softverskom proizvodu.

Softverske komponente mogu se klasifikovati u sledeće funkcionalne grupe:

- seniorske (skupljaju informacije iz okoline sistema),
- pokretačke (prouzrokuju izmene u okolini sistema),
- računske (na osnovu određenih ulaza proračunavaju izlaze),
- komunikacijske (omogućavaju komunikaciju ostalih softverskih komponenti),
- koordinacijske (koordiniraju operacije ostalih komponenti) i
- interfejsi (transformišu prezentaciju izlaza jedne komponente u oblik razumljiv drugoj).

Tehnologije grupisane oko servisa WWW (World Wide Web) na Internetu svojim snažnim razvojem nametnutim opštom primenljivošću, a zasnovane na specifičnim funkcijama WWW interfejsa, otvorile su prvo pravo tržište gotovih softverskih komponenti razli-

čitih proizvođača. Ove komponente karakteriše standardizacija okruženja koja je nezavisna od platforme i operativnog sistema na kojima se one koriste. Primenom ovih komponenti ubrzava se razvoj novih, čime učešće gotovih komponenti postaje dominantno u realizaciji distribuiranih informacijskih sistema.

Drugi poznati primer softverskih komponenti predstavljaju takozvani šabloni (design patterns) koji se primenjuju u objektno orjentisanom dizajnu i programiranju. Pod ovim terminom podrazumevaju se tipske uloge, strukture, odnosi i mehanizmi obuhvaćeni jedinstvenim rešenjem koje je zajedničko za više softverskih proizvoda. Vrlo je popularna literatura u kojoj su dati i detaljno opisani brojni šabloni koji se mogu efikasno koristiti u razvoju objektno orjentisanih programa. Za njihov opis, pored imena, potrebno je navesti i niz drugih specifikacija kao što su *namena*, *primenljivost*, *struktura*, *učesnici* i *njihove veze*, *implementacija*, *primeri korištenja* i drugo.

Razlikujemo dve klase softverskih proizvoda:

- ***generički proizvodi*** su samostalni sistemi koje je izradila razvojna organizacija i prodaje ih na otvorenom tržištu zainteresovanim kupcima,
- ***proizvodi po meri*** su sistemi specijalno formirani za određenog kupca.

Do 1980.godine, dominirali su proizvodi po meri, koji su se izvodili na velikim računarima i bili su skupi, jer je kompletan razvoj plaćao jedan kupac. Tržište personalnih računara donelo je tržišnu prevagu generičkih proizvoda, koji se prodaju u hiljadama primeraka i zbog toga su znatno jeftiniji. Ipak, još uvek se više napora ulaže u izradu proizvoda po meri. Integrisana aplikativna rešenja sa značajnim referencama i znatno nižim cenama utiču na to da se danas tržište sve više orjentiše na generičke proizvode.

Danas se razlikuje više različitih klasifikacija softverskih proizvoda po raznim kriterijumima. U ovom tekstu biće data njihova klasifikacija prema funkcijama. Svrha ove klasifikacije je grubi prikaz najvećeg dela onoga što se na tržištu podrazumeva pod

pojmom softverski proizvodi. Prema ovoj klasifikaciji, na prvom nivou razlikujemo:

- softverske proizvode opšte namene i
- softverske proizvode posebne namene.

Daljom podelom, kod *softverskih proizvoda opšte namene* razlikujemo sledeće grupe:

- operativne sisteme,
- sistemske programe i
- kontrolne i dijagnostičke programe.

Softverski proizvodi posebne namene dele se na:

- aplikativne programe i
- korisničke softverske proizvode.

Tabela 1.1.: Klasifikacija softverskih proizvoda prema funkcijama, [2,16]

Softverski proizvodi (SP)				
SP opšte namene		SP posebne namene		
Operativni sistemi	Sistemski programi	Kontrolni i dijagnostički programi	Aplikativni programi	Korisnički softverski proizvodi
-opšte namene -real-time -transakcioni -multi procesorski	-asembleri -revodioci -punjači -vezni -spuleri -translatori	- debageri - drajveri - pristupne rutine	- SRBP - CASE - GUI - Internet alati - OLAP	- CAD - CAP - CAM - CAQ - PPS - MIS

Operativni sistemi se mogu podelititi na više načina. Tako razlikujemo *operativne sisteme sa prividnom memorijom* (virtual

storage) i *operativne sisteme bez prividne memorije*, zatim *operativne sisteme vezane za određeni tip računara* i *relativno nezavisne operativne sisteme*. Može se izvršiti dalja podela na sledeće klase operativnih sistema: operativni sistemi opšte namene, operativni sistemi za rad u realnom vremenu, transakcioni operativni sistemi, multiprocesorski operativni sistemi i mrežni operativni sistemi.

Sistemski programi predstavljaju prvi nivo korisničkih alata, neophodnih za razvoj i korišćenje drugih softverskih proizvoda. Mogu se podeliti na sledeće klase: asembleri, softverski prevodioci, punjači, vezni programi, spuleri i translatori.

Kontrolni i dijagnostički programi predstavljaju grupu pomoćnih programa koja sadrži sledeće klase: debagere, drajvere i pristupne rutine.

Aplikativni programi predstavljaju skup programskih biblioteka, alata i razvojnih produkata koji se koriste pri razvoju softverskog proizvoda u njegovoj redovnoj produkciji kao pomoćno sredstvo. Njih čine sledeće grupe: komunikacijski programi, stono izdavaštvo, sistemi za rukovanje bazama podataka, generatori programa, CASE alati, hipertekst, sistemi veštačke inteligencije, sistemi za obradu teksta, sistemi spregnutih tabela, grafički interfejsi, translacijski softver, alati za testiranje, menadžerski softverski alati, alati za definisanje zahteva, internet serveri, internet čitači, multimedijalni sistemi, OLAP sistemi, integrisani softverski proizvodi i drugo.

Korisnički softverski proizvodi razvijaju se za tačno određenog korisnika ili za grupu određenih, odnosno potencijalnih korisnika. Ovim softverskim proizvodima automatizuju se procesi iz pojedinih funkcija sistema konkretnog korisnika, kao što su knjigovodstvo, finansiranje, skladišno poslovanje, proizvodnja, pružanje usluga, nabavka i prodaja, upravljanje ljudskim resursima i drugo.

Bitna strategija projektovanja korisničkih softverskih proizvoda naziva se CIM (Computer Integrated Manufacturing), koja se pojavljuje prvi put 1985. godine na sajmu informatike u Hanoveru. Predstavlja integralni pristup projektovanju softvera jedne organizacije, koji se sastoji iz sljdećih modula: CAD (Computer Aided Design), CAP (Computer Aided Planning), CAM (Computer Aided Manufacturing), CAQ (Computer Aided

Quality) i PPS (Pro-duction Planning System), dok se, kao prateći, skoro uvek poja-vljuje MIS (Management Information System), uz niz drugih opcionih modula.

1.1. Paradigme arhitekture softvera (Programming paradigms)

Arhitektura softvera se menja tokom vremena. Za promene postoji veći broj razloga, a osnovne treba tražiti među onima koji bitno utiču na troškove razvoja i eksploatacije sistema. Promene su evulucionog karaktera sa nekoliko karakterističnih paradigmi koje se međusobno nadovezuju. Analiza koja sledi može pomoći otkrivanju trenda na osnovu kojeg možemo sa više izvesnosti govoriti o budućoj evoluciji arhitekture softvera.

Ako se pođe od stava da osobine programske podrške objedinjuju arhitekturu sistema, onda se može konstatovati da je u drugoj polovini XX veka evolucioni put arhitektura određen sa svega nekoliko vladajućih paradigmi primene računara. Pojava novih paradigmi dovodila je do skoka ne samo u pogledu kvaliteta, nego i poboljšanja performansi, kako u fazi razvoja, tako i u fazi eksploatacije informacionih sistema. Imajući u vidu rezultate date u [4], u ovom poglavlju će se izložiti jedan od mogućih pristupa u prikazu osnovnih karakteristika evolucije informacionih sistema.

Monolitna programska podrška

Ova faza evolucije arhitekture se može locirati na period od 1948. do 1965.godine, a dominantne karakteristike informacionih sistema (IS) zasnovanog na primeni računara bile su pre svega određene tadašnjim računarskim hardverom koji je bio izuzetno skup. Cena personala koji je koristio računar po jedinici vremena u odnosu na ekvivalentne troškove hardvera, skoro da se mogla zanemariti. Softverska podrška, kao interfejs između korisnika i mašine, bila je pre svega prilagođena zahtevima hardvera. Korisnik je morao da "misli poput mašine". Bila je to faza monolitne programske podrške u kojoj je izvršni kod koji proizilazi iz algoritma rešenja problema "srastao" sa pogonskim rutinama ulazno-izlaznih uređaja i monitorskim programom.

Rešavanje nekog problema se tretiralo kao kreativan, unikatan rad programera. Podrška je pisana na mašinskom ili asemblerskom jeziku, sa osnovnim zadatkom da se program izvrši u što kraćem vremenu. Korisnik je bio osoba školovana za rad sa konkretnim sistemom. Imajući u vidu činjenicu da korisnik sa računarskim sistemom, namenjenim da vrši masovna izračunavanja, može da komunicira isključivo posredstvom korisničkog interfejsa, u prvoj fazi evolucije interfejs se dizajnirao u stilu "spartanskog" komfora. Cena korisničkog interfejsa, tipične električne pisaće mašine u funkciji konzole, bila je zanemarljiva u odnosu na cenu hardvera.

Dihotomija aplikacija- Sistemski softver

Proizvođači hardvera da bi pospešili prodaju svojih mašina počeli su potencijalnim korisnicima da nude ne samo hardver već i deo programske podrške koji se pokazao kao obavezan pri eksploataciji računarskog sistema. Ovaj deo gotove programske podrške poznat je pod imanom sistemski softver i lociran je kao interfejsni sloj između aplikacije koju razvija korisnik i hardvera za izračunavanje.

Sistemski softver se može nadalje podeliti na operativni sistem, bliži hardveru, i na pomoćne programe, deo bliži aplikaciji. Operativni sistem objedinjuje funkcije ulazno-izlaznih aktivnosti, mehanizme obrade hardverskih i softverskih prekida, mehanizme za planiranje, izvršavanje i praćenje izvršavanje zadataka.

Promena paradigme korišćenja računarskog sistema nije samo podigla produktivnost programera pri razvoju aplikacija, već je smanjila i fiksne troškove njihove obuke. Oni nisu više morali da budu upoznati sa detaljima rada i karakteristikama hardvera, već pre svega sa načinom rešavanja problema uz pomoć višeg programskog jezika. Programerima je bilo omogućeno da u realizaciji koriste ranije napisane i proverene delove programskog koda.

Jednom napisana aplikacija na nekom višem programskom jeziku mogla se prenositi i izvršavati na drugim računarskim siste-

mima istog ili različitih proizvođača. Portabilna aplikacija je time stekla svojstvo gotove robe.

Ako se ima u vidu da je većina gotovih aplikacija nastajala u okviru firmi koje su proizvodile pre svega hardver, odnosno sopstveni sistemski softver, onda je razumljivo da su takve aplikacije projektovane pre svega sa svojstvom tzv. vertikalne kompatibilnosti, tj. mogućnosti instalacije na sistemima istog proizvođača različite obradne snage.

Računari su se projektovali da omoguće simultani rad većeg broja korisnika, koji se dele u dve kategorije, kategoriju profesionalnih korisnika - programera i kategoriju pravih korisnika.

Za ovu fazu evolucije IS-a karakteristično je i poboljšanje performansi harverskog korisničkog interfejsa. Vizuelno alfanumeričko komuniciranje posredstvom ekranskog terminala postaje opšte prihvaćen standard. Ova faza se vezuje za period od 1965. do 1985.godine.

Paradigma korisniku bliskog sistema

Osamdesetih godina prošlog veka, odigrao se veliki skok u razvoju IS, pojavom jeftinog i svima dostupnog personalnog računara (PC). Pojava personalnog računara dovela je do nove paradigme u korišćenju računarskih sistema.

Tipičan korisnik PC-ja je vrlo malo upoznat sa tehničkim karakteristikama i mogućnostima računara. On kupuje računar da bi rešio probleme u svom okruženju i podigao sopstvenu produktivnost. Zapravo korisnik kupuje rešenje svog problema, a rešenje čini mikroračunar sa skupom gotovih aplikacija. Sada se veći deo hardvera i softvera javlja u ulozi "prilagodnog" sloja, interfejsa između korisnika i aplikacije. Kako je vizuelno simboličko komuniciranje zasnovano na korišćenju "ikona", jasna je važnost grafičkog korisničkog interfejsa, i to ne samo u hardverskom delu sa ekranom u boji visoke rezolucije i odgovarajućom grafičkom karticom, već i softverskim grafičkim interfejsom kakav je, na primer, WINDOWS.

PC omogućava multimedijsku prezentaciju, a skup gotovih programa omogućava potpunu automatizaciju kancelarijskog poslovanja. Kao posledica pojave PC-ja može se navesti raspad tradicionalno jedinstvenih proizvođača hardvera, operativnih sistema i velikih programskih paketa. U ovoj fazi razvoja IS na tržištu se jasno razlikuju proizvođači standardnih hardverskih komponenti, proizvođači sistemskog softvera i specijalizovane kuće za određenu kategoriju aplikacija.

Računarska mreža

Iako nije zamišljen kao sredstvo za realizaciju složenih IS, PC kao radna stanica u lokalnoj računarskoj mreži je ubrzo nakon pojave postao moćno sredstvo za realizaciju IS firmi srednje veličine. Razmena podataka sa drugim entitetima u okruženju se obavlja pomoću telekomunikacione računarske mreže, koja može biti sastavljena od računara sa različitim hardverom i operativnim sistemima. Problem konektivnosti, odnosno međusobnog povezivanja heterogenih računara, rešen je sedamdesetih godina razvojem posebnih komunikacionih protokola, od kojih je najpoznatiji TCP/IP. Aplikacija je ponovo podeljena na specijalizovani sistemski deo - ljusku distribuiranog operativnog sistema i pravu aplikaciju. Danas najveća svetska mreža INTERNET koristi TCP/IP protokol. Računarske mreže poput Interneta su omogućile realizaciju globalnih IS, u kojem svaka informacija u sistemu, bez obzira na njen trenutan položaj, postaje potencijalno dostupna svim korisnicima sistema. Usavršena je tehnologija za efikasno skladištenje, organizovanje i međusobno povezivanje različitih tipova informacija u računarskim mrežama, bilo da su tekstualne, numeričke, tabelarne, grafičke, slikovne, zvučne ili video zapisi.Takve pogodnosti pruža World Wide Web (WWW). Korisnik mreže ima mogućnost da kompletnu svetsku mrežu vidi kao jedinstven gigantski disk sa svim raspoloživim multimedijskim informacijama.

Globalni IS

Sadašnje stanje u oblasti komercijalnih tehnologija za međusobno povezivanje računarskih sistema omogućava kreiranje efikasne infrastrukture za trenutnu isporuku uskladištenih informacija na proizvoljno mesto. Stvoreni su preduslovi za prevazilaženje

ograničenja koja nameću heterogeni distribuirani računarski sistemi u odnosu na koncept dinamičke raspodele procesorskih opterećenja. U osnovi je jednostavna ideja: "Ako se već postojeći mehanizam na kojem leži WWW koristi za prenos informacija, zašto se na isti način ne bi preneli i programi koji rade sa tim informacijama." Dakle, isti programi za sve računare u mreži. To je relativno jednostavno u slučaju homogene mreže, ali to je moguće i u heterogenoj mreži, ukoliko se svi programi pišu za unapred dogovorenu apstraktnu mašinu, a potom se za računare različitih proizvođača napiše kratak program - simulator, orijentisan na rad sa WWW serverom. Takve apstraktne mašine su raspoložive od 1995.godine. Ova softverska paradigma je evolutivna, a u osnovi nasleđuje svojstva pethodnih paradigmi.

Tabela 1.2: Pregled paradigmi arhitekture softvera i njihovih glavnih
osobina, [2,3]:

Naziv paradigme	*Glavne osobine*
faza monolitne programske podrške	• proces obrade vezan za uređaje, • programi u mašinskom jeziku ih asembleru, • interfejs nekomforan i malih mogućnosti
dihotomija aplikacija - sistemski softver	• pojava multiprogramskog rada, • mogućnost prenosa programa, • interaktivno komuniciranje i razmena podataka pomoću telekomunikacija
okruženje blisko korisniku	• kupovina gotovih rešenja, • pojava grafičkog korisničkog interfejsa, • aplikacije kao konfekcijski proizvodi
računarske mreže	• povezivanje računara sa različitim hardverom i operativnim sistemima, • korišćenje Interneta, • globalizacija informacijskih sistema
globalni multimedijalni informacijski sistem	• upotreba različitih tipova informacija, • interaktivna isporuka informacija i pratećih programa, • jedinstveno praćenje informacija iz svetskog okruženja

Literatura:

1. Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin, 2001.
2. B.Jošanov,P.Tumbas, *Softverski inženjering*, VPŠ, Novi Sad, 2002.
3. D.Starčević, *Evolucija arhitekture softvera*, INFOTEH 96,Donji Milanovac,str.34-40
4. D.Verne Morland, *The Evolution of Goftware Architecture*, Datamation, February 1.,1985.
5. Brad J. Cox, *Object-Oriented Programming - An Evolutionary Approach*, Addison-Wesley,1987.

2. ALATI ZA RAZVOJ SOFTVERA (Software development tools)

Danas se na tržištu može naći veliki broj alata za razvoj softvera. Među njima se mogu naći i alati specijalizirani za izvršavanje samo jedne funkcije kao i kompletni paketi razvojnih alata namenjenih svim životnim ciklusima softvera, tzv. IDE (*integrated development environment*). Iako je broj alata i priručnika za njihovu upotrebu vrlo velik, mali je broj članaka o softverskim alatima napisan sa nekog opšteg tehničkog aspekta. Dodatni problem su česte i brojne promene pojedinih alata. Iako svaki od alata ima svoja ograničenja, dobar IDE ili neki drugi razvojni alat, upotrebljen na odgovarajući način, značajno će doprineti uspehu projekta.

Alati za razvoj softvera su alati čija je svrha podrška procesima životnog ciklusa softvera. Alati omogućavaju automatizovanje repetitivnih, precizno definisanih aktivnosti, čime se reduciraju napori softver inženjera, te im se time pruža mogućnost da se skoncentrišu na kreativne aspekte projekta. Alati su često dizajnirani kao podrška određenim softver inženjering metodama čime se značajno reduciraju administrativni poslovi. Svrha alata, kao i metoda uostalom, je sistematizirati pristup softver inženjeringu.

SWEBOK[1] deli softver inženjering alate u deset kategorija:

1. Software Requirements Tools,
2. Software Design Tools,
3. Software Construction Tools,
4. Software Testing Tools,
5. Software Maintenance Tools,
6. Software Configuration Management Tools,
7. Software Engineering Management Tools,
8. Software Engineering Process Tools,
9. Software Quality Tools,

[1] *Guide to the Software Engineering Body of Knowledge; standard IEEE Computer Society-a, objavljen u februaru 2004. godine*

10. Miscellaneous Tools.

1. Software Requirements Tools
Ovi alati namenjeni su specificiranju softverskih zahteva, a dele se u dve kategorije: alate za modeliranje i *traceability* alate.

1.1.Alati za modeliranje zahteva. Ovi alati namenjeni su otkrivanju, analiziranju, specificiranju i validaciji softverskih zahteva. Primeri alata za modeliranje zahteva (pri čemu neki od alata namenjeni su samo za modeliranje zahteva, ali ih se većina može koristiti i za druge aktivnosti u svim fazama životnog ciklusa softvera):

- AnalystPro od Goda Software
- Caliber RM™ od Starbase®
- C.A.R.E. od Sophist Group
- IRqA od TCP Sistemas e Ingeniería
- OnYourMark Pro od Omni Vista
- Requirements Design & Traceability (RDT©) od IGATECH Systems Pty Ltd.
- RequireIT™ od Telelogic
- Requisite Pro® od Rational®
- ScenarioPlus for Use Cases
- Slate Require™ od EDS
- Vital-Link od Compliance Automation Inc.
- Volere od The Atlantic Systems Guild
- Cross-Tie Requirements Tracer Software (XTie-RT®) od Teledyne Brown Engineering (TBE)

1.2. Traceability alati. Važnost ovih alata se neprestano povećava s kompleksnošću softvera. Budući da su ovi alati relevantni i za ostale procese životnog ciklusa, često se u klasifikacijama definišu kao zasebna kategorija. Traceability alati omogućavaju inženjerima da povežu zahteve s njihovim izvorima, s promenama zahteva kao i modeliranje elemenata koji će zadovoljiti zahtieve.Ovi alati omogućavaju praćenje zahteva kroz niz dokumenata koji su nastali kao popratna dokumentacija u toku razvoja sistema.

Primeri *traceabilitiy* alata:

- AnalystStudio; proizvođač:Rational Software; Opis: Tool Suite. uključuje RequisitePro, Rose, SoDA and ClearCase
- Caliber-RM; proizvođač: Technology Builders, Inc (TBI); Opis: Requirements traceability tool; Opis: CASE alat namenjen inženjeringu celog životnog ciklusa softvera (requirements management, behavior modeling, system design,verification);
- CORE; proizvođač: Vitech Corporation; Opis: omogućava analizu zahteva, behavorial analiza, definisanje arhitekture i verifikacija, validacija dizajna, business process modeling project.
- DOORSrequirelT; proizvođač: Telelogic; Opis: Requirements trace alat integrisan s Microsoft Wordom.

2. *Software Design Tools*

Alati ove skupine omogućavaju kreiranje i proveru dizajna softvera. Kao posledica velikog broja različitih metoda i notacija u dizajniranju softvera postoji i mnoštvo različitih alata, ali niti jedna opšte prihvaćena podela.

2.1.Jedan od danas najčešće korištenih jezika za modeliranje je UML *(Unified Modeling Language)*. UML je industrijski, standardni jezik za specificiranje, vizualizaciju, konstrukciju i dokumentiranje softverskog sistema. Neophodno je imati model softverskog sistema pre nego se uopšte počne pisati kod. Dobro opisan model olakšava komunikaciju među članovima projektnog tima kao i dekomopoziciju kompleksnog softverskog sistema na manje i jednostavnije zadatke. UML se sastoji od:

- elemenata modela - osnovni koncepti i semantika,
- notacije - vizualizacija elemenata modela,
- smernica - načini upotrebe.

UML je i programski jezik i razvojni alat. Mnogi proizvođači (npr. Rational Software -Rational Rose) nude UML alate. UML uključuje skup razvojnih koncepata visokog nivoa kako što su *collaboration, framework i pattern*. Iako je UML dizajniran za objektno-

orjenitisani pristup, moguće ga je koristiti i za modele koji nisu objektno orjentisani.
UML-om su definisani sledeći dijagrami:

- *use case* - dijagram slučajeva upotrebe,
- *class* - dijagram klasa,
- *statechart* - dijagram stanja,
- *activity* - dijagram aktivnosti,
- *sequence* - sekvnecijalni dijagram,
- *collaboration* - kolaboracijski dijagram,
- *component* - dijagram komponenti.

Pomenuti dijagrami omogućavaju veći broj različitih perspektiva na probleme analize i razvoja softvera.

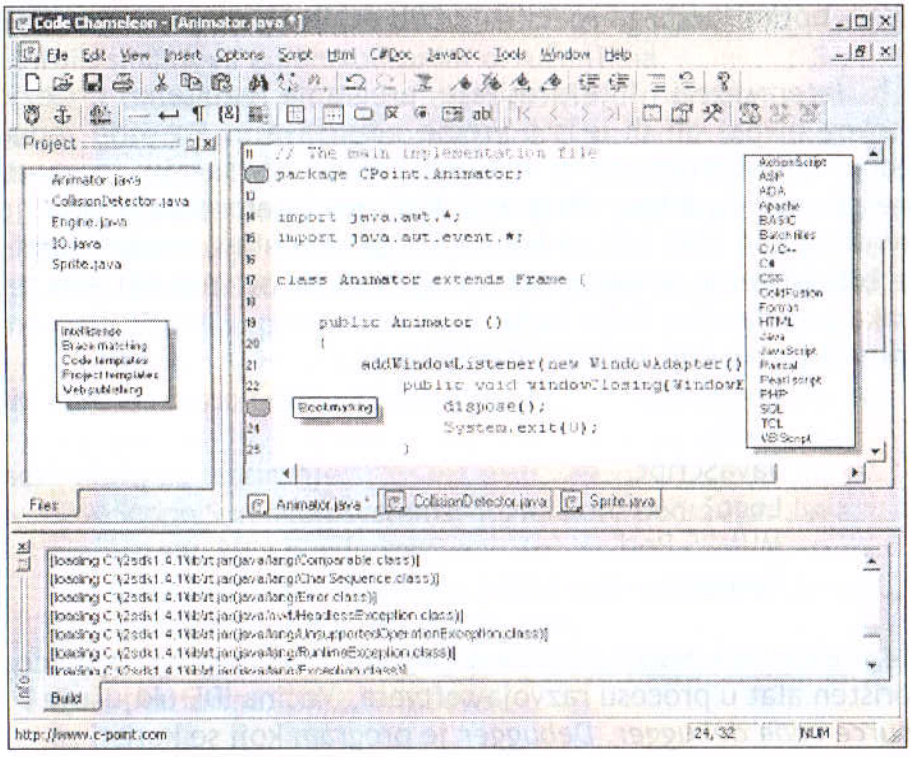
2.2.Alati za testiranje dizajna
Ova grupa alata omogućava softver inženjerima da odluče koje je testove neophodno uraditi, te da generišu testne podatke.

- ALLPAIRS; Test Case Generation Tool; konstruiše mali set testnih slučajeva koji uključuju uparivanje svake vrednosti sa setom parametara;
- AllPairs.java; All-pairs test case generator;
- Assertion Definition Language (ADL); Automatic test generation tool;Assertion;
- DARTT Automated testing over subprogram parameter range; alat za verifikaciju softvera i kvalitativnu analizu koda;
- Datagen2000; Test Data Generation Tool; ovaj alat generirše testne podatke za Oracle baze podataka;
- Datatect Test Data; Generator Test data;
- McCabe Test; Test Design Tool ; vizuelno okruženje za planiranje softverskih testnih resursa;
- Orchid; test case design tool ; omogućava dizajniranje efikasnih testnih slučajeva;
- Panorama C/C++; alat za planiranje i testiranje dizajna.

3. Software Construction Tools
Ovi alati se koriste za kreiranje i prevođenje programske reprezentacije-*source* koda u detaljnu i eksplicitnu formu koju je moguće izvršiti. *Software construction* alate možemo podeliti na:

3.1. Program editori. Ovi alati se koriste za kreiranje i modificiranje programa i dokumenata koji su povezani s tim programima. Danas su editori obično deo IDE.
Primeri program editora:

- CriSP; proizvođač: Vital, Inc ;Opis: file editor za Unix i Windows okruženja;
- EditPad Lite; proizvođač: JGsoft - Just Great Software; Opis: basic text editor, zamena za Notepad;
- VbsEdit;proizvođač: ADERSOFT; Opis: služi za editiranje VBS datoteka;
- Code Chameleon, proizviđač: C Point;
- Shusheng SQL Tool; Proizvođač: CELC ; Opis:web-based SQL programski alat;
- HexCmp; proizvođač: Fairdell Software; Opis: convenient visual binary file comparison application.



Slika 2. 1: Izgled program editora Cameleon

3.2 *Kompajleri i generatori koda.* Tradicionalno shvatanje kompajlera podrazumeva *non-interactive* prevodioce *source* koda, ali danas postoji trend da se kompajleri i tekst editori integrišu u jedinstveno programsko okruženje (IDE). U ovu grupu alata spadaju i pre-procesori, linker/loader i generatori koda. Jedna od ključnih karakteristika kompajlera je brzina. Dobar kompajler omogućava tri nivoa kompajliranja. Osnovni mod ne nudi informacije za ispravljanje grešaka (debug information). Prilikom kompajliranja velikih programa u procesu razvoja, ova opcije može bit vrlo korisna jer je najbrža. No, ukoliko želite dodatne informacije o greškama biće potrebno više vremena za kompajliranje. Najsporije kompajliranje ćete dobiti u slučaju kada isključite opciju za optimizaciju. Nivo optimizacije se razlikuje od kompajlera do kompajlera i u većini slučajeva biste trebali proveriti što vaš kompajler nudi:

- nekoliko dodatnih optimizacija opšte namene,
- optimizaciju za specifični CPU instrukcijski set,
- optimizaciju za specifičnu CPU arhitekturu.

3.3. Interpreteri. Interpretiraju programe napisane u jezicima visokog nivoa, ali ih u isto vreme i izvršavaju. Prevode jednu po jednu liniju programa u mašinski jezik, izvršavaju je, a zatim prelaze na narednu. Programi koji se interpretiraju sporije se izvršavaju od onih koji se kompajliraju. Međutim, ovakakv alat može biti izuzetno koristan jer omogućava da se interaktivno testira svaka pojedinačna linija koda.
Primeri jezika koji koriste interpretere:

- Euphoria,
- Forth,
- JavaScript,
- Logo,
- MUMPS Perl,
- Python.

3.4. Debuggeri. Nakon editora *debuggeri* su verovatno najčešće korišten alat u procesu razvoja softvera. Većina IDE uključuje i *source code debugger. Debugger* je program koji se koristi za ispravljanje grešaka u drugim programima.
Primeri *debuggera*:

- GNU Debugger (GDB) radi pod Unix sistemima i na mnogim programskim jezicima uključujući i C, C++ i FORTRAN;

- CodeView i Visual Studio Debugger; Microsoft debugger za programe pisane u Microsoft C i CodeView;
- T-Bug, the integrated debugger pod Perl 5;
- Broadway;
- DAEDALUS
- Java Platform Debugger Architecture;
- Adb;
- Sdb;
- Dbx;
- Dynamic debugging technique (DDT);
- Purify;
- Ladebug.

4. Software Testing Tools

U ovu kategoriju alata spadaju:

4.1. Test generatori. Ovi alati omogućavaju razvoj testnih slučajeva.

4.2 Test execution frameworks. Ovi alati omogućavaju izvršavanje testnih slučajeva u kontrolisanim okruženjima.

4.3. Test evaluation alati. Ovi alati olakšavaju procenu rezultata testa, određujući da li je posmatrano ponašanje jednako očekivanom.

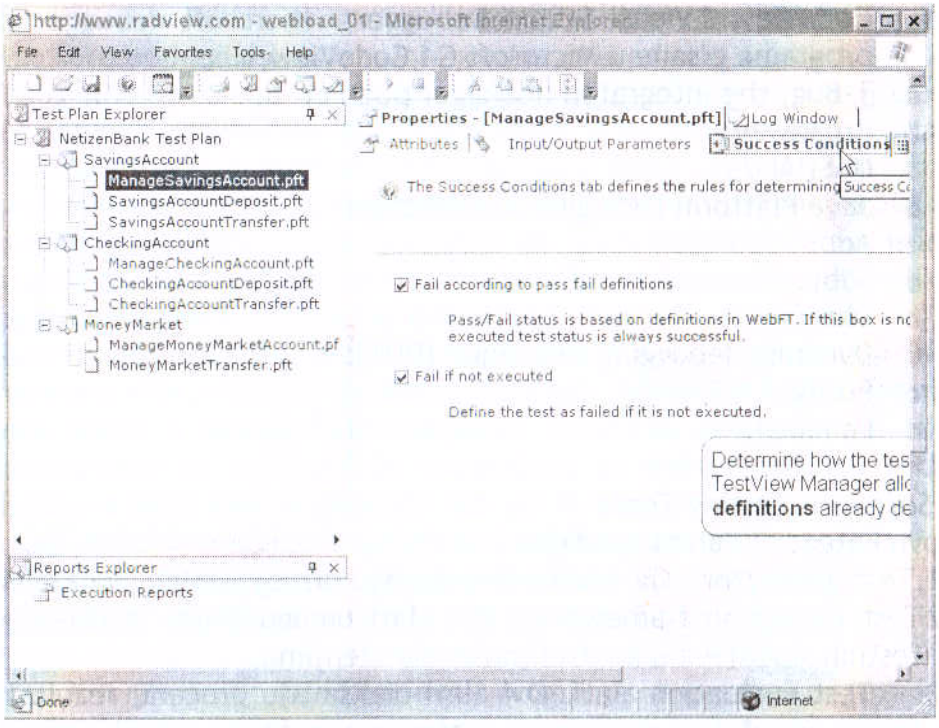
4.4. Test management alati. Ovi alati nude podršku za sve apkete procesa testiranja softvera.

4.5. Alati za analizu performansi. Ova grupa alata se koristi za merenja i analizu performansi softvera.

Testni alati su često i sastavni dio IDE.

Primeri alata za testiranje:

- JavaScope Sun Microsystems; jezici koje podržava: Java; platforma: Java platform;
- Pegasus Ganymede Software; jezici koje podržava: Java, C, C++; platforma: Unix, Windows, Mac;
- WebLoad Radview Software; jezici koje podržava: Java, C, C++; platforma: Unix, Windows.



Slika 2. 2: Izgled alata za testiranje, WebLoad Radview Software

5. Software Maintenance Tools

Ova kategorija obuhvaza alate neophodne za održavanje postojećeg softvera. Razlikujemo dve kateorije:

5.1. *Comprehension* alati. Ovi alata olakšavaju ljudima razumevanje programa. Obično uključuju alate za vizuelizaciju.

5.2. Reinženjering alati. Reinženjering se definiše kao preispitivanje i preoblikovanje softvera u novu formu kao i implementacija tog novog oblika softvera. Reinženjering alati podržavaju takve aktivnosti. Postoje i tzv. *reverse engineering* alati koji podržavaju proces radeći obrnutim redosledom - od postojećeg produkta kreiraju specifikaciju i opis dizajna koji se onda može upotrebiti u transformaciji i generisanju novog produkta iz postojećeg..

Primeri *maintenance* alata:

- BPR Toolkit; Business Process Reengineering Toolkit;
- Ensemble ; Automated maintenance and testing for existing C code;

- 4D ; C/C++ Reverse Engineering and Documentation Tool;
- Vantage Team; Integrated development environment for client-server maintenace and development.

6. *Software Configuration Management Tools*
Ovaj skup alata se može podeliti u tri kategorije:

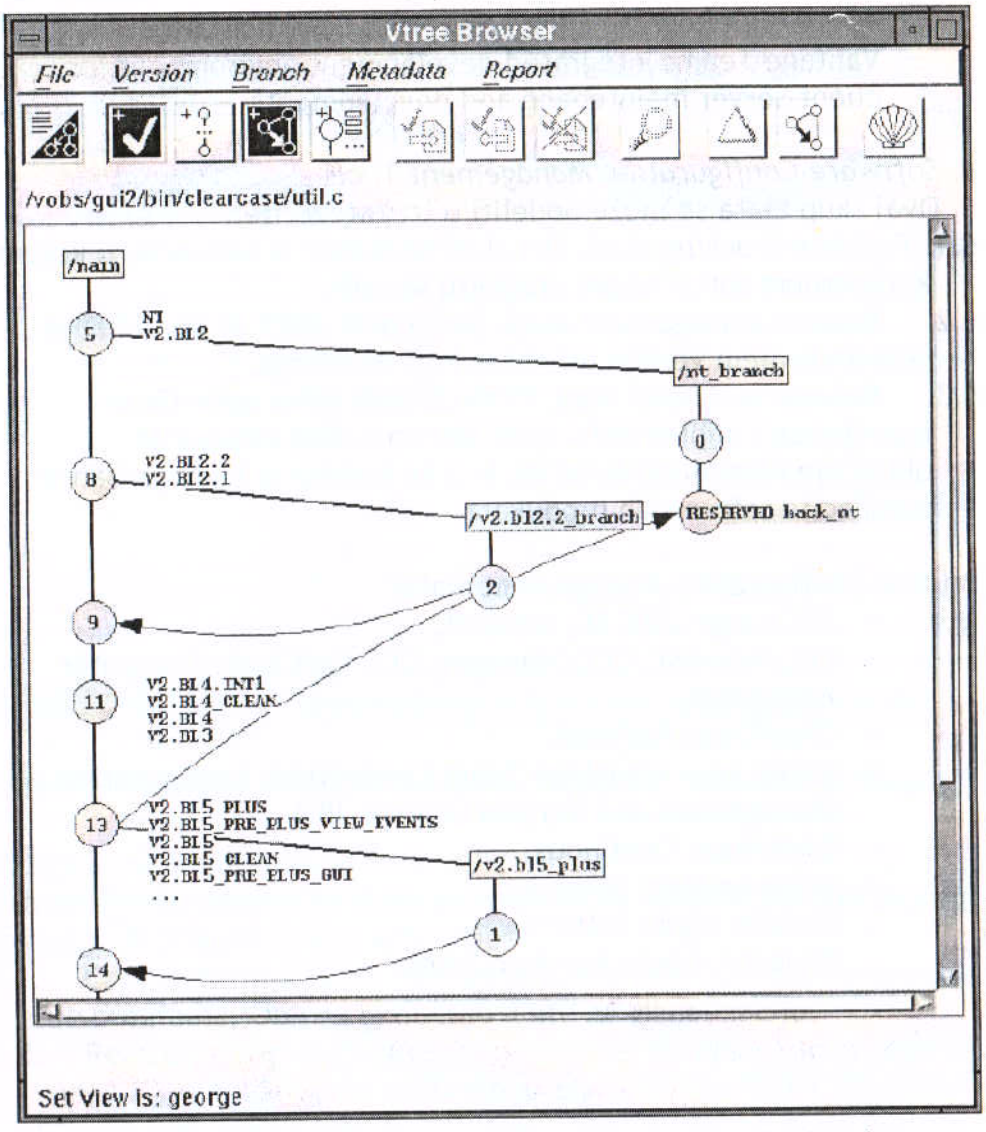
6.1. *Problem-tracking* alati. Ovi alati se koriste u zavisnosti o kojem konkretnom softverskom produktu se radi.

6.2. *Version management* alati. Svrha ovih alata je upravljanje velikim brojem verzija produkta koji se razvija.

6.3. *Release and build* alati. Pomoću ovih alata upravlja se izgradnjom i implementacijom softvera. Ova kategorija uključuje i instalacijske alate koji se koriste za konfigurisanje instalacije softverskih produkata.

Primeri Configuration Management alata:

- AllChange 2000 SE; IntaSoft,
- CCC/Harvest, CCC/Manager, CCC QuikTrak; Computer Associates,
- ClearCase; Rational,
- CMVC, now VisualAge Team Connection; Configuration Management and Version Control, IBM,
- Continuus; Continuus,
- eChange Man; Serena,
- Enabler aqua; Softlab,
- Endevor; Computer Associates.



Slika 2.3: Izgled alata ClearCase kompanije Rational

7. Software Engineering Management Tools
Engineering management alati se dele na tri kategorije:
7.1. Alati za planiranje i prećenje projekta (*Project planning and tracking tools*). Ovi alati se koriste za procenu i merenja troškova i pravljenje rasporeda toka projekta.
7.2. *Risk management* alati. Ovi alati se koriste za identifikaciju, procenu i nadgledanje rizika.

7.3. Alati za merenja. Ovaj skup alata omogućava sprovođenje merenja softverskih produkata.
Primeri *engeneering managemnet* alata:

- DecisionPro; Vanguard Software Corp,
- PRA - Project Risk and Contingency Analysis;Katmar Software,
- Software Risk Evaluation; Software Engineering Institute,
- C-Cover: test coverage, measurement,
- ActionPlan; Netmosphere Inc.,
- AIO WIN; Knowledge Based Systems, Inc.

8. Software Engineering Process Tools
Softver *engineering* proces alati se dele na:
8.1. Alate za modeliranje procesa (Process modeling tools). Ovi alati se koriste za modeliranje i istraživanje softver inžinjering procesa.
8.2. Alate za upravljanje procesima (Process management tools). Zadatak ovih alata je osigurati podršku *management*-u softver inženjeringa.
8.3. Integrisana CASE okruženja. Ovaj skup alata je podrška za više faza životnog ciklusa softvera.
8.4. Process-centered software engineering environments. Ova okruženja uključuju informacije o procesima životnog ciklusa softvera, te vode i omogućavaju nadzor nad definisanim procesima.

Primeri *Software Engineering Process* alata:

- Computer Associates BPWin,
- First Place Group Proprietary Tools,
- Microsoft Visio.

9. Software Quality Tools
Quality alati se mogu podeliti u dve kategorije:
9.1. *Review and audit tools*. Glavni zadatak ovih alata je nadzor, (*These tools are used to support reviews and audits*).
9.2. *Static analysis tools*. Ovi alati se koriste za analizu softvera, a uključuju sintaksne i semantičke analizatore, analizatore zavisnosti kao i kontrolu toka podataka.

Primeri *quality* alata:

- AceProject, web-bazirani projekt *management* softver i *bug tracking* alat,
- AdventNet QEngine; *comprehensive cross-platform tool with a compelling business value proposition for automating large-scale java and J2EE application testing,*
- Aimware; pomoć pri test *management*-u.

10. Miscellaneous tools (Ostali alati)
Ovu kategoriju sačinjavaju alati koji na neki način podržavaju razvoj softvera, ali se s obzirom na navedenu podelu ne uklapaju niti u jednu kategoriju. Kategoriju alata ostali možemo podeliti na:

10.1. Integrisani alati (IDE - interactive development environment)
Integracija alata u jedinstveno okruženje je izuzetno važna zbog omogućavanja individualnim alatima da rade zajedno. Ova kategorija alata se preklapa sa integrisanim CASE okruženjima, međutim ova vrsta okruženja danas sve više dobija na značaju, te je stoga potrebno posebno da se naglasi. Kvalitetan IDE sadrži širok raspon alata kao što su editori, debugger i alati za testiranje koji podržavaju razvoj softvera. IDE omogućava integraciju sastavnih komponenti tako da olakšava programeru rad u *edit-compile-debug* ciklusu.

Primeri IDE:

- Bean Machine IBM Java; Windows, OS2, Unix; Builder Xcessory Pro Integrated Computer Solutions Java, C, C++ Unix, Windows
- CodeWarrior Professional Metrowerks; Java, C, C++, Pascal;Unix, Windows, Mac
- Java Workshop Sun Microsystems Java; Solaris, Windows
- JBuilder Imprise; Java; Windows, AS400
- SuperCede for Java Supercede;Java; Windows
- UIM/X VisualEdge Software; Java, C, C++; Unix, Windows, Mac
- Visual Cafe for JavaSymantec; Java; Windows
- VisualAge IBM; Java; Unix, Windows
- Visual J++ Microsoft; Java; Windows

10.2. Meta alati. Meta-alati generišu druge alate;
10.3. Evaluacijski alati.

Još neki primeri alata kategorije *ostali*:

- Aivosto VB Watch; VB coverage, performance, debug and error detection tool.
- AQtime; Integrated profiling and leak-detection tool
- ASSIST (Asynchronous/Synchronous Software Inspection Support Tool); Enforcement and support of the inspection process
- Camtasia Video Screen Recorder Software
- Phoenix ImageCast MFG Disk imaging tool
- IPCheck Server Monitor Uptime/Downtime Monitoring Tool
- Norton Ghost Disk imaging tool
- NULLSTONE Automated Compiler Performance Analysis Tool Automated Compiler Performance Analysis Tool
- Q-CHESS Quality managament, web-based checklist support system
- Rational Quantify Performance Profiling Tool
- ReviewPro ReviewPro is a proven web-based, collaborative Technical Review and Inspections solution.
- SSW SQL Total Compare Database Tool
- Virtual PC Virtualization software
- VMware Virtualization software

Literatura:

1. Marc Hamilton; *Software Development: Building Reliable Systems*, Publisher : Prentice Hall PTR; Pub Date : March 22, 1999; ISBN : 0-13-081246-3
2. http://www.rational.com/products/rose/features.html.
3. http://www.ganymedesoftware.com.
4. http://www.radview.com.
5. http://sun.com.
6. http://www.computer.org/certification/Swebok_2004.pdf
7. http://www.site.uottawa.ca:4321/oose/index.html#softwarecrisis
8. http://epic.onion.it/workshops/w09/slides01/
9. http://www.itmweb.com/essay544.htm#f5

3. DIZAJN SOFTVERSKOG PROIZVODA (Software products design)

Polazeći od izvršene analize sistema, faza projektovanja, tj. dizajna treba da predloži kako realizovati softverski proizvod. Osnovni cilj ove faze je kreiranje jednostavnih, jasnih i preciznih specifikacija za efektivnu i efikasnu izradu i implementaciju softverskog proizvoda.

Procedura projektovanja može da teče na sledeći način:

1. Na osnovu identifikovanih i specifikovanih informacionih zahteva formulišu se neophodni ulazi koje treba obezbediti da bi softverski proizvod mogao formirati tražene izlaze.
2. Vrši se dekompozicija funkcija i pri tome se dekomponuju i predviđeni ulazi i izlazi.
3. Formira se predlog organizacije podataka.
4. Definišu se funkcije sa kojima će se obezbediti unos ulaznih podataka i realizovati informacioni zahtevi, u skladu sa predviđenom organizacijom podataka.
5. Daju se detaljni opisi funkcija iz br.4, sa dokumentovanim ulazima, izlazima, rečnikom i organizacijom podataka, koji će poslužiti kao baza za specificiranje programskih zahteva, na osnovu kojih se sprovodi kodiranje.

Uloga projektovanja je da navede alternative rešenja datog problema. Ovaj proces predstavlja alternative rešenja, modelira ih i razvija u skladu sa specificiranim zahtevima.

Proces projektovanja ima za cilj da definiše kako izgraditi sistem da bi se ponašao na način opisan zahtevima. U ovom procesu se izrađuje niz dokumenata koji treba da obezbede ulaze u proces implementacije. Kao forma procesa rešavanja problema, ovaj proces uključuje aspekt fizičkog i logičkog projektovanja.

Dva osnovna pristupa u današnjoj praksi su:

1. strukturni dizajn i
2. objektno-orijentisani (OO) dizajn.

Osnovne karakteristike strukturnog dizajna su:

- modeluje rešenja problema, a ne sam problem,
- rešenje se hijerarhijski razlaža na jednostavnije funkcionalne celine,
- problemi se rešavaju u određenim algoritamskim koracima, na višem ili nižem nivou hijerarhije,
- kada je potrebno izvršiti promene u programu, najčešće je potrebno menjati i algoritme,
- nakon dodavanja i izmena u programu, potrebno je ponovo proveriti širi kontekst projektovanog rešenja.

OO dizajn karakteriše da se:

- ovim pristupom modeluju problemi, a ne rešenja,
- problemi se razlažu na objekte, za koje se određuje šta, a ne kako rade,
- objekti se u dizajnu sistema tretiraju kao crne kutije,
- nad objektima se izvršavaju spoljne akcije,
- izmene i dodavanja vrše se najčešće u određenom objektu,
- dodavanje novih objekata je fleksibilno,
- postoji mogućnost ponovnog korišćenja komponenti ili njihovih delova.

U dijzajnu softverskih proizvoda se koristimo modeliranjem i nastojimo iskoristiti važnost modeliranja za ciljeve koji želimo postići.

3.1. Modeliranje

Modeliranje je centralni deo svih aktivnosti koje vode do generisanja dobrog software-a. Jedna firma za proizvodnju softvera je uspešna u meri koliko proizvodi na konzistentan način kvalitetan softver koji zadovoljava potrebe korisnika.

Za proizvodnju kvalitetnog softvera je potrebno:

-povezati i uključiti sve korisnike u cilju prikupljanja svih realnih zahteva sistema,

- kreirati čvrstu arhitektonsku bazu koja će takođe biti fleksibilna na promene,

-koristiti čvrste razvojne procese koji su prilagodivi mogućim promenama problema koji rešavamo.

Ukoliko proizvodimo softver malih dimenzija, pogrešimo, a rezultati ovog softvera nemaju posledice za okruženje u kojem će delovati, ni po podatke kojima će manipulisati, ne moramo ulagati puno truda u planiranje i modeliranje.

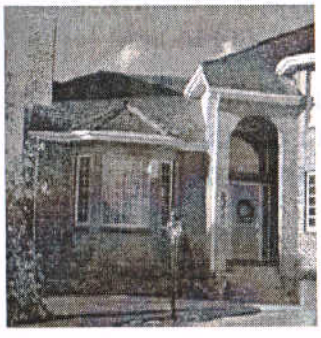
Ukoliko proizvodimo softver velikih dimenzija i čija efikasnost i rezultati u velikoj meri utiču na okruženje i najmanja greška može imati katastrofalne posledice. Zbog toga je potrebno veće projekte pažljivo planirati i uložiti trud u modeliranje.

Dobar primer je poređenje gradnje kućice za psa, kuće za jednu obitelj i poslovne zgrade.

Kuća za psa:

- Nije nam potrebno puno alata.
- Nije potrebno puno planirati.
- Ukoliko mu se ne svidi, nije problematično!

Kuća za jednu porodicu:



- Potrebno je planirati.
- Verovatno je potrebna ekipa (tim).
- Različiti planovi (spratovi, vodovod, ...).
- Poodica je zahtevna.

Poslovna zgrada:

- Biće potrebno planirati više mjeseci.
- Cena neuspeha je jako visoka!

Ali često se dogodi da:

- mnogi projekti imaju ambicije konstruisati neboder, ali prilaz problemu im je kao da konstruišu kućicu za psa,
- neki opet krenu sa ciljem da konstruišu kućicu za psa koja vremenom dostigne veličinu nebodera,
- dolazi se do momenta kad se kućica sruši na psa!

Model

Model je pojednostavljena realnost. Pruža nam planove sistema kao što su: opšti planovi (globalna vizija) i detaljni planovi pojedinih delova. Kreiramo modele da bismo bolje razumeli sistem koji želimo konstruisati. Modeli nam pomažu da vizueliziramo kakav je sistem ili kakav želimo da bude, daju nam šablon koji nas vodi u konstrukciji jednog sistema i dokumentuju odluke koje smo doneli. Modelom takođe specificiramo strukturu i ponašanje sistema.

Postoji ljudski limit da razume kompleksnost. Putem modeliranja reduciramo problem koji se izučava i centriramo se u svakom momentu samo na jedan aspekt. Modeliranjem se potencira ljudski um: model koji je adekvatno izabran dozvoljava da radimo na većem nivou apstrakcije.

Principi modeliranja

Upotreba modeliranja ima dugu istoriju u svim inženjerskim disciplinama. Ova istorija sugeriše četri osnovna principa:

1. Izbor koji model kreirati ima veliki uticaj u pristupu problemu i kako se daje forma rešenju.

2. Svi modeli mogu biti izraženi sa različitim nivom preciznosti (detaljnosti).
3. Najbolji modeli su usko vezani za realnost. Svi modeli su pojednostavljena realnost. Umetnost je u tome da pojednostavljenja koja su napravljena ne prikrivaju ni jedan važan detalj iz realnosti.
4. Jedan model nije dovoljan. Bilo kojem netrivijalnom sistemu se prilazi na bolji način putem manjih skupova modela, skoro nezavisnih.

Zašto orijentisanost ka objektima u modeliranju?

Orjentisanost ka objektima je u širokoj primeni zbog sličnosti koncepata modeliranja u odnosu na realne entitete, boljeg prikupljanja i validiranja zahteva i približavanja problema njegovom rešenju.

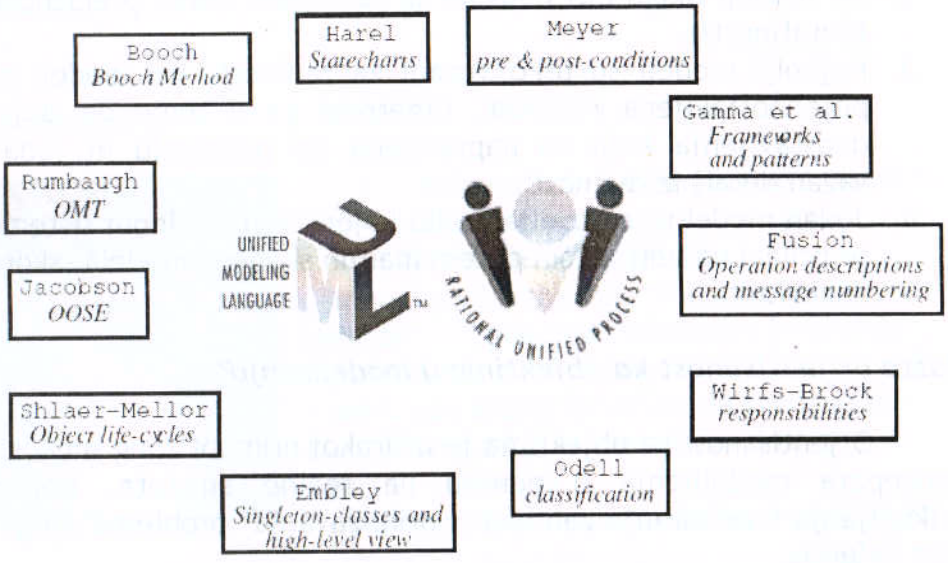
Zajednički koncepti modeliranja tokom analize, dizajna i implementacije: olakšavaju prelazak na sledeću fazu, poboljšavaju interaktivni razvoj i postavljaju granicu između «šta» i «kako».

3.2. UML - Unifed Modelig Language

Jedna od najpoznatijih OO metodologija jeste UML.

Prvi OO programski jezici su se pojavili 60-tih godina:
- SIMULA (1965. godine i zatim 1967. godine),
- SMALLTALK (70-tih i u potpunosti 80-tih XX veka).

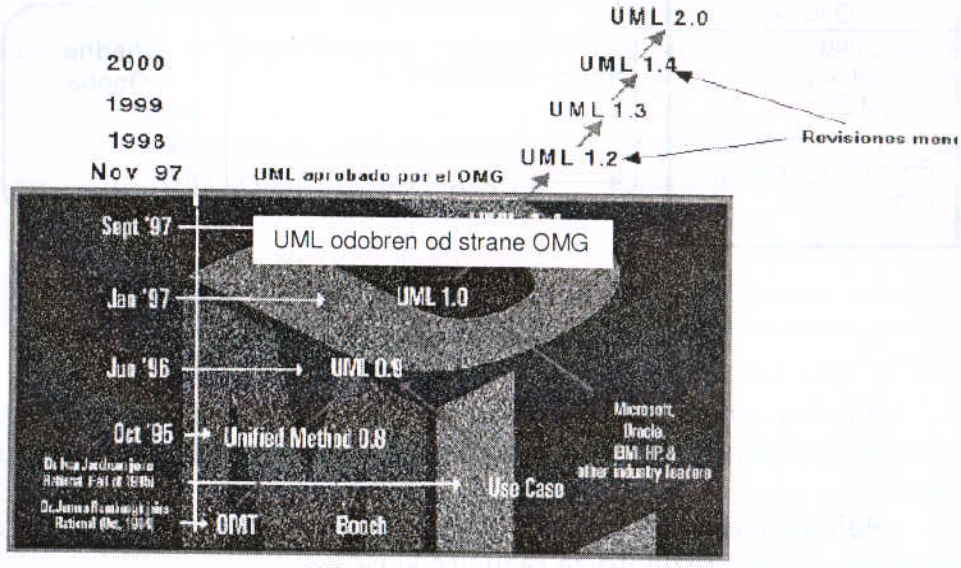
Tradicionalne metode analize i dizajna (strukturalna metodologija) nisu najbolje prilagođene ovim programskim jezicima



Slika 3.1.: *A ring to bind them all . . .*

Unified Modelling Language ili *UML*, je jezik za grafičko modeliranje, konstrukciju i dokumentaciju elemenata koji formiraju softverski sistem objektno orjentisan. Postao je standard u softverskoj industriji.

Notacija UML-a je opšte prihvaćena zahvaljujući prestižnosti njenih kreatora (Grady Booch, Ivar Jacobson i Jim Rumbaugh) i zbog toga što sadrži prednosti svih metoda na kojima se bazira (u osnovi Booch, OMT i OOSE). UML je stavio tačku na takozvani «rat metoda» koji smo imali tokom devedesetih godina. Sa UML-om se ujedinuju notacije navedenih metoda i napravljen je jedinstveni alat koji dele svi inženjeri softvera koji rade na objektno orjentisanom razvoju softvera.



Slika 3.2.: Istorija UML-a

3.3. Procesi objektno orjentianog razvoja informacijskih sistema

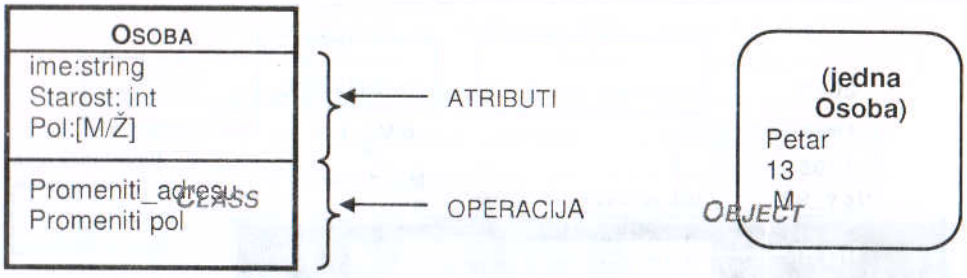
3.3.1. Osnovni koncepti objektno orjentisane paradigme

Objekti su osnovni elementi objektno orjentisane (OO) paradigme. Prezentiraju entitete iz realnog sveta (klijent, bankovni račun, student, itd.). Poseduju atribute (ime klijenta, adresa, i td.), operacije (ponašanje) i sopstveni identitet nezavisan od vrednosti atributa. Objekti su vlasnici svojih atributa i može im se prići samo putem operacija.

Klase su apstrakcije koje opisuju važna svojstva jedne aplikacije. Objekti se grupišu u klase. Klasa je najvažniji koncept OO. Apstraktna su dva tipa svojstava:

- *atributi*: podaci koji karakterišu objekte klase,
- *operacije(metode)*: ponašanje objekata klase. To su akcije ili transformacije koje objekti realizuju ili trpe.

Odabir klasa zavisi od potreba sistema.



Slika 3.3.: Klasa i objekat

Poruka (massage) je način na koji komuniciraju objekti. Porukom možemo reći jednom objektu da realizuje određenu operaciju.

Polimorfizam je koncept koji definiše slučaj kada se jedna operacija ponaša različito za različite klase.

Nasleđivanje je relacija specijalizacije između različitih klasa. Potklase specijaliziraju podatke i ponašanja superklasa.

Prednosti OO metoda:

1. Smanjuje kompleksnost.
2. Svi objekti su definisani, implementirani i testirani tako da se mogu ponovo upotrebiti u drugim sistemima.
3. Sistemi razvijeni ovom metodom su fleksibilniji, tako da se mogu modificirati ili dodati novi tipovi objekata.
4. Analitičari rade na nivou realnog sveta, kao i korisnici.
5. OO metode su idealne za razvoj Web aplikacija.
6. OO metode opisuju različite elemente IS-a u korisničkim terminima, tako da korisnik može lakše da shvati šta će raditi novi sistem i kako će postizati ciljeve.

3.3.2. Aktivnosti softverskog inženjeringa u procesu objektno orijentisanog razvoja informacionih sistema

Svaki složeni softver treba razvijati *iterativno i inkrementalno*. To znači da se postepeno, u svakom koraku, sprovodi celi ciklus: zahtevi, analiza, dizajn, implementacija za manji skup slučajeva upotrebe. U svakom sledećem koraku (iteraciji), dodaju se realizacije novih slučajeva upotrebe, tako da sistem postepeno dobija na funkcionalnosti i složenosti. Svaki sistem koji dobro funkcioniše po pravilu je nastao iz manjeg sistema koji je dobro funkcionisao.

Imajući u vidu postavke vezane za IDEF0[2], UML[3], IDEF1X[4], kao i potrebe za reinženjeringom poslovnih procesa, može se reći da se objektno orjentisani razvoj informacionih sistema izvodi kroz četiri procesa:

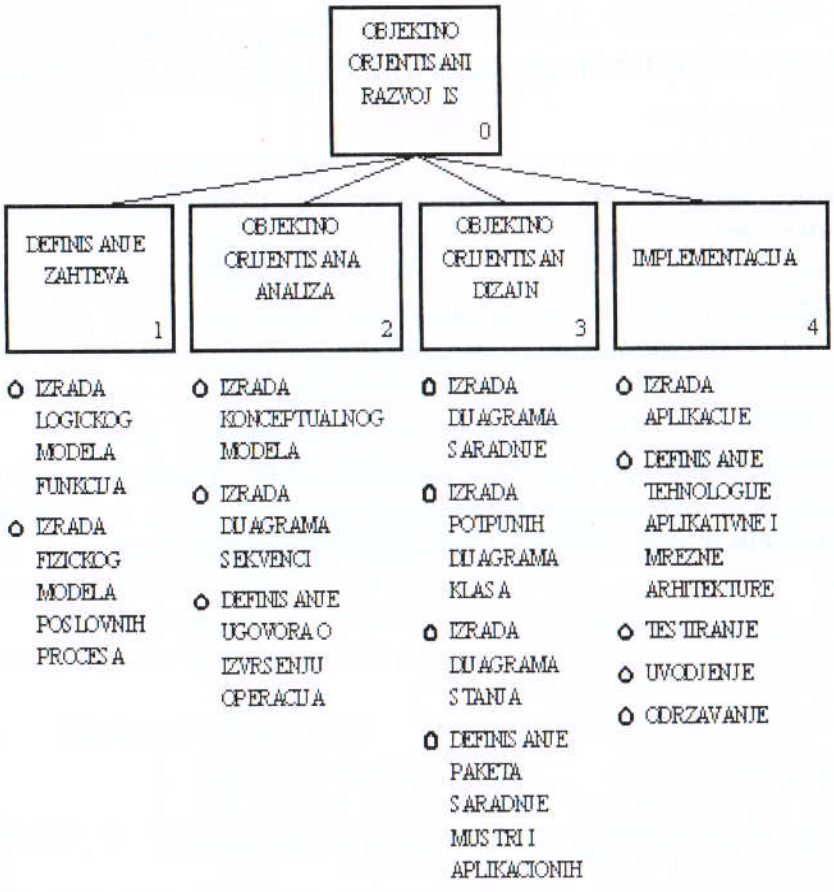
- Definisanje zahteva,
- Objektno orjentisana analiza,
- Objektno orjentisani dizajn,
- Implementacija.

Na slici 3.4. prikazano je stablo aktivnosti vezano za objektno orijentisani razvoj informacionih sistema.

[2] *IDEF0 je metoda dizajna za modeliranje odluka, akcija i aktivnosti jedne organizacije ili sistema.*

[3] *UML(Unified Modeling Language) je case alat za specifikaciju, konstrukciju i dokumentaciju sistema OO metodom*

[4] *IDEF1X je metoda dizajna relacionih baza podataka koja posjeduje sintaksu koja podržaje neophodnu semantiku za razvoj konceptualnih šema.*



Slika 3.4.: Stablo aktivnosti postupka objektno orijentisanog razvoja informacionog sistema

3.3.2.1. Definisanje zahteva

Prvi proces "Definisanje zahteva" se definiše kao:

- Izrada logičkog modela funkcija i
- Izrada fizičkog modela poslovnih procesa.

U okviru aktivnosti "**Izrada logičkog modela funkcija**" treba izvršiti funkcionalnu specifikaciju informacionog sistema. Logički model funkcija definisan je nezavisno od organizacionog ili tehnološkog okruženja u kome će posao biti implementiran. Logički

model funkcija je stabilniji, sporije se menja i može se potpuno ili delimično ponoviti i u drugim organizacijama.

Ovom aktivnošću identifikuju se granice posmatranog sistema, vertikalno povezivanje funkcija kroz definisanje stabla logičkih funkcija, horizontalno povezivanje kroz izradu dekompozicionog dijagrama i definisanje logike primitivnih funkcija.

Aktivnost "Izrada fizičkog modela poslovnih procesa" detaljno opisuje poslovne procese (koristeći UML dijagram aktivnosti) kao sekvence aktivnosti koje se obavljaju u konkretnom organizacionom i tehnološkom okruženju (organizaciona struktura, sistematizacija radnih mesta, tehnologija obavljanja posla). U ovoj aktivnosti se razmatraju poslovi (poslovni procesi) koji se moraju uraditi po određenom redu izvršavanja. Fizički model poslovnih procesa podložan je čestim izmenama zbog promene organizacije i tehnologije obavljanja posla. Ovom aktivnošću definiše se dinamika, odnosno način odvijanja posla gde se poslovni procesi dekomponuju do nivoa primitivnih procesa.

Poslovni procesi se opisuju slučajevima upotrebe. Slučajevi upotrebe opisuju funkcionalnost sistema iz korisničke perspektive i polazni su korak za prikaz upotrebe sistema od strane budućih korisnika u raznim karakterističnim situacijama. Opisivanje dinamike slučajeva upotrebe kao i logike jednog slučaja upotrebe izvodi se *sistemskim dijagramom sekvenci* (sekvencijalnim dijagramom) čime se definiše redosled poruka ili dijagramom aktivnosti za opis onih slučajeva upotrebe gde se opisuje paralelizam u procesima. Implementacija slučajeva upotrebe izvodi se preko saradnje (kolaboracije).

Definisanjem zahteva izvršena je identifikacija sistema nalaženjem funkcionalnog modela sistema. U sledećem koraku potrebno je izvršiti realizaciju (objektno orijentisanu analizu i objektno orjentisani dizajn) sistema nalaženjem modela sistema u izabranom prostoru stanja.

UML-ov dijagram stanja

Automat stanja (*state machine*) je ponašanje koje specificira sekvence stanja kroz koje prolazi neki objekat i modelira istoriju života nekog objekta.

Objekat može biti instanca klase, slučaja korišćenja ili čak sistem u celini. Objekat reaguje na događaje promenom stanja koje takođe izaziva nove događaje.

Dijagrami stanja prikazuju automate stanja fokusirajući se na događajima vođeno ponašanje. Dijagrami aktivnosti takođe prikazuju automate stanja, ali se fokusiraju na tok aktivnosti. Dijagrami stanja se kreiraju za apstrakcije čiji objekti pokazuju bitno dinamičko ponašanje.

Automat stanja se primenjuje da specificira ponašanje:
- objekata koji moraju odgovarati na asinhrone događaje,
- objekata čije tekuće ponašanje zavisi od istorije,

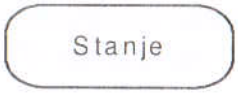
Uspešno se koristi za modeliranje ponašanja reaktivnih sistema (onaj koji odgovara na signale koje daju akteri iz spoljašnjeg sveta).

Elementi dijagrama stanja su:
- stanja,
- događaji koji uzrokuju promenu (tranziciju) stanja,
- akcije koje su rezultat promene stanja.

Stanje objekta je uslov ili situacija u kojoj taj objekat može da postoji. U jednom stanju objekat zadovoljava neki uslov, obavlja neku aktivnost ili čeka događaj.

Primeri:
- uslov - student je u stanju GLADAN ili SIT
- aktivnost - student je u stanju VEČERA
- čekanje - student je počeo sa jelom, i čeka neki prekidni signal koji će ga obavestiti da je SIT

Grafička notacija stanja je pravougaonik sa zaobljenim uglovima:



Početno i završno stanje su dva specijalna stanja:

početno  završno: 

Početno i završno stanje su pseudostanja. Tranzicija iz početnog u završno stanje može imati sve elemente osim okidajućeg događaja.

Elementi stanja su:

- Ime - tekst koji razlikuje jedno od drugih stanja; stanje može biti i anonimno (bez imena)

-Ulazna akcija - atomska radnja koja se obavi pri ulasku u stanje

- Izlazna akcija - atomska radnja koja se obavi pri izlasku iz stanja

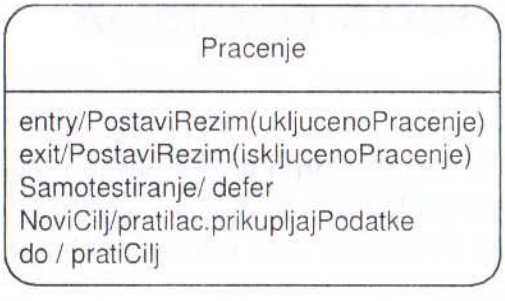
- Aktivnost - neatomska radnja koja se izvršava dok je objekat u datom stanju

- Podstanja - stanja koja postoje unutar datog stanja, sekvencijalno ili konkurentno aktivna

- Odloženi događaji - lista događaja koji se ne obrađuju u datom stanju nego se smeštaju u red

- Unutrašnje tranzicije - tranzicije koje obrađuju događaj i zadržavaju objekat u istom stanju; različite su od samotranzicije: ne izazivaju izlaznu pa ulaznu akciju

Grafički prikaz:



Slika 3.5.: Unutrašnje tranzicije stanja

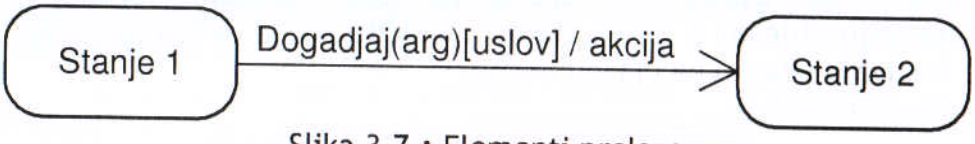
Prelaz (tranzicija) je relacija između dva stanja. Ukazuje da objekat napušta jedno stanje, obavlja akciju i ulazi u drugo stanje kada se dogodi specificirani događaj i kada je ispunjen specificirani uslov.

Grafička notacija - strelica:

Slika 3.6.: Prelaz iz stanja u stanje

Elementi prelaza:

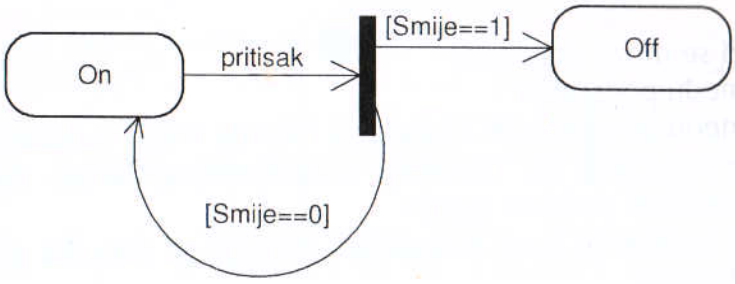
- Događaj je zbivanje koje nema trajanje i može prouzrokovati prelaz
- Zaštitni uslov je Boole-ov izraz koji čini prelaz mogućim kada je uslov ispunjen
- Akcija je atomska radnja koja je pridružena prelazu i može biti:
  - poziv operacije objekta vlasnika automata stanja ili drugog objekta koji je vidljiv datom objektu
  - kreiranje ili uništavanje drugog objekta
  - slanje signala nekom objektu (ključna riječ *send*)



Slika 3.7.: Elementi prelaza

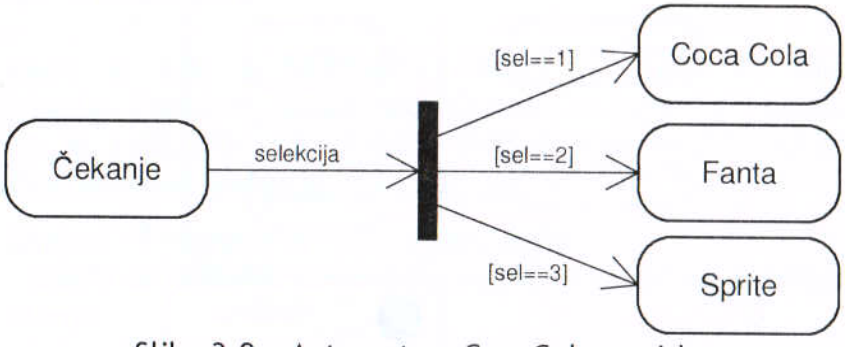
Grananje prelaza

Svaki prelaz može da ide do stanja ili do spojne tačke. Ako ide do spojne tačke onda se prelaz dijeli na segmente, i za svaki segment posebno možemo da postavimo neki uslov i/ili akciju.



Slika 3.8.: Grananje prelaza

Primer dijagrama stanja:



Slika 3.9.: Automat za CocaCola napitke

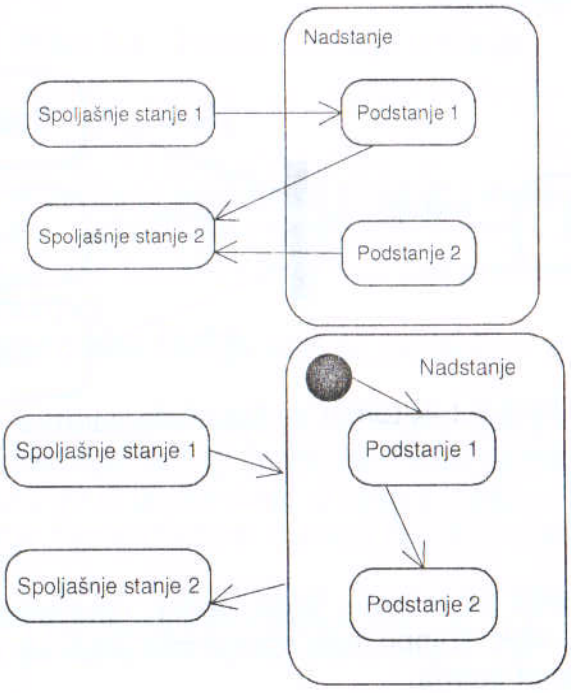
Vrste stanja

Jednostavno stanje je stanje koje nema unutrašnju strukturu automata stanja, dok kompozitno stanje ima unutrašnja stanja, tj. predstavlja automat stanja.
Ugnježđena stanja se koriste da bi se smanjila grafička kompleksnost.
Nadstanje (kompozitno stanje) je stanje koje obuhvata više unutrašnjih (ugniježđenih) stanja.
Podstanje je unutrašnje (ugnježđeno) stanje.
Kada se objekat nalazi u podstanju - istovremeno se nalazi i u nadstanju.
Postanja mogu biti:

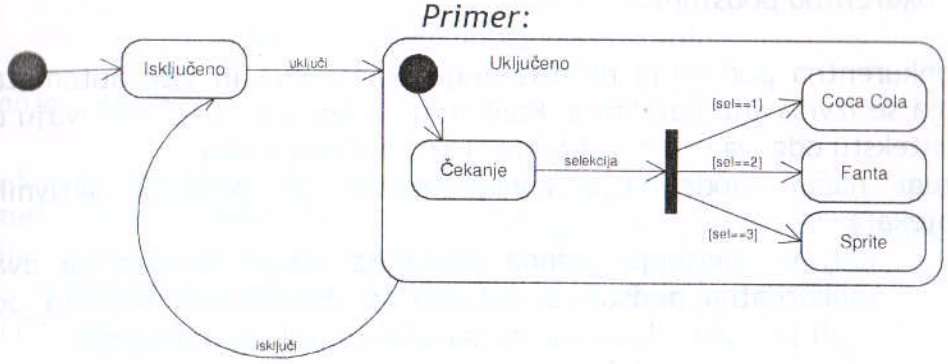
- sekvencijalna,
- konkurentna (paralelna).

Sekvencijalna podstanja

- Prelazi se mogu događati:
  - između podstanja
  - između podstanja ili nadstanja i stanja izvan nadstanja
- Ako je nadstanje cilj tranzicije iz spoljašnjeg stanja - nadstanje mora sadržati početno stanje
- Ako je nadstanje izvor tranzicije - najprje se napušta podstanje pa nadstanje
- Pri tranziciji u/iz nadstanja izvršavaju se ulazne/izlazne akcije i nadstanja i podstanja



Slika 3.10.: Podstanja

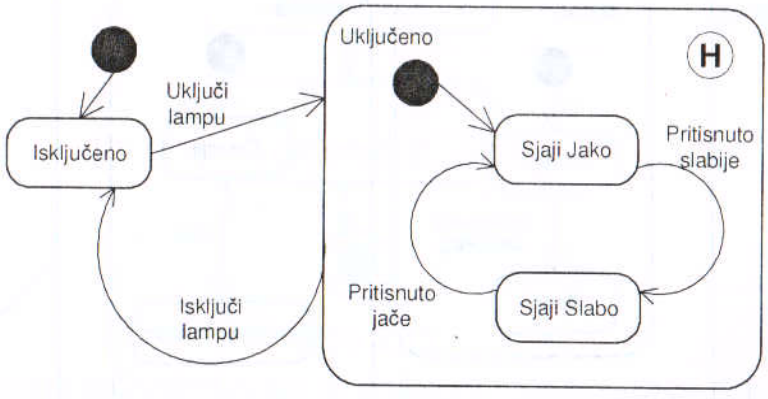


Slika 3.11.: Prošireni automat za CocaCola Napitke

Stanje sa historijom

Kada se uđe u nadstanje obično se kreće od inicijalnog podstanja. Ponekad postoji potreba da se krene od podstanja iz kojeg je nadstanje napušteno. Simbol H u kružiću ukazuje da nadstanje pamti historiju.

- Simbol (H) označava "plitku" historiju:
  - pamti se historija samo neposredno ugnježđenog automata stanja
- Simbol (H*) označava "duboku" historiju
  - pamti se historija do najugnježđenijeg automata stanja proizvoljne dubine



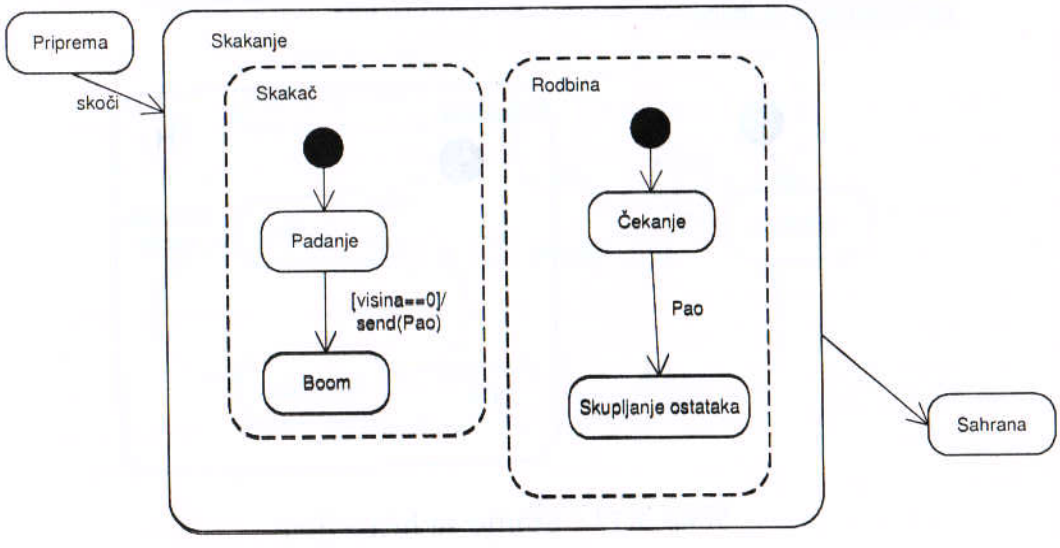
Slika 3.12.: Stanje sa historijom

Konkurentna podstanja

Konkurentna podstanja predstavljaju stanja dva ili više automata koja se izvršavaju paralelno. Konkurentna podstanja se izvršavaju u kontekstu odgovarajućeg objekta, kao i sekvencijalna.
Drugi način modeliranja konkurentnosti je pomoću aktivnih objekata

- umjesto deljenja jednog automata stanja objekta na dva konkurentna podstanja definišu se dva aktivna objekta od kojih je svaki odgovoran za ponašanje jednog podstanja

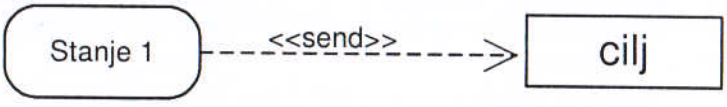
- Od više sekvencijalnih podstanja na jednom nivou - objekat može biti samo u jednom
- Od više konkurentnih podstanja na jednom nivou - objekat je u svakom od njih
- Prelaz u stanje sa konkurentnim podstanjima predstavlja *fork* grananje
- Ako jedno konkurentno podstanje stigne do završnog stanja pre drugog - čeka na drugo
- Sva konkurentna podstanja moraju biti završena da bi se izvršio prelaz *join* iz nadstanja
- Ugnežđeni konkurentni automati stanja ne mogu imati početno, završno i stanje historije
- **Sekvencijalna podstanja koja obrazuju svako od konkurentnih podstanja imaju ova stanja**



Slika 3.13.: Konkurentna podstanja

Slanje signala

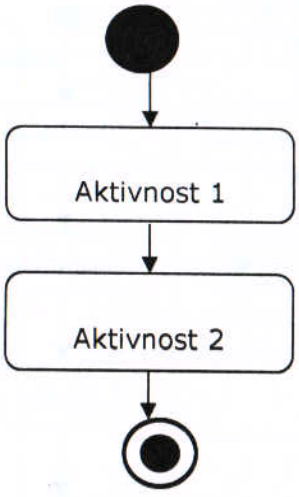
Signal se šalje komandom *send* čiji je opšti oblik: *send (ImeDogadjaja, imeStanja)*. Ako ImeDogadjaja jednoznačno određuje događaj, onda se imeStanja može zanemariti. Kada se iz akcije šalje signal nekom objektu, taj se objekat može prikazati na dijagramu.



Slika 3.14.: Slanje signala

UML-ov dijagram aktivnosti

Dijagrami aktivnosti služe za pojednostavljeni prikaz događanja tokom operacije ili procesa. Svaka aktivnost je prikazana sa elipsom (zaobljenim pravougaonikom). Strelica prikazuje prelaz s jedne aktivnosti na drugu. Početak je prikazan punim krugom, a kraj metom, Slika 3.15:

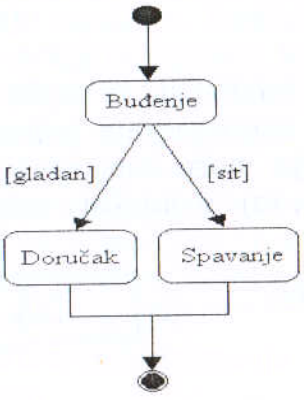


Slika 3.15

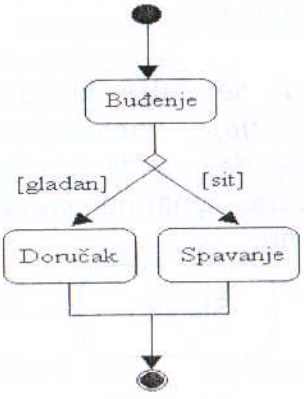
Tačke grananja aktivnosti mogu se prikazati na dva načina, Slika 3.16. a i b:

- staze izlaze (dolaze) direktno iz aktivnosti

-prelaz aktivnosti iz malog dijamanta i onda moguće staze iz tog simbola.

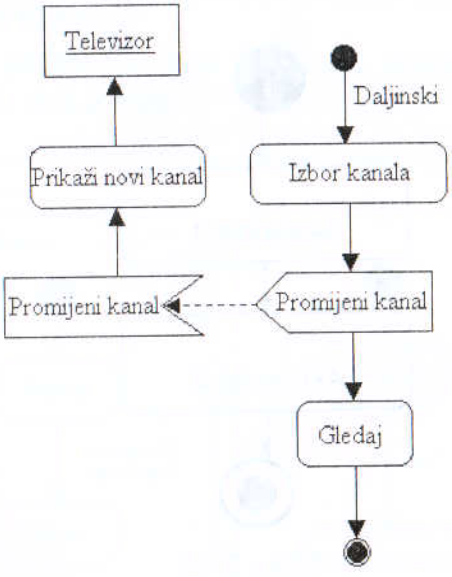


Slika 3.16.a



Slika 3.16.b

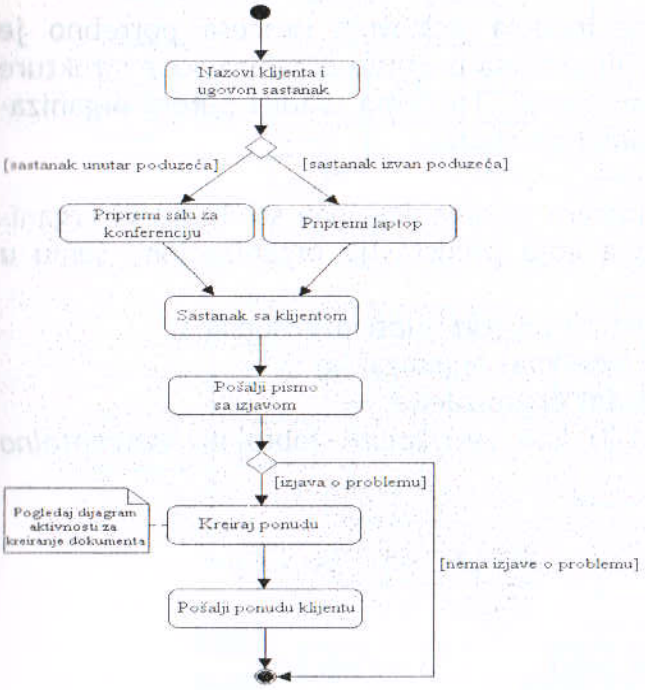
Tokom aktivnosti može se poslati ***signal***. Signal koji se šalje prikazan je konveksnim petouglom, a signal koji se prima, konkavnim petouglom, Slika 3.17.

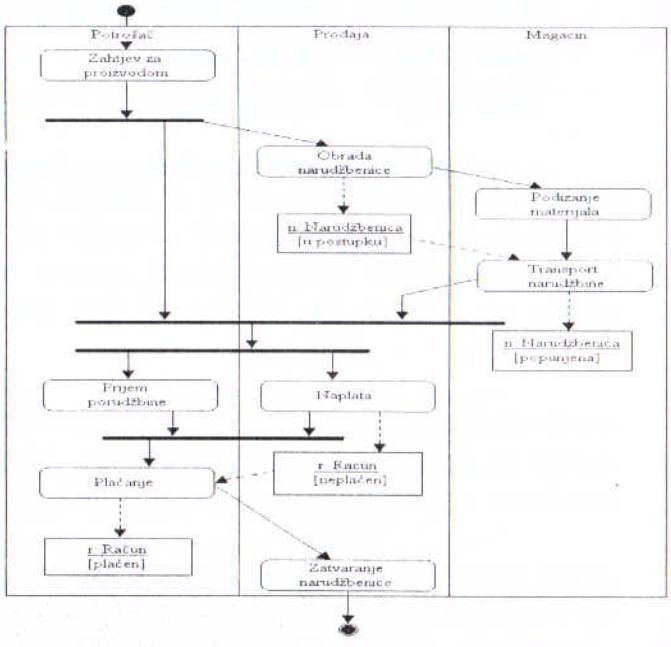


Slika 3.17.

Dijagram aktivnosti vizuelno prikazuje paralelne segmente pomoću *swimlanea*. Svaki *swimlane* pokazuje naziv (na vrhu) i ulogu aktivnosti,

Slika 3.18.:





Slika .19.Primer

Kupovina

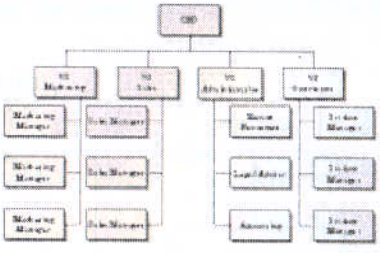
Organizacioni dijagram (Organization chart)

U okviru fizičkog modela poslovnih procesa potrebno je predstaviti okruženje ovih procesa u smislu organizacione strukture ili sistematizacije radnih mesta. To ćemo uraditi putem organizacionog dijagrama (*organization chart*).

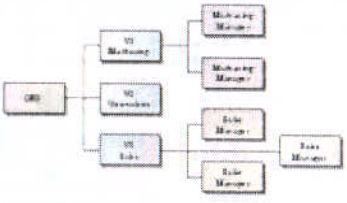
Organizacioni dijagram je organizaciona struktura ili organizaciona (ustrojena) šema koja predstavlja organizacionu šemu u obliku stabla i ilustruje:

- odnose (relacije) među ljudskim resursima,
- sektorima (odelima) organizacije ili
- podele unutar organizacije.

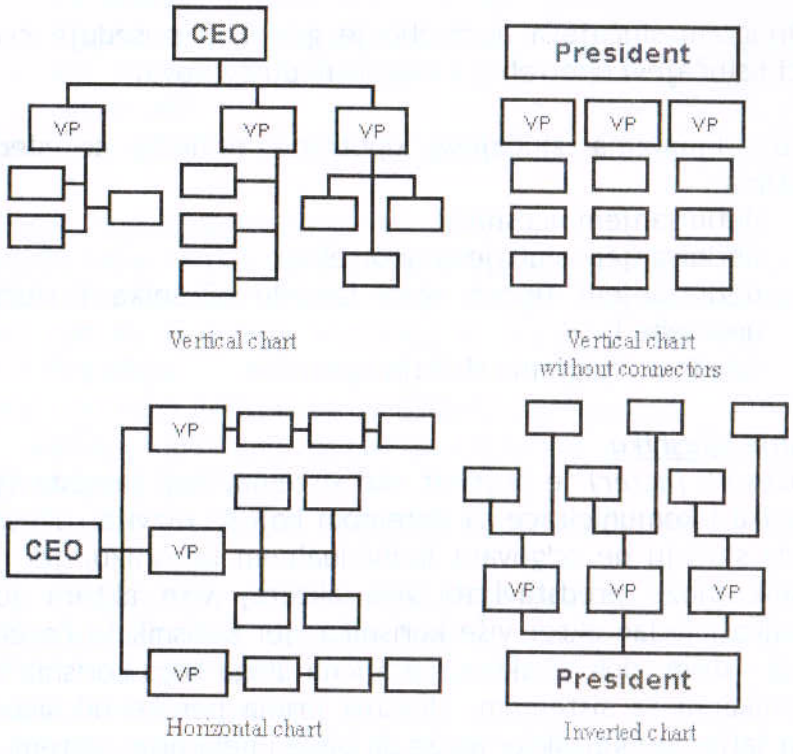
Ova šema se može crtati kao *vertikalno* stablo ili *horizontalno* stablo, Slika 3.20:



Vertikalni organization chart



Horizontalni organization chart



Slika 3.20

UML-ov Use case dijagram

Use case predstavlja sistem sa stanovišta korisnika i pomaže nam da shvatimo korisničke zahteve. Dijagrami slučajeva upotrebe *(use case)* se definišu na osnovu *funkcionalnog modela sistema* i opisuju interakciju objekata koji se nalaze van sistema sa samim informacionim sistemom.

Skup slučajeva upotrebe predstavlja sve pretpostavljene načine korištenja sistema, tj. određuje ponašanje sistema ili njegovog dela i opisuje akcije koje sistem izvodi da bi postigao rezultate koji su od koristi korisniku.

Slučajevi upotrebe se koriste da bi se ostvarilo željeno ponašanje sistema koji se razvija, bez obaveze da se odrede načini realizacije ponašanja. Slučajevi upotrebe koriste krajnjim korisnicima da razumeju sistem.

Dijagram slučajeva upotrebe je graf koji poseduje čvorove (učesnici i slučajevi upotrebe) i veze između čvorova.

Razvoj dijagrama slučajeva upotrebe definiše se sledećim aktivnostima:

- definisanjem učesnika,
- definisanjem slučajeva upotrebe,
- definisanjem tipova veza između učesnika i slučajeva upotrebe i
- izradom dijagrama slučaja upotrebe.

Definisanje učesnika

Učesnik (acter) je objekat van sistema, koji predstavlja tip korisnika i komuniciraće sa sistemom koji se razvija. Učesnik ni u kom slučaju ne odgovara individualnom korisniku, jer jedan korisnik može predstavljati više aktera, više aktera jednog korisnika i jedan akter više korisnika, jer korisnik je čovek koji koristi sistem, dok je akter specifična uloga koju korisnik ima u komunikaciji sa sistemom. Učesnik prima poruke od sistema i istom šalje poruke. Akter može da bude i neki drugi sistem.

Učesnik je jedna vrsta klase i može da bude u svim relacijama koje važe i za klase. Jedna važna relacija je generalizacija, gde izvedeni učesnik ima sve osobine osnovnog učesnika i može da učestvuje u svim slučajevima upotrebe kao i osnovni učesnik.

Učesnike je moguće identifikovati na osnovu odgovora na sledeća pitanja:

- Ko će koristiti osnovnu funkcionalnost sistema (primarni akteri)?
- Kome će biti potrebna podrška sistema u obavljanju dnevnih zadataka?
- Ko treba da upravlja, administrira i održava sistem(sekundarni akteri)?
- Kojim hardverskim uređajima treba da upravlja sistem?
- S kojim drugim sistemima dotični sistem treba da bude u vezi? Ti sistemi mogu da se podele na sisteme koji će da iniciraju komunikaciju sa našim sistemom i na one koje će naš sistem da kontaktira. Sistemi mogu da budu

računarski sistemi, kao i druge aplikacije na računaru u kojima se sistem nalazi.
- Ko ili šta je zainteresovan za rezultate koje sistem proizvodi?

Učesnici su u tesnoj vezi sa slučajevima upotrebe.

Definisanje slučajeva upotrebe

Opisuje karakteristične sekvence akcija u tipičnim situacijama upotrebe sistema, što znači da je to tehnika kojom se snima poslovni proces sa strane korisnika ili scenario koji opisuje skup sekvenci događaja.

Slučajevi upotrebe grafički se predstavljaju elipsom i imenom.

Naziv slučaja upotrebe treba da bude ne predugačak i jasan (tipično glagolski oblik).

Slučaj upotrebe opisuje se tekstualno gdje se detaljno opisuje *šta radi*.

- Svaki primitivni proces predstavlja jedan slučaj upotrebe.
- Slučaj upotrebe definiše funkcionalnost sistema sa strane korisnika. Korisnici ga iniciraju. Svako izvođenje slučajeva upotrebe = **instanca**.
- Opisuje normalan način rada ne uzimajući u obzir moguće greške ili anomalije → šablon ponašanja delova sistema.

Zaključak: logička jedinica posla ili sekvencija bliskih transakcija koje izvode korisnici u dijalogu sa sistemom.

Karakteristike slučajeva upotrebe:

- slučaj upotrebe uvek inicira učesnik, tj. izvršava se na zahtev učesnika koji mora direktno ili indirektno da «naredi» sistemu da izvrši neki slučaj upotrebe (ponekad učesnik ne mora da bude svestan da je on inicirao slučaj upotrebe);
- slučaj upotrebe pruža neku vrednost za učesnika, koja ne mora uvek da bude upadljiva, ali mora da bude prepoznatljiva;
- slučaj upotrebe mora da bude definisan kao potpun opis, a nije potpun sve dok se neka vrednost ne proizvede;
- česta greška je podeliti jedan slučaj na manje slučajeve upotrebe, koji se implementiraju jedni druge, slično funkcionalnim pozivima u programskim jezicima;
- slučaj upotrebe je jasno uočljivo ponašanje sistema koje se može testirati;

- može se shvatiti i kao odgovor sistema na neki događaj ili spoljnu akciju koju proizvede učesnik.
- slučaj upotrebe će biti implementiran pomoću scenarija koji ispunjava dato ponašanje sistema;
- slučaj upotrebe navodi samo šta sistem radi, ali ne i kako to radi;
- specifikacija slučaja upotrebe predstavlja skup opisa tokova događaja, uključujući i varijante (izuzetne situacije, situacije sa greškama i sl.);
- slučajeve upotrebe treba navoditi eksplicitno, u vidu tačaka nabrajanja;
- za svaki slučaj upotrebe treba navesti ko ga i kako pokreće (inicira), kao i precizan tok događaja, sa detaljima o tome koje informacije, i kada, učesnik i sistem razmenjuju;
- opis slučaja upotrebe može da ima navedene i ostale zahteve vezane za ponašanje: izgled korisničkog interfejsa, zahteve u pogledu performansi, zahteve u pogledu pouzdanosti i slično.

Za svakog učesnika mogu se definisati sledeća pitanja, kojima se identifikuju slučajevi upotrebe:

- Koje funkcije učesnik zahteva od sistema?
- Da li učesnik treba da čita, kreira, briše, izmeni, ili da unese neke informacije u sistem?
- Da li učesnik treba da bude obavješten o događajima u sistemu i da li učesnik treba da obavesti sistem o promenama u okruženju? Šta ovi događaji predstavljaju u odnosu na funkcionalnost?
- Da li svakodnevni rad učesnika može da se pojednostavi kroz nove funkcije sistema (obično funkcije koje trenutno nisu automatizovane kroz sistem).
- Kakav ulaz/izlaz zahteva sistem? Gdje ulaz/izlaz ide, odnosno odakle dolazi?
- Koji su glavni problemi trenutne implementacije sistema?

Na osnovu postavljenih pitanja odgovori se definišu u okviru scenarija, odnosno tekstualnog opisa slučajeva upotrebe.

Tekstualni opis slučajeva upotrebe je poseban dokument koji služi za dalju detaljizaciju informacija definisanih u modelu slučaja upotrebe.

Svrha opisa slučaja upotrebe je definisanje prioriteta za izvršavanje grupa logičkih zahteva i određivanje nivoa prioriteta baziranih na važnosti identifikovanog opsega projekta.

Definisanje veza između korisnika i slučajeva upotrebe

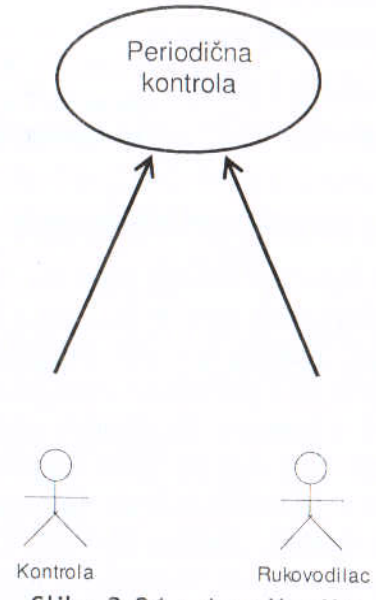
Slučajevi upotrebe po pravilu nisu nezavisni. Analizom uočavamo relacije između slučajeva upotrebe, jer se time složena funkcionalnost dekomponuje i uočavaju se zajednički delovi. ***Jedan čvor učesnika uvijek je povezan barem sa jednim čvorom slučaja upotrebe i obrnuto - jedan čvor slučaja upotrebe povezan je sa barem jednim čvorom učesnika.***

Direktna interna komunikacija između konkretnih slučajeva upotrebe nije dozvoljena. Međutim, mogu se definisati asocijacije između slučajeva upotrebe i između učesnika, da bi se jednostavnije prikazao neki složeni model.

U okviru UML standarda definiršu se sledeće veze:

- asocijacija (association)
- asocijacija između slučajeva upotebe
  <<include>> ili <<extends>>
- generalizacija (generalization-inheritance)
- zavisnost (depedency)

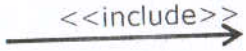
Asocijacija je bidirekcionalna veza koja spaja učesnike i slučajeve upotrebe. Asocijacija između samih učesnika, ili slučajeva upotrebe, definiše logičku povezanost tih elementata.



Slika 3.21.: Asocijacije

Asocijacije između slučajeva upotrebe su: *include* i *extend*.

<<include>>



U složenim sistemima se pojavljuju slučajevi upotrebe sa sličnim ponašanjem. Ovo slično ponašanje se izdvaja i dele ga slični slučajevi upotrebe.

3.3.2.2.Objektno orjentisana analiza

Proces "**Objektno orjentisana analiza**" razmatra objekte sadržane u realnom sistemi kao i njihove međusobne odnose. Objektno orijentisana analiza (OOA) ističe u prvi plan istraživanje problema tj. pronalaženje i opisivanje objekata ili koncepta u domenu problema, ne dajući odgovore na pitanje kako su rešenja definisana. OOA prestavlja najkritičniju fazu jer je potrebno uočiti koji se sve objekti pojavljuju u realnom sistemu. Zatim se vrši specifikacija najvažnijih atributa unutar objekata i interakcija između objekata. Analiza je proces ispitivanja korisničkih zahteva u cilju njihovog definisanja na osnovu čega se pristupa objektno orijentisanom dizajnu (OOD).

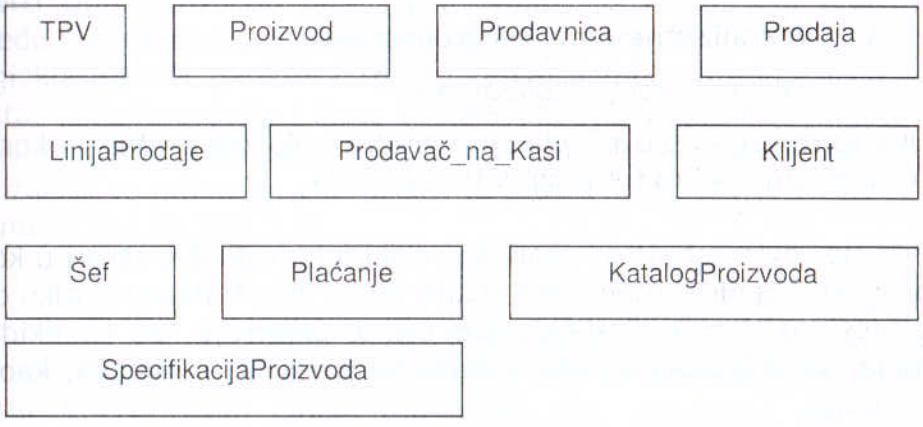
Ovaj proces se definiše kao:

- Izrada konceptualnog modela
- Izrada dijagrama sekvenci i
- Definisanje ugovora o izvršenju operacije

3.3.2.2.1.Izrada konceptualnog modela

Aktivnost "Izrada konceptualnog modela" treba da definiše dijagram klasa bez definisanih operacija za ciljne koncepte unutar domena posmatranja. Konceptualnim modelom definišu se koncepti, odgovarajući atributi i potrebne asocijacije između koncepata. Konceptualni model odgovara klasičnom modelu objekti veze. U okviru ove aktivnosti definiše se lista kandidata korištenjem liste kategorije koncepta i identifikacijom imenica iz fraza, potom se kreira konceptualni model, dodaju atributi i odgovarajuće asocijacije. Da zaključimo u ovoj aktivnosti se identifikuju uloge ljudi i stvari od interesa koji će biti angažovani u procesima.

U UML-u konceptualni model odgovara **diagramu klasa** sa: konceptima, relacijama među konceptima i atribitima koncepata. Prilikom identifikacije koncepte prepoznajemo u relevantnim imenicama. Bolje je preterati u specifikaciji konceptualnog modela sa mnogo koncepata (fine - grained) nego da bude nedovoljan broj koncepata.



Slika 3.22.: Identifikovani koncepti konceptualnog modela

Nakon idetifikacije koncepata potrebno ih je klasificirati po kategorijama koristeći **popis kategorija** koncepata.

Popis kategorija I
- Fizički objekti *(TPV, Avion)*,
- Opisi *(SpecifikacijaProizvoda, OpisLeta)*,
- Mesta *(Prodavnica, Aerodrom)*,
- Transakcije *(Prodaja, Plaćanje, Rezervisanje)*,
- Delovi transakcija *(ProdajaLinijaProizvoda)*,
- Uloge osoba *(Prodavač na kasi, Pilot)*,
- Kontejneri *(Prodavnica, Avion)*,
- Stvari unutar jednog kontejnera *(Proizvodi, Putnici)*,
- Drugi računarski sistemi *(SistemAutorizacijeKredita)*,

Popis kategorija II
- Organizacije *(Odsek za prodaju)*,
- Događaji *(Prodaja, Let)*,
- Procesi *(Rezervacijajedišta)*,
- Pravila *(Pravila plačanja)*,
- Katalozi *(KatalogProizvoda)*,
- Registri, ugovori *(Vračanje, Ugovor_o_Zapošljavanju),itd*

Identifikirane koncepte je potrebno povezati relacijama (vezama među konceptima). **Relacija** je odnos između dva ili više koncepata koja ukazuje na vezu između njih.
Postoje četiri tipa relacija unutar UML modela:

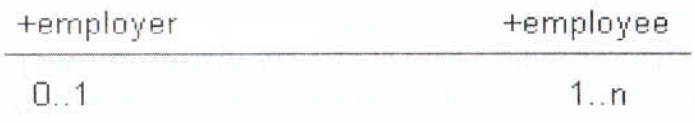
1. Ovisnosti *(eng. dependency)*
2. Asocijacije *(eng. association)*
3. Generalizacije *(eng. generalization)*
4. Realizacije *(eng. realization)*

Ove su relacije osnovni relacijski gradivni blokovi u UML-u i koriste se za pisanje dobro formisanih UML modela.

Prvi tip, *ovisnost*, je semantička relacija između dve stvari u kojoj promena u jednoj (neovisnoj stvari) može uticati na semantiku druge (ovisne stvari). Grafički, ovisnost se prikazuje kao isprekidana linija, sa mogućom oznakom direkcije (strelica) i imenom, kao na Slici 3.23.

Slika 3.23: Ovisnost

Drugi tip, *asocijacija*, je strukturalna relacija koja opisuje skup veza između objekata. *Kombinacija (eng. aggregation)* je specijalan oblik asocijacije i reprezentuje strukturalnu relaciju između celine i njenih delova. Grafički, asocijacija se predstavlja kao puna linija, uz mogućnost određivanja smera, i ponekad sadrži i dodatne stvari kao kardinalitet (n-arnost) i ime. Prikazana je na Slici 3.24.



Slika 3.24.: Asocijacija

Treći tip, *generalizacija (eng. generalization)*, je relacija specijalizacije/generalizacije u kojoj objekti specijaliziranih elemenata (deca) se mogu zameniti objektima generaliziranih elemenata (roditelja). Na ovaj način, deca dele strukturu i ponašanje roditelja. Grafički, relacija generalizacije se prikazuje kao puna

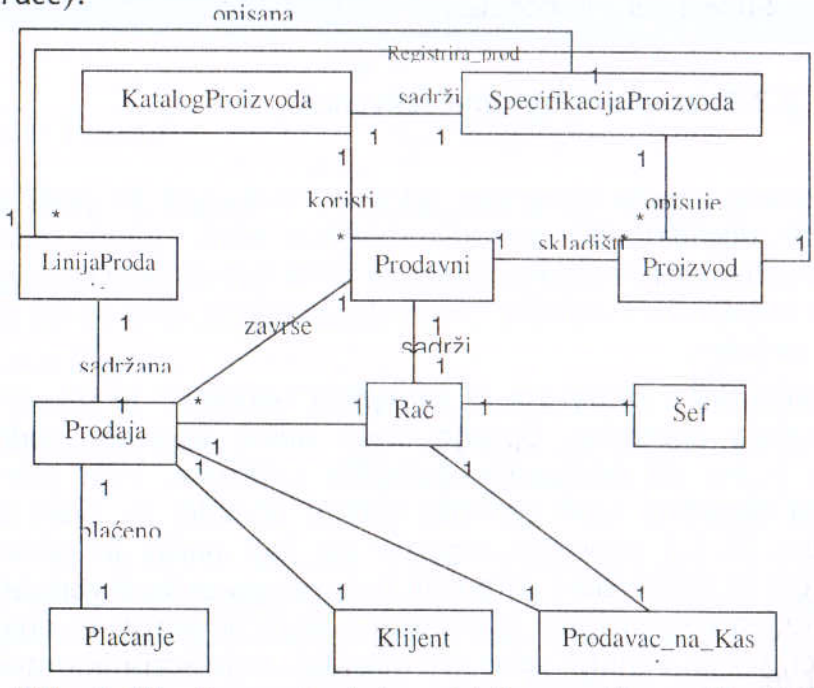
linija sa šupljom strelicom koja pokazuje prema roditelju, kao na Slici 3.25.

Slika 3.25.: Generalizacija

Četvrti tip, *realizacija (eng. realization)*, je semantička relacija između klasifikatora, gdje jedan klasifikator specificira ugovor koji drugi klasifikator garantira izvršiti. Realizacija se susreće na dva mesta: između sučelja i klasa ili komponenata koje ih realizuju, i između slučajeva korištenja i saradnji koje ih realizuju. Grafički, realizacija se prikazuje kao prelaz između generalizacije i relacije ovisnosti, kao na Slici 3.26.

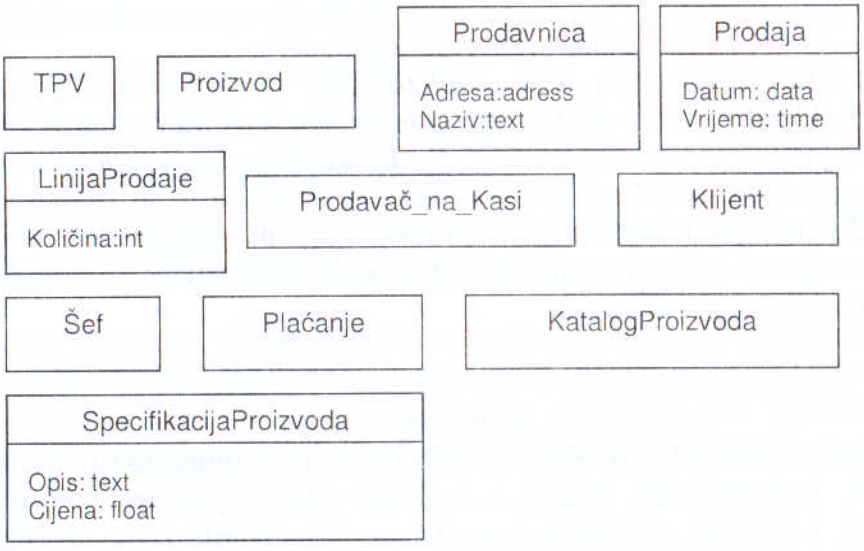
Slika 3.26.: Realizacija

Ova četiri elementa su osnovne relacije koje se mogu uključiti osim u UML konceptualni model i u druge UML dijagrame. Postoje, takođe, i njihove varijacije, kao što su uključivanje *(eng. include)*, proširenje *(eng. extend)*, rafiniranje *(eng. refinement)*, i praćenje *(eng. trace)*.



Slika 3.27.: Konceptualni model (koncepti, asocijacije i kardinaliteti)

Konceptima je potrebno dodeliti sopstvene atribute predviđene zahtevima. Atributi su važne karakteristike koncepata za domenu problema.



Slika 3.28.: Konceptualni model (koncepti i atributi)

3.3.2.2.2.Izrada dijagrama sekvenci

Aktivnost "Izrada dijagrama sekvenci" pokazuje za svaki slučaj upotrebe, događaje koje generiše spoljni učesnik i njihov redosled. Drugim rečima sekvencijalni dijagram prikazuje dinamičku saradnju između objekata u vremenu. Ovom aktivnošću se opisuje šta sistem radi, a ne kako.

Sekvencijalni dijagram ili dijagram sekvenci: je sekvencijali opis slučaja upotrebe. Opisuje svaki slučaj upotrebe, odnosno opisuje jednu od realizacija slučajeva upotrebe, koja pokazuje redosled dagađaja koje generišu spoljni učesnici za svaki slučaj upotrebe. Slučaj upotrebe sugeriše na koji način je učesnik u interakciji sa softverskim sistemom (*učesnik generiše događaje*).

Dakle, za sekvencijalni opis slučaja upotrebe koristi se dijagram sekvenci, jer on definiše sekvencu događaja koje korisnik (interfejs) prosleđuje sistemu u jednom slučaju upotrebe. Sistem je «crna kutija» koja pokazuje komunikaciju korisnika sa sistemom.

Dijagram sekvenci se koristi za specifikaciju vremenskih zahteva u opisu složenih scenarija, tj. za opis toka poruka između objekata kojima se realizuje odgovarajuća operacija u sistemu. Na dijagramu sekvenci se ne prikazuju veze između objekata, već se prikazuje šta sistem radi, tj. identifikuju se sistemski događaji i sistemske operacije za odgovarajući slučaj upotrebe.

Vreme u sekvencijalnom dijagramu se prikazuje u vertikalnoj, a objekti u horizontalnoj dimenziji.

Dijagram sekvenci je jedan od dijagrama interakcije. Interakcija je ponašanje koje obuhvata skup poruka koje se razmenjuju između skupa objekata u nekom kontekstu sa nekom namenom. Poruka je specifikacija komunikacije između objekata koja prenosi informaciju. Nakon poruke očekuje se da usledi aktivnost. Prijem jedne poruke se smatra instancom jednog događaja. Kada se pošalje poruka sledi akcija, odnosno izvršenje naredbe koja predstavlja apstrakciju metode.

Vrste akcija UML-a na osnovu poslatih poruka:

Tabela 3.1.

Akcija	Opis
Poziv (call)	Pokreće operaciju objekta primaoca (može pozivati i sam sebe)
Povratak (return)	Vraća vrednost primaocu
Slanje (send) operacija	Asinhrono se šalje signal primaocu
Kreiranje (create) <<create>>	Kreira se objekat
Uništavanje(destory) <<destory>>	Uništava se objekat (objekat može biti i suicidan)
Postajanje (become) <<become>>	Objekat menja prirodu (na obe strane veze je isti objekat)

Interakcija se koristi za modeliranje dinamičkih aspekata modela definisanih preko dijagrama interakcije koji čine dijagram sekvenci i dijagram kolaboracije. Dijagrami sekvenci naglašavaju vremensko uređenje interakcije, a kolaboracijski dijagram naglašava strukturnu vezu između učesnika u interakciji. Dijagram sekvenci je povoljniji za prikaz kompleksnih slučajeva upotrebe u interaktivnim

sistemima i sistemima koji rade u realnom vremenu, dok je kolaboracijski dijagram povoljniji za opis složenih algoritama.

Ove dve vrste dijagrama vizuelizuju na različite načine iste informacije. Semantički su potpuno ekivalentni i CASE alati ih automatski konvertuju jedan u drugi.

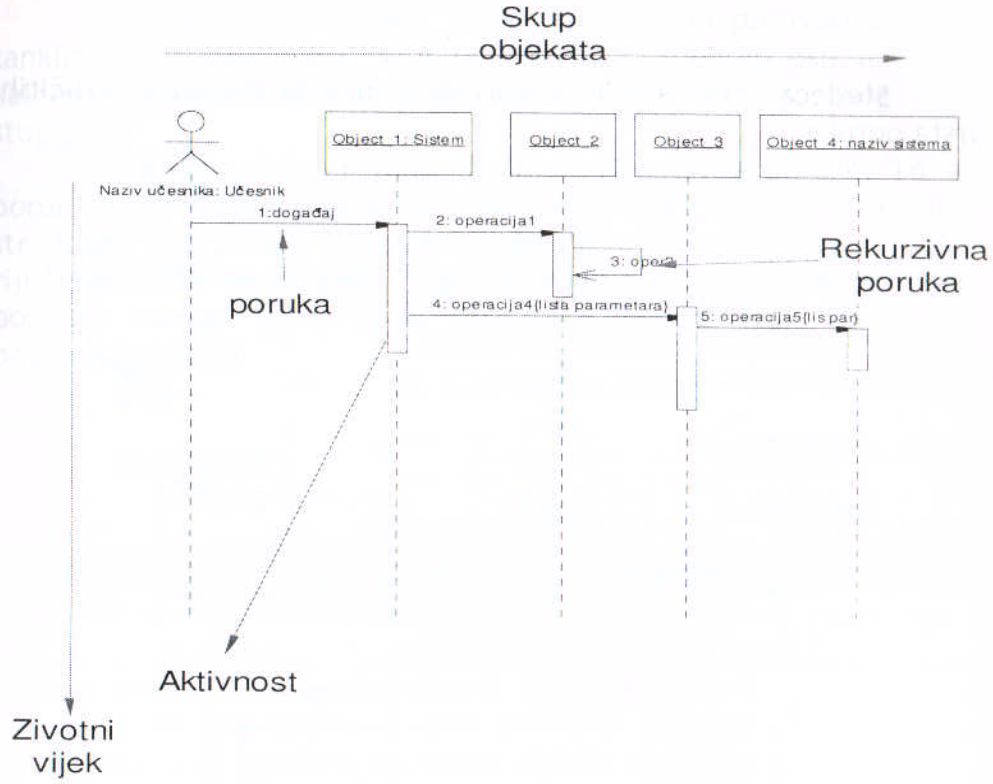
U okviru izrade dijagrama sekvenci definišu se sledeće aktivnosti:

1. definisanje objekata,
2. definisanje poruka,
3. definisanje indikatora sinhronizacije.

1.Definisanje objekata

Objekti u sekvencijalnim dijagramima su predstavljeni vertikalnim linijama. Na vrhu linije se navodi naziv objekta i/ili simbol objekta. Aktiviranje objekata se predstavlja *uskim pravougaonikom* na liniji objekta, a predstavlja operaciju (akciju) koju objekat, u periodu predstavljenog dužinom aktivacije, obavlja. Na vrh aktiviranog objekta se prikazuje događaj (poruka) koju je aktivirao objekat, a na dnu, povratna poruka objektu koji je aktivirao promatrani objekat. Povratna poruka često se ne prikazuje.

Sekvencijalni dijagram se formira tako što se prvo na vrh dijagrama duž ose X, postave objekti koji učestvuju u interakciji. Obično se objekti koji započinju interakciju stavljaju levo, a objekti koji slede redom desno. Nakon toga poruke koje ovi objekti šalju i primaju smeštaju se duž ose u rastućem nizu po vremenskom redosledu od vrha ka dnu. Ovo pruža jasnu asocijaciju na tok kontrole u protoku vremena.



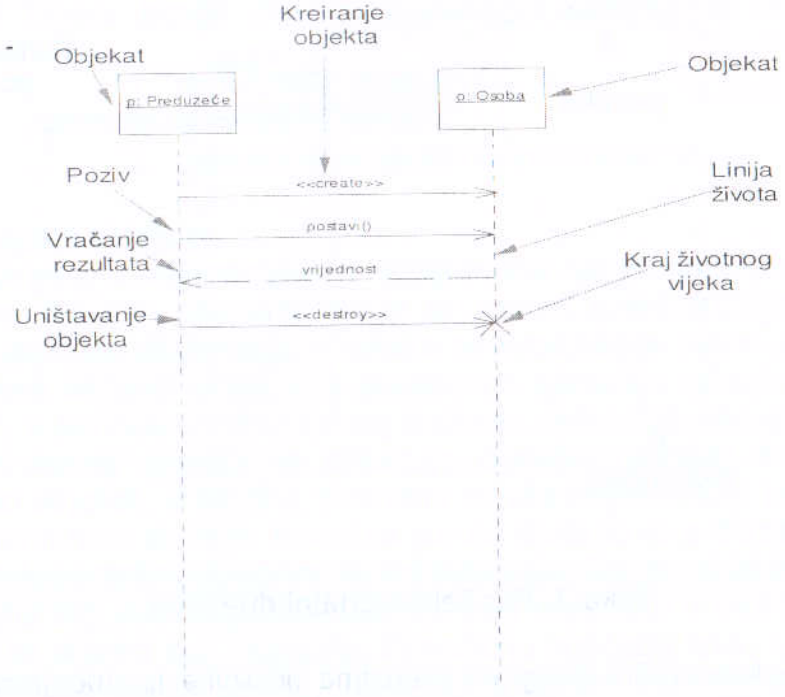
Slika 3.29.: Sekvencijalni dijagram

Sekvencijalni dijagram preuzima učesnike iz slučajeva upotrebe, a ako se prvo definiše konceptualni dijagram, preuzimaju se objekti. U sekvencijalnom dijagramu učesnici mogu biti korisnik ili veštački entitet (software ili entitet). Objekt je instanca koncepta koji šalje i prima poruke. Moguće je da objekat pošalje poruku sam sebi. Objekti se definišu nazivom objekta i nazivom koncepta, /koji su podvučeni/:

Naziv objekta: Naziv koncepta

Objekti se prikazuju u pravougaonicima i ime objekta se obavezno unosi. Iz svakog objekta polazi vertikalna isprekidana linija prema dole, koja predstavlja životni vek objekta *(lifeline)*. Životni vek objekta je vremenski period postojanja istog. Većina objekata postoji dok traje interakcija, ali objekti se mogu praviti i dok traje interakcija. Njihov životni vijek počinje nakon prijema poruke <<create>>. Objekti mogu biti uništeni u toku interakcije. Njihov životni vek se završava nakon prijema poruke *destroy*.

Sledeća slika je primer sekvencijalnog dijagrama i različitih vrsta poruka:



Slika 3.30.: Različite vrste poruka u sekvencijalnom dijagramu

<<create>> → kreira objekat za koncept osoba

- postaviti() → postavlja se osoba na radno mjesto
- akcija vrijednost → vraća se rezultat u objekat p koncepta preduzeće
- <<destroy>> → uništava se objekat koncepta osoba

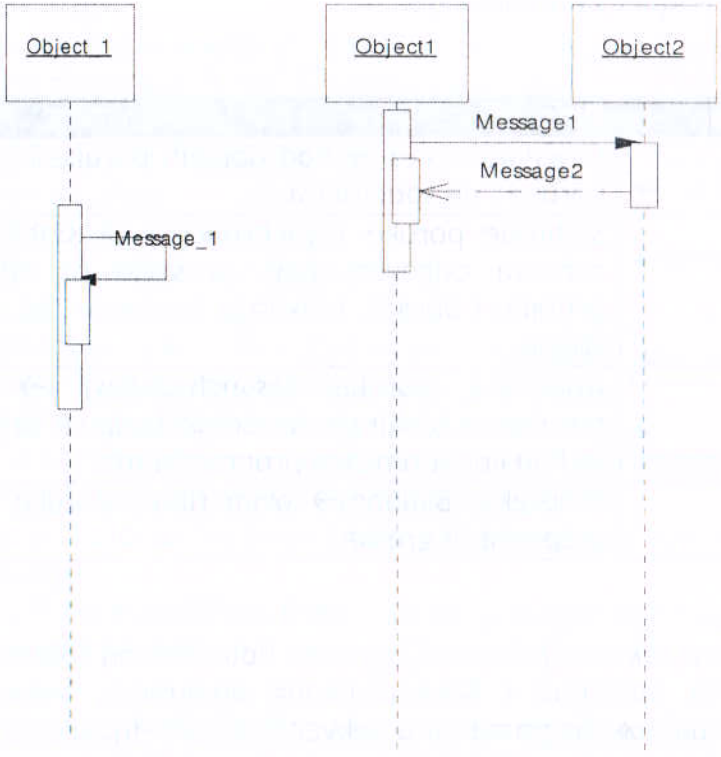
2.Definisanje poruka

Porukama (događajima) komuniciraju objekti. Poruke se prikazuju usmerenim linijama.

Argumenti se navode u običnim, a uslovi tranzicije u uglastim zagradama.

Aktivnosti koje objekat izvodi u vremenu predstavljaju se tankim pravougaonicima, koji prekrivaju vremensku osu u vertikalnom pravcu i u veličini koja odgovara vremenu njegovog postojanja.

Dijagram sekvenci može da ima i rekurzivne poruke. To su poruke koje objekat upućuje sam sebi. Rekurzija je prikazana strelicom koja polazi iz objekta i završava se u objektu. Drugim riječima može se prikazati ugnježđavanje (izazvano rekurzijom, pozivom samog sebe) zbog povratnog poziva (*call back*) od pozvanog objekta.



Slika 3.31.: Ugnježđene i rekurzivne poruke

Dakle, objekti međusobno komuniciraju slanjem *poruka*. Poruke se predstavljaju strelicama, koje polaze od jednog objekta, a završavaju na drugom. Poruka (Message) je definisana nazivom i

parametrima. Parametrima se definiše jedan ili više argumenata. Poruka je opisana na sledeći način:

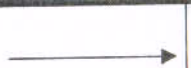
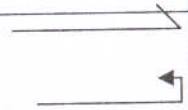
Naziv poruke (lista parametara)

Uz naziv poruke može se navesti i broj kojim se definiše redosled izvođenja poruka. Lista parametara je opciona i male zagrade se obavezno navode bez obzira na listu parametara.

3.Definisanje indikatora sinhronizacije
Indikatorom sinhronizacije definiše se tip poruke.

Različiti oblici sinhronizacije:

Tabela 3.2:

Oznaka	Opis
	Obična(Simple) → kod običnih poruka nije bitan tip poruke i ne specificira se.
	Sinhrone poruke (Synchronous) → sinhrona poruka zahteva odgovor, koji se šalje sa odredišta po prijemu i obradi, blokiraju izvršenje dok odgovor ne stigne.
	Asinhrona poruka (Asynchronous) → asinhrone poruke ne blokiraju izvršenje kada se šalju i koriste se kod konkurentnog programiranja.
	Prepreka (Balking)→ ovim tipom poruke se definiše prepreka ili greška.

Ukratko o sekvencijalnom dijagramu: Polazimo od tekstualong opisa slučajeva upotrebe i konceptualnog dijagrama, zatim koncepti konceptualnog diagrama su u sekvencijalnom dijagramu objekti. Iz tekstualnog opisa slučajeva upotrebe definišu se događaji i operacije za objekte. Dijagram sekvenci može poslužiti da se operacije pridruže klasama u dijagramu klasa.

3.3.2.2.3.Definisanje ugovora o izvršenju operacije

Aktivnost "Definisanje ugovora o izvršenju operacije" daje opis logike operacije koje predstavljaju odziv sistema na specificirane događaje.

Sekvencijalni dijagram sadrži operacije sistema, **gde se opisuje ponašanje operacija i rezultati njihove primene?**

U slučajevima upotrebe se opisuju događaji koji se pojavljuju u sistemu.
Ukoliko su operacije kompleksne ili slučajevi upotrebe nisu detaljni, opis ponašanja operacija može biti nezadovoljavajući (npr. može biti opširan i nekompletan).

Ugovor o izvršenju operacija sistema koje su komplikovane ili nedetaljne daćemo u funkciji sledećih koncepata:

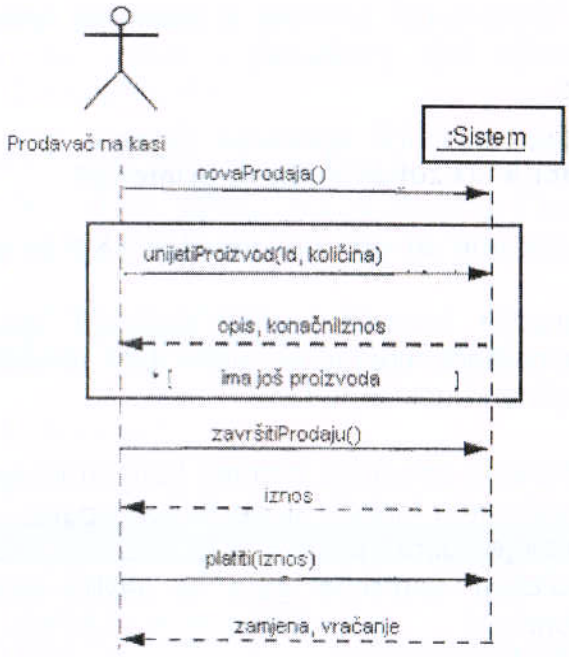
- Zaglavlje operacije
- Slučajevi upotrebe gdje se javlja ova operacija (opcionalno)
- Preduslovi operacije
- Postconditions operacije

Postcondition jedne operacije opisuje promene prouzrokovane ovom operacijom na objektima iz modela domena.
Ove promene se mogu sintetizovati na sledeći način:

- Instance koje su kreirane i destruirane ovom operacijom
- Atributi koji su modificirani ovom operacijom
- Asocijacije koje su kreirane i koje su destruirane primenom ove operacije

Da bismo opisali *postconditions* koristimo se modelom domena koji smo prethodno konstruirali.

*Primer ugovora:*Ako imamo sledeći sekvencijalni dijagram sistema:



Slika 3.32.

Ugovori koji se mogu pridružiti ovim operacijama su:

Ugovor za: NovaProdaja

- **Operacija**: NovaProdaja ()
- **Slučaj upotrebe**: Procesirati prodaju
- **Preduslov**: Sistem funkcioniše
- **Postcondition**:
  - Kreirala se instanca P Prodaja (*kreiranje instance*)
  - P se pridružuje sa TPV (*kreiranje asocijacije*)
  - P se inicializira sa atributima datum i vreme (*modificiranje atributa*)

Ugovor za: unjetiProizvod

- **Operacija**: unjetiProizvod(identif:Id, količina)
- **Slučaj upotrebe**: Procesirati prodaju
- **Preduslov**: U fazi smo procesiranja prodaje P
- **Postcondition**:
  - Kreirala se instanca plp Prodaja-LinijaProizvod (*kreiranje instance*)
  - plp se pridružuje sa P (*kreiranje asocijacije*)

- Atributu količina od plp je dodeljena vrednost (*modificiranje atributa*)
- plp se pridružuje SpecifikacijiProizvoda čiji je identifikator Id (*kreiranje asocijacije*)

Ugovor za: završitiProdaju

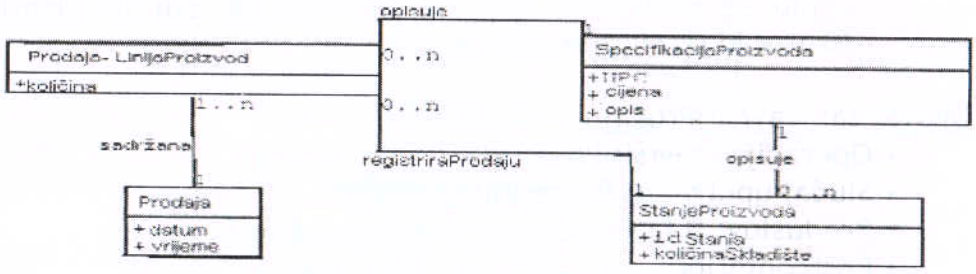
- **Operacija**: završitiProdaju()
- **Slučaj upotrebe**: Procesirati prodaju
- **Preduslov**: U fazi smo procesiranja prodaje P
- **Postcondition**:
  - Atributu kompletna od P se dodeljuje vrednost *true* (*modificiranje atributa*)
  - Za svaki kupljeni proizvod, atribut stanje se adekvatno aktuelizirao (ostatak, nekupljena količina)

Na temelju detalja iz ugovora poboljšavamo model domena uvodeći neke promene.

U slučaju završitiProdaju uvodimo sledeće promene:

1. Dodajemo atribut *kompletna* tipa boolean klasi Prodaja
2. Dodajemo klasu StanjeProizvoda sa atributima: idStanja i količinaSkladište, koja registruje stanje.

Sledi prikaz promene:



Slika 3.33

Ugovor za: Platiti

- **Operacija**: Platiti(iznos)
- **Slučaj upotrebe**: Procesirati prodaju
- **Preduslov**: U fazi smo procesiranja prodaje P
- **Postcondition**:
  - Kreirala se instanca P Prodaja (*kreiranje instance*)
  - Inicializiran je atribut količina od P sa vrednošću parametra *količina* (*modificiranje atributa*)
  - P se povezuje sa v (*kreiranje asocijacije*)
  - P se povezuje sa TPV prodavnice u kojoj je prodato (*kreiranje asocijacije*)

Značajne činjenice za ugovore su:

- Postconditions opisuju **ŠTA** (koje promene su se dogodile u domenu objekata i njihovom stanju), a ne **KAKO** se događaju te promene

- Ugovori slede model «crne kutije»: vidimo stanje objekta pre izvršenja operacije i posle, ali ne tokom izvršenja.
Postcondition opisuje promene stanja objekta *pre* i *posle* izvršenja operacije.
- Kreiramo ugovore za one operacije koje su komplikovane, detaljne ili kritične. Nije potrebno kreirati ugovore za *sve* operacije sistema.
U procesu razvoja softvera nije potrebno kreirati ugovor za svaku operaciju sistema, kao što je to učinjeno u prethodnom primeru.

• Moguće je da neki od detalja *postconditions* operacija sistema ne uvidimo pre faze dizajna.

• U opisu postcondition operacije ne smemo zaboraviti navesti asocijacije koje se kreiraju kao posledica primene ove operacije

• Često ugovori doprinose obogaćenju modela domena (npr. otkriti nove atribute i asocijacije). U prethodnom primeru to može biti atribut *kompletna* klase Prodaja.

Vodič za ugovore:

1. Prepoznati operacije sistema na osnovu sekvecijalnog dijagrama sistema

2. Kreirati ugovor za one operacije koje su komplikovane ili zahtevaju detaljan opis ili nisu dovoljno precizirane u slučaju upotrebe

3. Opsati postcondition ugovora operacije:
   - Instance koje su kreirane i koje su destruirane operacijom
   - Atributi koji su modificirani operacijom
   - Asocijacije koje su kreirane i koje su destruirane primenom operacije

Semi-formalizacija ugovora:

• (op. platiti): Inicializirao se atribut količina od p sa vrednosti parametra količina
• (op: unjetiProizvod): vlp se povezuje sa instancom SpecifikacijaProizvoda (*kreiranje asocijacija*)

Jezik specifikacije zahteva OCL omogućava opis na precizan način za postconditions i ograničenja.

Primer:
System::Platiti(iznos:količinaNovca)
Pred: ....

Post: P.iznos=količinaNovca

...

3.3.2.3. Objektno-orijentisani dizajn

Proces "Objektno orjentisani dizajn" treba da omogući definisanje logike softverskih objekata (koji imaju atribute i operacije), koji će se u sledećoj fazi implementirati pomoću objektno orjentisanih programskih jezika.

Kako smo u prethodnoj fazi identifikovali poslovne procese i uloge u ovoj fazi je potrebno odrediti kako se procesi izvršavaju tj. treba da damo dogovor na pitanje ko šta radi i kakva je saradnja. Ovo zahteva dodeljivanje poslova i odgovornosti (responsibility) različitim softverskim objektima za izradu buduće aplikacije. Ovako definisani softverski objekti sarađuju međusobno po istoj analogiji kao i saradnja ljudi. Tokom dizajna definišu se detalji. Dodaju se nove klase koje ne predstavljaju ključne apstrakcije već su deo implementacije mehanizama. Dodaju se novi atributi i operacije potrebne za realizaciju mehanizama i ključnih apstrakcija. Dodaju se relacije između klasa potrebne za implementaciju.

Dakle, u OOD se precizno specificiraju:

- klase koje postoje u sistemu i definisanje njihovih međusobnih odnosa;
- identifikovanje svih mehanizama koji se koriste u interakciji objekata;
- definisanje gde se u programima deklarišu pojedine klase i objekti.

Ovaj proces se definiše preko sledećih aktivnosti:

- Izrada dijagrama saradnje (kolaboracijskih dijagrama)
- Kreiranje potpunih dijagrama klasa
- Izrada dijagrama stanja
- Definisanje paketa, saradnje mustri i aplikativnih kostura

Aktivnost "Izrada dijagrama saradnje" koristi se za detaljnu specifikaciju svake operacije sistema. OOD daje specifikaciju logike softvera koji će ispuniti funkcionalne zahteve bazirane na dekompoziciji po klasama ili objektima. Korištenjem dijagrama saradnje osnovno je izvršiti lociranje odgovornosti na objekte i odrediti kako oni deluju preko poruka i to prikazati u grafičkom ili mrežnom formatu.

Kolaboracijski dijagram ili dijagram saradnje (Collaboration diagram)

U prethodnoj fazi (analiza) smo definisali zahteve izradom dijagrama slučajeva upotrebe, koji opisuju spoljni statički pogled na sistem, gde se na uopšten način opisuje korištenje sistema ili njegovih delova.

- sekvencijalni dijagram → tok kontrole po vremenskom redosledu

U dizajnu definišemo unutrašnji dinamički pogled dijagrama slučajeva upotrebe u vidu *interakcije među objektima.*

- kolaboracijski dijagram → tok kontrole po organizaciji

Kolaboracije predstavljaju implementaciju slučajeva upotrebe i obuhvataju skup klasa koje sarađuju i načine njihove interakcije definisane preko *scenarija.*

Sekvencijalni dijagram i kolaboracijski dijagram su izvedeni iz istih informacija u UML-ovom meta-modelu, tj. oni su EKVIVALENTNI DIJAGRAMI, ali semantika dijagrama saradnje je bogatija.

Jedan dijagram se može prevesti u drugi, bez bilo kakvog gubitka informacije, ali ***ne predstavljaju iste informacije.***

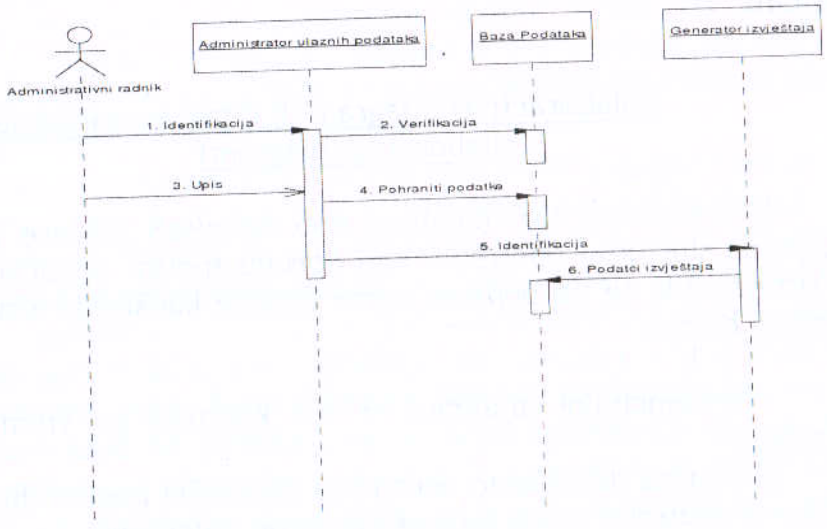
Za modelovanje tokova kontrole po *organizaciji* koristi se dijagram saradnje. Modelovanje toka kontrole po organizaciji naglašava strukturne odnose između instanci u interakciji, između kojih se rezmenjuju poruke. Dijagrami saradnje su znatno bolji za predstavljanje složenih interakcija i grananja višestrukih konkurentnih tokova kontrole.

Sekvencijalni dijagram i kolaboracijski dijagram imaju sličnu strukturu, ali kolaboracijski dijagram osim objekata i poruka

prikazuje i veze, odnosno interakciju između skupa objekata koji se predstavljaju kao čvorovi jednog grafa.

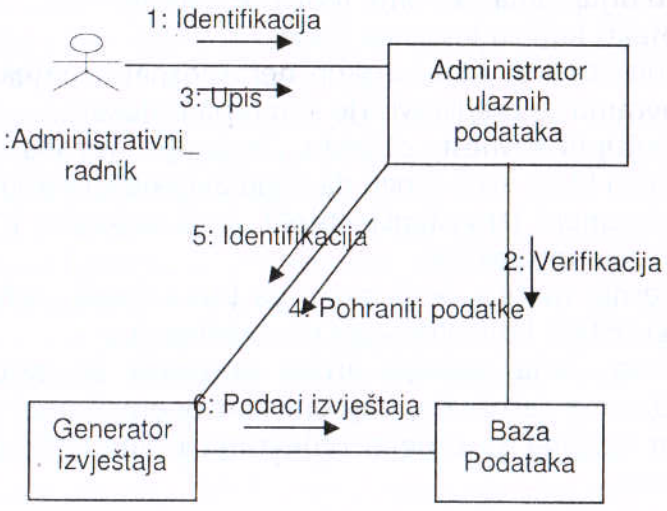
FORMIRANJE DIJAGRAMA SARADNJE

Ako krenemo od sledećeg sekvencijalnog dijagrama, Slika 3.34:



1. Objekti koji učestvuju u interakciji postavljaju se u uglove dijagrama.
2. Definisanje veza između objekata (lukovi dijagrama).
3. Veze se opisuju porukama koje objekti šalju i primaju.

Ovo daje jasnu asocijaciju na tok kontrole u kontekstu strukturne organizacije objekata koji sarađuju. Poruke koje objekti razmenjuju ostvaruju očekivano ponašanje i određenu funkcionalnost sistema.



Slika 3.35.: Kolaboracijski dijagram

Pre nego što se pristupi izradi dijagrama saradnje potrebno je definisati konceptualni model i realne slučajeve upotrebe. Slučajevi upotrebe sugerišu događaje koji su prikazani u sekvencijalnim dijagramima.

U okviru izrade dijagrama saradnje definišu se sledeće aktivnosti:

- Kompletiranje objekata saradnje,
- Definisanje relacija između objekata saradnje,
- Definisanje indikatora sinhronizacije,
- Definisanje indikatora vidljivosti.

KOMPLETIRANJE OBJEKATA SARADNJE

Objekti koji učestvuju u saradnji ili su konkretni objekti (osobe) ili prototipovi. Kod saradnje objekti predstavljaju prototipske stvari sa određenim ulogama, a ne tačno određene objekte u realnom svetu. Uloga objekata specificira stepen uticaja objekta na tok kontrole. Objekat može da bude kontroler (Controller) ili kolaborator (Collaborator). Saradnja je skup objekata povezana u odgovarajući kontekst. Koristi se za prikaz strukture uloge objekata i njihovih veza.

U okviru dijagrama saradnje mogu se definisati:
- objekti koji su instanca klase
- multiobjekti, koji su skup neraspoznatih objekata, a u dijagramu se predstavljaju kao hrpa objekata,
- multiplikativnost objekta, koja je vezana za broj instanci klase koje mogu da budu aktivne istovremeno,
- privremeni (transient) objekt, koji se kreira i uništava za vreme interakcije,
- učesnik (acter), koji može da bude čovek, software ili drugi sistem koji učestvuje u modeliranju,
- klasa, koja opisuje grupu objekata sa zajedničkim osobinama (atributi), zajedničkim operacijama, zajedničkim vezama sa drugim objektima i zajedničkom semantikom.

U kontekstu interakcije mogu da se nađu instance klasa, komponenata, čvorova i slučajeva upotrebe. Iako apstraktne klase i interfejsi, po definiciji, ne moraju da imaju direktne instance, njihove instance mogu da se nađu u interakciji. Takve instance ne predstavljaju direktne instance apstraktne klase neke konkretne klase koja realizuje taj interfejs.
Statički aspekti interakcije mogu da se predstave preko dijagrama objekta, pripremajući podlogu za interakcije specificiranjem svih objekata koji sarađuju. Interakcija ide i korak dalje, uvodeći dinamičku sekvencu poruka koje je moguće prenositi preko veza između ovih objekata.

Dijagram saradnje može se definisati u dva oblika:
- do nivoa specifikacije,
- do nivoa pojave.

Do *nivoa specifikacije* definišu se uloge klasa objekta i veza između njih. Saradnja se ne definiše u kontekstu klasa i veza između njih, već u kontekstu uloga koje posmatrane klase imaju, kao i njihovih međusobnih veza. Ovde je uloga specificirana kao projekcija klase, tj. jedan njen pogled koji je značajan za opis saradnje.

Na *nivou pojave* navode se objekti neophodni za realizaciju operacije, klase ili slučaja upotrebe i veza između posmatranih

objekata, kao i poruka koje objekti međusobno razmenjuju radi izvršavanja posmatrane operacije ili slučaja upotrebe.
Specificiraju se osobine koje pojave klasa i veze između njih, moraju imati kako bi mogle da učestvuju u interakciji. Jedan objekat može da se pojavi u više dijagrama, a i više puta u jednom dijagramu sa različitim ulogama.

Ako je dijagram saradnje definisan za realizaciju *operacije*, tada se daju i definicije parametara, atributa i drugih objekata koji se pozivaju iz date operacije i koji su neophodni za implementaciju operacije. Poruka između objekata predstavlja jedan korak u metodi kojom se implementira posmatrana operacija.

Ako se dijagram saradnje definiše za *dijagram klase*, onda se posmatra opis svih operacija jedne klase i tada se opisi saradnje skrate tako da se dobija opis saradnje kojim se predstavlja implementacija jedne klase.

Definisanje relacija između objekata saradnje

Relacija između objekata saradnje definisana je pomoću interakcije. Interakcija opisuje način na koji objekti u sistemu međusobno komuniciraju radi ostvarenja očekivanog ponašanja i izvršavanja odgovarajućeg zadatka. Objekti međusobno razmenjuju *poruke*(strelice koje polaze iz jednog objekta a završavaju na drugom). Kada objekat primi poruku izvršava određenu operaciju, odnosno metodu koja je implementacija određene operacije.

U dijagramu saradnje ne postoji vremenska dimenzija pa se sekvence poruka predstavljaju sekvencijalnim dijagramima.

Dijagram klasa (Class diagram)

Dijagram klasa (Class Diagram) prikazuje skup klasa, interfejsa i saradnji i njihovih relacija. Dijagram klasa je statička struktura klasa u sistemu koje između sebe uspostavljaju relacije tipa asocijacija (veza sa svakom drugom), zavisnosti (jedna klasa zavisi/koristi se od druge klase), specijalizacije (jedna klasa je specijalizacija druge klase), i paketa (grupisanje u jednu jedinicu tj. paket).

Dijagram klasa u stvari nastaje kao posledica poboljšavanja i detaljisanja konceptualnog modela, dodajući konceptima atribute i operacije koje ih karakterišu.

Primer dijagrama klasa: Aplikacija koja predstavlja sajt za kupovinu. Sistem funkcioniše kao posrednik između kupaca (koji posećuju Web sajt radi kupovine) i dobavljača. Sistem čine dva odvojena sajta:

- **Front office**: sajt koji je dostupan svim korisnicima; omogućava pregledanje kataloga proizvoda i njihovo naručivanje.

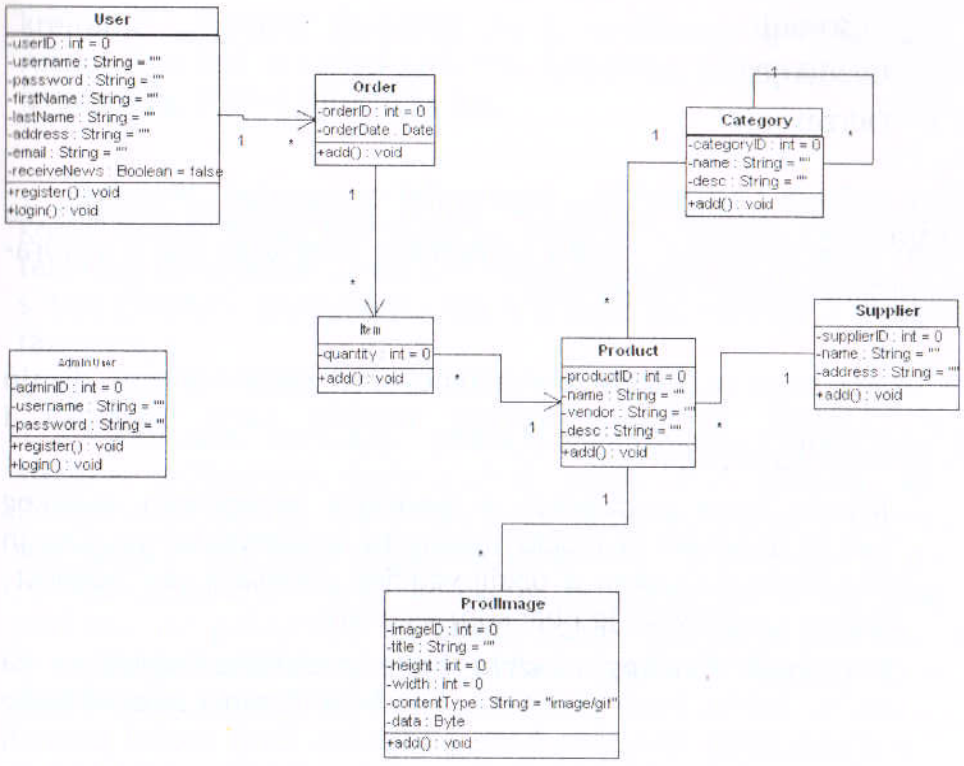
- **Back office**: sajt koji je dostupan samo administratorima sistema; omogućava izvršavanje administrativnih funkcija.

Front office funkcije

- Registracija i autentifikacija korisnika
- Pregled ponuđenih proizvoda, organizovanih hijerarhijski u kategorije
- Izbor proizvoda za kupovinu i njihovo naručivanje ("shopping cart / potrošačka korpa" sistem)

Back office funkcije

- Registracija i autentifikacija korisnika
- Ažuriranje baze dobavljača, proizvoda i kategorija



Slika 3.36.: Dijagram klasa

3.3.2.4. Implementacija

Proces "Implementacija" koristi objektno orijentisano programiranje tj. jezike tipa C++, JAVA, VB i dr. za dizajniranje komponenti. U okviru ove faze pristupa se izradi baze podataka, aplikacije kao i prevođenju i testiranju. Ovaj proces zahteva izabran SUBP (sistem za upravljanje bazama podataka) i odgovarajući programski jezik u kojem će biti realizovana aplikacija.

Implementacija podrazumeva programiranje koda u odabranom programskom jeziku. Najveći dio posla je definiranje tijela operacija (funkcija).

Ovaj proces se definiše preko sledećih aktivnosti:

- Izrada aplikacije,

- Definisanje tehnologije, aplikativne i mrežne arhitekture,
- Testiranje,
- Uvođenje,
- Održavanje.

Aktivnost "Izrada aplikacije" sadrži podaktivnosti vezane za izradu baze podataka, izradu korisničkog interfejsa kao i mapiranje, programiranje i prevođenje.

"Izrada baze podataka" podrazumeva tri podaktivnosti: dizajn baze podatka, kreiranje fizičkog modela baze podataka i generisanje Baze podataka.

Dizajn baze podataka je postupak prevođenja logičkog modela podataka u fizički dizajn za implementaciju. Dizajn baze podataka uzima u obzir veličine podataka, bezbednost, zahteve za arhiviranje i backup/recovery.

Kreiranje fizičkog modela baze podataka koristi se za analizu fizičke baze podataka, kao i za dizajn i razumevanje kompleksnosti relacione baze podataka. Ovaj model prevodi perzistentni logički (objektni) model u fizički model koji je orijentisan relacionim bazama podataka. Fizički model baze podataka koristi se za kreiranje šeme baze podataka.

Generisanje baze podataka izvodi se korištenjem jezika za definisanje podataka -Data Definition Language (DLL) za relacione baze podataka ili Program Specification Block (PSB) i Database Definition (DBD) za IMS, generisanje izvršivih baza podataka.

"Izrada korisničkog interfejsa" treba da obezbedi konzistentan, fleksibilan interfejs koji ispunjava korisničke zahteve i organizacione standarde. Kako je korisnički interfejs "aplikacija" za krajnjeg korisnika, veoma je važno da interfejs bude lak za korištenje, da bude fleksibilan i da reflektuje stil rada korisnika.

"Mapiranje, programiranje i prevođenje" je proces razvoja specifičnih logičkih izraza unutar modula koji će omogućiti

kreiranje programa spremnih za kompajliranje generatorom ili ručno. Ovaj kod se koristi za izradu aplikacija, konverziju podataka, izradu programskih interfejsa itd.

Aktivnost "Definisanje tehnologije, aplikativne i mrežne arhitekture" podrazumeva pre svega primenu kompozitnih i generativnih tehnologija sa jedne strane i primenu višeslojne arhitekture i u vezi s tim primenu implementacijskih dijagrama razvoja i dijagrama razmeštaja.

Aktivnost "Testiranje" podrazumeva izolovano i integralno testiranje realizovanih funkcionalnih tačaka i/ili njihovih delova. Potrebno je pokriti sve (bitne) regularne slučajeve i izuzetke i omogućiti ocenjivanje valjanosti programa, tj. testiraju se performanse programa i vrše korekcije programa da bi se njegove performanse prilagodile korisniku. Dakle, testiranje podrazumeva ocenjivanje karakteristika programa i njegovo revidiranje da bi se dostigli postavljeni ciljevi.

Aktivnost "Uvođenje" izvodi se počev od instaliranja preko vrednovanja softvera do izrade korisničkih uputstava i izrade plana obuke.

Aktivnost "Održavanje" je proces definisanja zahteva za promene na aplikaciji kada je ona već implementirana. Aktivnost sadrži praćenje rada softvera, ispravljanje grešaka, poboljšanje sistema i dodavanje novih funkcija i izmenu hardvera i softvera.

Aktivnost " Kreiranje potpunih dijagrama klasa" definiše potpun dijagram klasa sa detaljno specificiranim atributima, asocijacijama i dodatim operacijama i njihovim opisom. Ovo je najvažniji i najviše korišćeni UML dijagram i predstavlja statičku strukturu sistema. Dijagram klasa u obliku konceptualnog dijagrama se koristi u OOA gde se definiše *šta* sistem treba da radi i u OOD gde se definiše *kako* sistem radi.

Takođe dijagram klasa treba da odgovori na pitanje kako su objekti povezani i šta su to operacije klase. Odgovor na ovo pitanje daje dijagram saradnje koji sugeriše neophodne veze između objekata i operacije koja svaka klasa mora imati. Nasuprot konceptualnom modelu dijagram klasa ne ilustruje koncept realnog sveta, već opisuje buduće softverske komponente.

Aktivnost "Izrada dijagrama stanja" vezana je za samo jedan objekat i određenu operaciju unutar njega, za određenu klasu i na taj se način opisuju složene operacije klase. Ovom aktivnošću prikazuje se sekvenca stanja kroz koje objekat prolazi tokom vremena a kao reakcija na spoljne ili unutrašnje pobude. Dijagrami stanja se koriste i za opis logike jednog slučaja upotrebe.

Aktivnost "Definisanje paketa, saradnje mustri i aplikativnih kostura" vezana je za definisanje: paketa koji se koriste da bi organizovali elemente za modeliranje u veće celine kojima će se manipulisati kao sa nekom grupom. Saradnja predstavlja konceptualnu skupinu elemenata i njihove interakcije, mustra predstavlja neko opšte rešenje za opšti problem i aplikativni kostur koji ocrtava celokupnu arhitekturu zasnovanu na složenim mustrama kojima se definiše neka opšta aplikacija u posmatranoj oblasti.

Objektno orijentisani pristup povezuje podatke i procese u obliku klasa objekata što omogućuje apstrakciju procesa i podataka. Ovakav način rada omogućuje da implementacija počne od bilo koje klase u modelu, a zatim se model gradi deo po deo. Povezujući podatke i operacije koje koriste te podatke u obliku klase dolazi do izolacije tog dela od ostalog sistema tako da promena unutar same klase ne utiče na ostale klase u sistemu. Ovakva nezavisnost klasa omogućuje da se one vrlo lako mogu koristiti u više različitih sistema kao i da se svojstva klasa mogu nasleđivati.

Literatura:

1. Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin, 2001.
2. B.Jošanov,P.Tumbas, *Softverski inženjering*, VPŠ, Novi Sad, 2002.
3. Milićev D., *Objektno orijentisano modelovanje na jeziku UML*, Mikroknjiga, 2001.
4. Stanojević, D.Surla, *UML-Uvod u objedinjeni jezik modeliranja*, "Mala knjiga",Novi Sad,1999.
5. www.fit.ba /predavanja o UML-u u e-formatu/

4. REDIZAJN SOFTVERA
(Principle of Software product re-design)

Software Process Improvement (SPI), proces poboljšanja softvera, tradicionalno je fokusiran na pitanje kako povećati mogućnosti poboljšanja softvera putem sazrevanja i upoređivanja *benchmark*-a tog softverskog procesa.

'The Capability Maturity Model from the Software Engineering Institute' (CMM) je veoma konkretna inicijativa takve vrste. CMM je usmeren ka *incremental improvement* postojećeg softverskog procesa. CMM-ov najviši nivo, Optimizacija (nivo 5), karakteriše organizacije čiji su softverski procesi inkrementalno poboljšani i proćišćeni kroz nadgledanje, merenje i refleksivne analize dobro definisanih i dobro vođenih procesa. CMM ne pruža specifična uputstva kako poboljšati specifični razvoj softvera ili softverski korištenih procesa, ali pruža uputstva za to kako redizajnirati postojeće procese ili kako dizajnirati nove. Pored toga, nema nivoa koji je kompletiran i koji može odrediti kako značajno optimizirati softver kako bi postigli 10x poboljšanje u produktivnosti ili kvalitetu kroz *radical transformation*.

Da li su radikalne transformacije softverskih procesa isto što i inkrementalna evolutivna poboljšanja? Verovatno ne, doduše, jasno je da su oni usmereni ka jednostavnim dimenzijama ili merama koje karakterišu traženu skalu opsega izmene procesa. Slično tome, Software Process Redesign (SPR), proces redizajna softvera, je suštinski deo istraživanja kako bi se odredilo da li i na koji način softverski proces može voditi prema značajnom napredovanju u procesu efikasnosti i efektivnosti.

Studija prezentirana u ovom poglavlju opisuje kako koncepti, tehnike i alati procesa modeliranja, analize i simulacije softvera, mogu biti korišteni u podršci SPR studija. Tri istraživačka pitanja koja se postavljaju i razrađuju te teme mogu biti identifkovani kao sledeća:

- Šta je proces redizajna softvera?
- Kako modeliranje, analize i simulacija podržavaju redizajniranje softvera?
- Šta je potrebno postići kako bi se približili zahtevanom znanju za značajno i neprekidno poboljšanje softverskih procesa?

4.1. Šta je proces redizajna softvera?

SPR je sa identifikacijom, aplikacijom i redefiniranjem važan u novim načinima za značajno poboljšanje i transformisanje softverskih procesa. Softverski procesi uključuju ne samo softverski razvoj, oni takođe uključuju i dobiti softverskih sistema, korištenje i evoluciju. Proces redizajna heuristika, kao i izvori materijala iz kojih su dobijene, postavljen je kao glavni izvor znanja za SPR. On može biti nezavisan od domena aplikacija pa je zbog toga pogodan za veliki set procesa. Alternativno, proces redizajna heuristika može biti specifikacija domena pa stoga i primenljiv za specifikaciju procesa u posebnim postavkama.

U ispitivanju kako su SPR predstavljene, bitne su okolnosti u kojima različiti tipovi heuristika ili iskustava su ili najviše ili najmanje efikasni. SPR heuristike su interesantne, kao i tehnike određivanja transformacije organizacionih postavki u kojem su one predstavljene. Izvori materijala predstavljaju on-line narativne *report*-e iz formalnih analiza, primere, veoma dobre vežbe, izveštaje temeljene na iskustvu i naučene lekcije za pitanje kako značajno poboljšati vremenski ciklus, prevenciju od grešaka i cenu efikasnosti različitih tipova softvera/biznis procesa. Sve više, ti materijali se mogu naći na Internetu kao web publikacije ili hypertext dokumenti. Na primer, mogu se koristiti sva sredstva za pretragu na Webu kako za globalnu tako i za pretragu informacija. Ipak, može se naći mali deo za pretragu '*software process redesign*', tj. pretraga za 'proces redizajna'. Mogu se naći *case* studije, izveštaji temeljeni na iskustvu, vežbe ili naučene lekcije kao narativna dokumenta postavljena na akademskim, neprofitabilnim i komercijalnim stranicama. Jasno je da kvalitet 'znanja' i rezultati

pretrage iz izvora na *net*-u mogu biti više promenljivi nego oni što su postavljeni u studijama sistema istraživanja, ali i u pretraživanju za SPR heuristike, kada su potencijalni važni izvori materijala pronađeni, hyper linkovi mogu biti korišteni za određivanje veze između materijala i trenutnih heuristika. Ponekad, kada je heuristika potencijalni kandidat za korištenje u redizajniranju softverskog procesa, njen izvor materijala može biti tražen i ponovo korišten u pomoći pri određivanju njene sličnosti, relevantnosti, putanje i izlaza. Potom, interes u SPR aktivnostima i rastućim izlazima, kao i SPR *case* studije, izveštaji temeljeni na iskustvu, vežbe, naučene lekcije, prigovori i sl. mogu naći svoj put na Webu. Na taj način, razvoj SPR heuristika ili SPR repozitorija znanja trebao bi biti gledan kao područje za daljnja istraživanja, dok njihove aplikacije trebaju biti integrisane u inicijativama neprestanog procesa poboljšanja, više nego teme koje mogu biti potpuno analizirane sa limitirajućim naporom.

4.2. Kako modeliranje, analize i simulacija podržavaju SPR?

Sve je više studija i tehnika koji se odnose na modeliranje, analizu i simulaciju softverskih procesa, a ipak nijedna od proširenih studija nema za primarni fokus predmet SPR. S druge strane, SPR je često implicitni motivacioni faktor u praktičnim primenama modeliranja i simulaciji softverskih procesa. U takvoj situaciji, kako se može postići da modeliranje, analiza i simulacija softverskih procesa direktno podržavaju SPR?

Modeliranje procesa redizajna znanja

Kao što je već rečeno, SPR znanje se često javlja kao heuristika izvedena iz rezultata empirijskih ili teoretskih studija. Ovi rezultati se dalje mogu kodirati kao proizvodna pravila (zakoni) za primenu u *rule-based* ili *pattern-directed* sistemima zaključivanja, ili kao slogovi koji se čuvaju u relacionoj bazi podataka. Ovi mehanizmi se dalje mogu integrisati sa drugim oruđima za procesiranje.

Sa tačke gledišta modeliranja, postoji potreba da se potencijalno modeliraju različite vrste i oblici SPR znanja. Ovo uključuje:

a) proces koji treba redizajnirati u izvornom obliku, kako jeste pre redizajna ('*as-is*');
b) heuristika redizajna (ili transformacije) koju treba primeniti;
c) budući proces koji treba rezultirati iz redizajna ('*to-be*');
d) empirijski izvori (npr. narativne *case* studije) iz kojih će se heuristika izvesti.

Dalje, može se izabrati da se modelira još i e) niz koraka ('*here-to-there*' proces) kroz koje se različite heuristike za redizajniranje primjenjuju da progresivno transformišu '*as-is*' u '*to-be*' izlaz.

Modeliranje procesa identificiranih u (a), (c) i (e) već zalazi u oblast modeliranje procesa i mogućnosti simulacije. Ipak, (b) i (d) predstavljaju izazove koji se nisu prethodno pojavljivali u tehnologijama modeliranja procesa. Dalje, (b) i (d) se moraju međusobno povezati sa modelima procesa (a), (c) i (e) da bi imali najveću vrednost za eksternu validaciju, praćenje koraka procesa i za svrhe inkrementalne evolucije. Konačno, modeliranje softverskih procesa će igrati ulogu u (f) ostvarivanju kontinuirane evolucije i prečišćavanju mreže SPR znanja.

Analiziranje procesa za redizajn

Modeli softverskog procesa mogu biti analizirani na brojne načine. Te analize su generalno i dovoljno dobro usmerene na poboljšanje kvaliteta modela procesa, da mogu otkriti jednostavne greške i propuste koji se podrazumevaju u velikim modelima. Pored svega toga, proces redizajna softvera iznosi dodatne izazove kada analizira modele procesa.

Prvo, bitno je analizirati konzistenciju, kompletnost, pratiti korake procesa i korektnost mnogostrukih, međusobno povezanih procesa modela (npr. '*as-is*', '*here-to-here*', '*to-be*' modeli). Ovo je nešto što je analogno onome što se dešava u projektu razvoja

softvera kada se koriste mnogostruke notacije (npr. za sistemske specifikacije, arhitekture dizajna, kodiranje i testiranje), zbog toga se zahtevaju analize kao i različite notacije softverskih modela.

Drugo, bitno je objasniti za izvore softverskih procesa redizajnirani napor. Npr. jesu li očigledna sredstva u *'as-is'* procesu zamenjena, pomerena, označena ili uklonjena u *'to-be'* procesu? SPR može promeniti tok sredstava kroz proces i mi tu želimo posmatrati i meriti promene na procesu.

Zadnje, jedan pristup za određivanje 'kada heuristike redizajna procesa domena nezavisnosti' može dati rezultate merenja strukture atributa formalne ili interne reprezentacije procesa kao index za selektiranje heuristika procesa redizajna.

Svaki od ovih izazova iziskuje daljnje opisivanje i redefiniranje isto kao i definisanje toga kako oni mogu delovati u pojednastavljenju ili komplikovanju slučaja.

Simulacija procesa pre, za vreme i posle redizajna

Modeli softverskog procesa mogu biti simulirani na veliki broj interesantnih načina koristeći na znanju bazirane, diskretne entitete ili sistem dinamičkih sistema. Ipak, da li su drugi tipovi sistema potrebni za simuliranje procesa izvedeni od strane korisničkih procesa ili pod njihovom kontrolom? Obzirom na ulogu simulacije u podržavanju procesa redizajna softvera, broj izazova raste. Na primer, koliko performansi poboljšanja realizuju individulane heuristike procesa redizajna? Da li će različiti procesi *workload*-a ili *throughput*-a karakteristika voditi do odgovarajućih varijacija u simulaciji performansi u oba, i *'as-is'* i *'to-be'* modela procesa? Koliko performansi poboljšanja realizuju višestruke heuristike redizajna nakon što se posmatraju sa različitim *workload*-ima ili *throughput*-ima? Može li simulacija pomoći pri određivanju 'da li sve transformacije treba predstaviti kao jednu, ili ih treba realizovati kroz mala inkrementalna poboljšanja redizajna'? Kao takve, simulacije u kontekstu SPR iznose nove i interesantnije probleme koji dalje zahtevaju istraživanje i eksperimentisanje.

Kao što je predloženo ranije, potrebno je simulirati ne samo *'as-is'* i *'to-be'* procese nego i *'here-to-there'* transformacijske procese. Sledeći rezultate BPR istraživačke literature, transformiranje iz *'as-is'* procesa u njegov *'to-be'* duplikat, zahteva promene mišljenja menadžmenta. Proces korisnici koji bi trebali izvršiti i kontrolisati transformacijske procese mogu profitirati i doprineti modeliranju i analize *'as-is'* procesa. Slično tome, korisnici mogu prepoznati mogućnost patologije procesa kada posmatraju grafičke animacije simulacije procesa. Bilo kako, logika simulacije procesa ne sme biti transparentna ili laka za razumevanje, u izrazima koje korisnici procesa mogu lako čitati.

Konvencionalni pristupi simulaciji procesa ne osposobljavaju ljude koji primarno donose softver korištene procese. Umesto toga, druge opcije mogu biti potrebne: jedna gde drugi korisnici procesa mogu međusobno ispitivati novi *'to-be'* proces ili *'here-to-there'* proces preko kompjuterski podržanih procesa. U takvim simulacijama korisničke uloge nisu jednostavno modelovane kao objekti ili proceduralne funkcije; umjesto toga, korisnici koriste njihove vlastite uloge. Ovo je analogno tome kako je simulacija leta korištena kako bi pomogla pilotima u avio saobraćaju.

Pristup i rezultati modeliranja

Identifikujući izazove u prethodnim sekcijama odnosno kako modeliranje, analize i simulacije mogu podržati SPR, ova sekcija prezentuje pristup i inicijalne rezultate napora u svakom od ovih tri područja.

U razvoju modela procesa za SPR, koristima dva alata. Prvi, za reprezentaciju SPR znanja, fomalno i s razlogom, izabran *je the Loom* reprezentacijski sistem. Loom je formirani jezik i okruženje za konstrukciju ontoloških i inteligentnih sistema kojima može biti pristupano preko Weba. Koristeći Loom u reimplementaciji, mogu biti reprezentirane ontologije meta-modela Articular procesa, formalni modeli softverskih (ili business) procesa i klasifikacije heuristika redizajna procesa. Naizmenično, proces znanja može biti analiziran, ispitan i pretraživan dok relevantne alternative za procese mogu biti identifikovane i linkovane prema izvorima materijala na Webu. Ipak, Loom određuje discipline za formalnu reprezentaciju

struktura deklarativnog znanja u granici koncepta (objekti ili uzorci tipova), relacija (link tipovi koji udružuju koncepte) i instanci (koncepti, link, vrednosti atributa). Loom-ov *'deductive classifier'* koristi *forward chaining*, semantičku unifikaciju i objektno-orjentisanu podršku tehnologijama. To omogućava Loom-u da kompajlira deklarativno znanje u reprezentaciju semantičke mreže dizajniranu za efikasniju podršku on-line *deductive query* procesiranja.

Dalje, Loom-ov klasifikator (*classifier*) može biti korišten za procenu i up-date SPR znanja baziranog na novima unesenim SPR slučajevima i formalnom modelovanju. To najpre omogućava da se SPR znanje razvije sa automatizovanom podrškom.

Drugo, za podršku vizuelizacije baze znanja i modela procesa koji su konstruisani, u Loom sistemu je korišten *web browser interface*. *Ontosaurus* je *client-side* alat za pristup Loom serveru koji je napunjen sa jednom ili više baza znanja. To podržava upite prema Loom-u i iznosi Web stranice opisujući nekoliko aspekata baza znanja uključujući i linkove za materijale na Webu. Takođe, nudi jednostavne pogodnosti editovanje sadržaja u baze znanja.

Loom i Ontosaurus su bili korišteni u prototipu sistema baza znanja koji može reprezentovati i dijagnosticirati modele softverskih procesa redizajna heuristika i procena, isto kao i upravljanje hiperlinkovima materijala dostupnih na *net*-u. Sistem primenjuje *Articulator ontology* softverskih i business procesa koji su izraženi kao koncepti, atributi, relacije i vrednosti u Loom-u. Loom pruža okvir semantičkih mreža bazirane na opisnoj logici. Čvorovi (objekti) u Loom reprezentaciji definišu 'koncepte' koji imaju ulogu u specifikaciji njihovih atributa. Ključ opisa logičke reprezentacije je to što je semantika reperezentacionog jezika veoma precizno definisana. Ta preciznost specifikacije čini mogućim za klasifikatora određivanje jednog koncepta *subsumes* i drugog baziranog isključivo na formalnim definicijama dva koncepta. Klasifikator je važan alat za odvijanje ontologija jer može biti korišten za automatizovanje organizacije seta LOOM koncepata u klasifikatornu hijerarhiju ili procenu baziranu isključivo na njihovim definicijama.

Ova mogućnost je naročito važna kako ontologija postaje veća, jer tako će i klasifikator naći pretpostavljene relacije koje bi

ljudi mogli ispustiti, kao i modeliranje greški, koje bi mogle načiniti bazu znanja nekonzistentnom.

Uglavnom, 30 procesa redizajniranih heuristika su identifikovane i klasificirane. 6 taxonomija je takođe identifikovano za grupisanje i organizovanje pristupa materijalima procesa redizajna nađenih na Webu.

Ove taxonomije klasificiraju i identifikuju slučajeve na:

- Generički tip organizacije ili glavne aplikacije za proces redizajna- finansijski, proizvodni, istraživački, razvojno softverski...
- *'as-is'* 'problemi' sa postojećim procesom - off-line informacijsko procesiranje, protok kašnjenja nedostatak deljivosti informacija itd.
- *'to-be'* 'solucije' (ciljevi) neophodni za redizajniran proces- automatski off-line informacioni procesni zadaci, struja protoka, korištenje e-maila i baza podataka za deljenje informacija.
- Korištenje Intraneta, Extraneta ili web baziranih procesa redizajniranih rešenja, izgraditi Intranet portal za informisanje projektnog osoblja, pohranjivanje *version-controlled* softver izgrađenih objekata na Web server, korištenje HTML formi za unos podataka i proces validacije koraka.
- SPR *'how-to'* vodiči ili naučene lekcije- eksplicitne tehnike ili koraci 'kako razumeti i modelirati 'as-is' proces, identificirati alternative redizajna procesa, uključiti proces-korisnike u odabiru ovih alternativa itd.
- SPR heuristike- paralelizirati sekvence obostrano ekskluzivnih zadataka, proširiti *multi-stage* pregled/dozvola petlji, izbalansirati strukturu Project Managementa, prebacivanje provere validnosti kvalitete procesa ili podataka na početak, logički centralizirati informacije da mogu biti deljive u odnosu na rutirane.

Za uzvrat, svaka od ovih taksonomija može biti predstavljena kao hijerarhijsko ugnježđeni indexi Web linkova u odgovarajućim slučajevima. Navigacija kroz ugnježđene indexe koji su organizovani i prezentirani kao *portal-site* je poznata Web

korisnicima. Tipično, svaka taksonomija indeksira 60-120 slučajeva ili više od 200 narativnih izveštaja koji su nađeni na Webu i pročavani. Ovo znači da se neki slučajevi mogu pojaviti u jednoj taksonomiji, ali ne i u drugoj, dok se drugi slučajevi mogu pojaviti u više od jedne, a neki se ne moraju pojaviti ni u jednoj od ovih taksonomija, ako im je dodeljeno da ne moraju posedovati minimum informacija neophodnih za karakterizaciju i modeliranje.

Pristup i rezultati analiza

Prvi izazov u analiziranju procesa za redizajn naglašava tri tipa problema koji se javljaju kod procesiranja.

Prvo, mogu se javiti konzistentni problemi. Ovo označava konflikte u specifikaciji nekoliko vrednosti datog procesa. Npr. tipični konzistentni problem je imati proces (npr. za Softver Dizajn) sa istim nazivom kao i onaj za izlaz. Ovo je nešto što se javlja iznenađujuće često u praksi, verovatno iz razloga jer je *output* često najvidljivija karakteristika procesa.

Drugo, problemi potpunosti pokrivaju nepotpune specifikacije procesa, npr. tipičan primer se javlja kada specificiramo proces bez inputa. Ovakav proces se može smatrati „čudom", jer može proizvesti output bez inputa. Slično ovome, proces u nedostatku outputa predstavlja „crnu rupu", gde proces inputi nestaju bez generisanja bilo kakvog outputa.

Treće, problemi praćenja su uzrokovani netačnom specifikacijom porekla samog modela: svog autora, agenta odgovornog za autorizovanje ili *up-date* i *source* materijala iz kojih je nastao. Posledično ovome, za model procesa koji je konzistentan, potpun i s mogućnošću praćenja, može se reći da je interno ispravan.

Dakle, rešavanje ovih problema model-kontrole je neophodno onda kada su proces modeli formalizirani. Jedan od glavnih razloga što je Loom interesantan kao reprezentativni formalni proces-jezik je njegova mogućnost da predstavi apstraktne uzorke podataka koji su stvarna definicija problema o kojima je diskutovano ranije. Ova mogućnost je korisna u proizvodnji jednostavnih i čitljivih reprezentacija 'model-kontrole' analiza.

Koristeći proces modeliranje reprezentaciju prethodno spomenutu, korisnik opisuje proces model kroz Ontosaurus za procesiranje Loom-om. Jedna od prednosti korištenja Loom-a jeste kada smo jednom definisali činjenicu, Loom automatski aplicira svoj *classifier engin* da pronađe odgovarajuće koncepte i isto aplicira. Ovo je velika prednost jer nije potrebno specificirati algoritam za analizu procesa: umesto toga, proces modeli su automatski analizirani kao novi model koji je specificiran. Kao dodatak, klasifikator obavlja održavanje *„truth"*. Zbog toga, ako je izvršen update modela procesa da popravi problem koji je sistem pronašao, klasifikator će odmah utvrditi da se problem odnosi na proces.

Dakle, klasifikator automatizira aktivnost za akviziciju i update. U cilju obezbeđenja više direktnog interfejsa za sistem dijagnosticiranja analize procesa, Ontosaurus pretraživač je proširen da prikaže dva nova tipa stranica. Prvi prikazuje opis procesa na manje Loom specifičan način (npr. za svrhu reporta). Drugi prikazuje listu svih problema pronađenih u trenutnom proces modelu koji smo uneli.

Kako formalna prezentacija modela softver procesa može biti gledana kao semantička mreža ili direktno primenjen graf moguće je meriti kompleksnost parametara mreže/grafa kao baze za graf transformaciju, simplifikaciju ili optimizaciju. Ovo znači da vrednosti visoko vrednovanih *„process flow chart"* mogu dati vrednost kao što su broj koraka, dužina sekvencijalnog segment procesa, vrednost/stepen paralelizma u protoku proces kontrole i drugih.

Dakle, redizajn heuristika može biti kodiran kao predložak u strukturi proces-prezentacije. Time postaje moguće da se prati heuristika kao predložak orjentisanog interfejsnog pravila ili triggera čiji prethodnik ugovara predložak vrednosti kompleksnosti procesa i čija konsekvenca specificira optimizaciju transformacije koja se aplicira na prezentaciju procesa. Npr. kod analiziranja modela softver procesa ako je sekvenca etapa procesa linearna linija i input i output etapa obostrano ekskluzivni, tada se etape procesa mogu obavljati paralelno. Ovakva transformacija reducira vreme neophodno za izvršenje sekvence. Dakle, analize procesa za SPR se mogu bazirati na merenjima vrednosti formalne prezentacije

modela softver procesa koji je interno tačan. Ovo je slično tome kako kompajler obavlja optimizaciju koda tokom kompilacije nakon semantičke analize dok je prioritet na generisanju koda.

Pristup i rezultati simulacija

Pitanja koja se nameću za simulirane procese preko perfomansi ili korisničkog pre/posle redizajna procesa već se mogu naglasiti kao tehnike i simulacijski alati procesa. Nisu neophodne nikakve značajke unapred. Slično, mogućnosti znanje-baziranih simulacija mogu se iskoristiti da odrede performanse proces i poboljšanja kada se koriste različite redizajnirane heuristike da se kreira alternativni scenario za softver-proces-odluke. Ne manje bitan izazov je kako podržati transformaciju *'as-is'* softver-procesa u *'to-be'* redizajniranu alternativu.

Dakle, novi pristup je neophodan. Ključni zahtev za upravljanje organizacionim transformacijama prema redizajniranom softver procesu je angažman motivacija i ohrabrivanje korisnika procesa. Cilj je obezbediti ovim korisnicima da učestvuju i kontrolišu napore procesa redizajna kao i odabir alternativa redizajna procesa za implementaciju i odlučivanje. Kako direktna upotreba dostupnih simulacijskih paketa mogu predstavljati prepreku mnogim proces korisnicima, druga sredstva za podršku proces upravljanja i proces izmenama su neophodna.

Jedan od pristupa je da se uključi *process* korisnička zajednica u *multi-site* organizaciono okruženje i partnerstvo sa njima u redizajniranju njihovih softver procesa. Takođe, potrebno je razviti, obezbediti i demonstrirati prototip simulatora *wide-area* procesa koji će omogućiti participante proces redizajna sa sredstvima za modeliranje, redizajniranje i pregled procesa koji obuhvataju mnogobrojne vrednosti kojima se pristupa putem Interneta. Sa ovim okruženjem 10 proces redizajn heuristika je moguće primjeniti dok su proces korisnici odabrali 9 za implementaciju. Shodno tome, s vremenom su postigli faktor od 10x u redukciji vremena i redukciji u broju proces koraka između 2-1 i 10-1 u softver korištenim procesima koji su bili redizajnirani. Proces simulator je igrao glavnu ulogu u redizajnu, demonstraciji i izgradnji prototipa ovih procesa. Kako je ovo realizovano?

Primjer simulatora procesa

Prototip procesa je računarski podržana tehnologija za omogućavanje modeliranja softver procesa kako bi bili nezavisni, bez integrisanja alata od starne modeliranih procesa. To obezbeđuje proces-korisnicima mogućnost da interaktivno nadgle-daju i pregledaju proces model krak po korak po svim nivoima modelirane proces dekompozicije, koristeći *GUI* ili *Web browser*. Kreirajući bazno procesno izvršno *run-time* okruženje, uzimajući prototip proces modela i integrirajući alate kao aplikacijske pomoćnike koji manipulišu proces zadacima vezanih manuelno ili automatsko generisanih web/intranet hyperlink URL-ova.

Koristeći prototip proces tehnologija možemo raditi sa proces korisnicima da modeliramo njihov *'as-is'* i *'to-be'* proces. Dakle, modelirani procesi mogu biti interaktivno prebačeni koristeći okruženje Web pretraživača ka rezultujećem proces simulatoru. Proces korisnici neovisno od vremena i lokacije njihovog pristupa proces modelu, mogu obezbediti *feedback*, kao i evaluaciju onoga što su videli u Web baziranom proces simulatoru.

Simulatori su uspešni u pomaganju proces korisnicima da uče o operacionim sekvencama rešavanja zadataka koji čine softver proces. Simulatori leta su već demonstrirali ovaj isti rezultat mnogo puta sa operacijama letenja (piloti). Korištenje proces simulatora može identificirati potencijalne uzorke softver procesa kao i potencijalne performanse. Ova informacija zauzvrat može pomoći da se identifikuju vrednosti parametara za diskretne simulacije istog procesa. Ali, ovo još nije pokušano.

Dakle, diskretni-događaj i sistemi simulacije baza-znanja, zajedno sa simulatorima procesa, čine učenje, deljenje znanja, okruženje za merenje i eksperimentisanje koje može pružiti podršku i ohrabriti proces korisnike kad redizajniraju svoje softver procese.

Znači, ove mogućnosti proces simulacije, zajedno sa drugim organizacionim promenljjivim upravljačkim tehnikama bi trebale

minimizirati rizik neuspeha kod redizajniranja softver procesa korištenih u kompleksnim organizacionim postavkama.

Diskusija o SPR

Pored predstavljanja subjekta SPR-a, objašnjenja šta je to, objašnjenja kako se modelira softverski proces, analizira i simulira njegova sposobnost kao i demonstracija kroz primere kako to sve funkcioniše, ostalo je još puno posla da se uradi. Stoga, svrha ovog dela je da se identifikuju neke od budućih potreba koje, u ovoj investiciji, postaju očite.

Prvo, bilo kakvo postupanje sa *legacy software process* u *real-word* uređenju ili kako pretražujemo opise procesa nađenih na Webu, prihvatanje, formalizacija ili neko drugo modeliranje *'as-is'* procesa je nepogodno. Deo problema najčešće je eksplicitno nedostajanje organizacije *well-defined* i *well-managed* procesa kao startna tačka za SPR pokušaje. Kao posledica, pažnja je često usmerena na fokusiranje samo na kreacije *'to-be'* alternativa, bez utemeljenja na *'as-is'* osnovu. Bez osnove, SPR napori i pokušaji će rasti kao i verovatnoća grešaka. Stoga, javlja se potreba za nove alate i tehnike za rapidno prihvatanje i kodifikaciju *'as-is'* softver procesa prema poboljšanju SPR-a.

Drugo, postoji potreba i za rapidnom generacijom *'to-be'* i *'here-to-there'* procesa i modela. SPR heuristike, isto kao i alati i tehnike za njihovo postizanje i prihvatanje, zahtevaju značajne vrednosti. Oni mogu pomoći bržem, rapidnom proizvodu alternative *'to-be'* procesa. Bilo kako, znanje za 'kako konstruisati' ili 'odrediti transformaciju *'here-to-there'* procesa na način da spoje tehnike menadžmenta i alate proces menadžmenta', je otvoren problem.

Treće, SPR heuristike ili transformacije taksonomija mogu poslužiti kao temelj za razvijanje teoretskog 'okvira' za najbolje predstavljanje SPR znanja. Slično tome, treba postaviti okvir za koji tip koncepta softverskog procesa, linkova i instanci treba predstaviti, modelirati i analizirati za olakšanje SPR-a. Ipak, tu su, takođe, praktične potrebe dizajna i izrade SPR taksonomija za specificiranje domena softver procesa i organizacionih postavki. Od te tačke, nejasno je koje heuristike za redizajniranje softvera

procesi koriste i koje su jednako prihvatljive za dobit softvera, razvoj ili evoluciju. Isto se može reći i za bilo koju drugu kombinaciju ovih tipova softver procesa.

Četvrto, u prethodno rečenom, predstavljeni su softver alati koji podržavaju modeliranje, analize i simulaciju softver procesa za redizajn. Bilo kako, ti alati nisu izgrađeni iz početka kao jedno integrisano okruženje. Stoga, njihove mogućnosti mogu biti demonstrirane za pomoć pri razjašnjavanju šta je moguće, ali i što ta mogućnost ne može biti praktična za *widespread* razvijanje ili produkciju korištenja. Tako da postoji potreba za novim okruženjima koji podržavaju modeliranje, analize i simulacije softver procesa koji mogu redizajnirati životni krug inženjerstva i neprestano poboljšanje iz znanja prihvaćenog sa Weba.

Posljednje, kao najviše postavljeno u istraživačkim studijama u biznis proces redizajnu i na osnovu iskustava iz prve ruke, proces korisnik treba biti uključen u redizajniranje svojih vlastitih procesa. Prema tome, iskušenje u traženju automatskih zahteva u generisanju dizajn alternativa *'to-be'* procesa iz analiza *'as-is'* modela procesa mora biti smanjeno. Ovde je važno razumeti kada je i ako je automatski SPR poželjan i koji je tip organizacije.

Jedan od ciljeva SPR-a bi trebao biti minimizacija grešaka SPR pokušaja. Solucije fokusa su ekskluzivno na tehnološki-vođen ili samo tehnološki pristup SPR-u. Stoga, ostaci izazova za te ekskluzivne izbore puteva tehnologije prema SPR-u efikasno demonstriraju kako takvi pristupi mogu slediti tip organizacije i tip veštine participacije procesa.

Prethodno izneseno ukazuje na tri istraživačka pitanja koja identifikuju i opisuju šta je proces redizajna softvera, kako se modeliranje i simulacija uklapaju u to i kako izgledaju zahtevi za SPR. SPR je predložio, kao tehniku za radikalna ostvarenja, ostvarenja reda veličine ili redukcije u softver proces atributima. SPR se gradi na empirijskim i teoretskim rezultatima u području biznis proces reinženjeringa. Ono se takođe gradi na znanju preuzetom s Web-a. Doduše, kvalitet takvog znanja je više promenljiv. Centralni rezultat znanja dugo sakupljanog je to što SPR mora svoj fokus kombinovati na obe tehnike za promene

organizacije gde će softver procesi biti redizajnirani isto kao identificiranje kako softver inžinjering i proces menadžment alati bazirani na informacionoj tehnologiji i koncepti mogu biti prihvaćeni.

Tehnologije procesa softverskog modeliranja, analize i simulacije mogu biti uspešno korištene u podršci SPR. Obećavajuće načine za iskorištavanje i apliciranje u ovom pogledu, jasno nude, naročito, alati bazirani na znanju, tehnike i koncepti. Bilo kako, izazovi novih procesa modeliranja, analiza i simulacija su takođe identificirani. Ovo ispoljava potrebu za investiranjem u nove alate i tehnike za dobijanje, reprezentaciju i upotrebu novih formi procesa znanja. Znanje, kao što su SPR heuristike, mogu imati glavnu ulogu u rapidnom identifikovanju alternativa procesa redizajna. Tehnike softver proces simulacija mogu zahtevati ljudski vođene, ali kompjuterski podržane simulatore procesa, koji dopuštaju korisnicima posmatranje alternativa procesa redizajna. Mogućnosti procesa modeliranja softvera, analiza i simulacija koje podržavaju SPR aktivnosti mogu biti potrebni za razmeštanje u načinima za angažovanje i olakšavanje potreba korisnika koji dele procese kroz višestruke organizacije uređenja, koristeći mehanizme koji mogu biti razmešteni na net-u.

Predstavljen je dolazak SPR koji upotrebljava Web bazirane alate za softver proces modeliranje, analize i ljudski vođene simulacije. Prvobitna iskustva u korištenju ovih alata, zajedno sa biznis proces reinžinjeringom i promenom menadžment tehnika utelovljuju indiciju objektiva ili red-veličine redukcije u krugu softver procesa i koraka procesa, mogu biti demonstrirani i ostvarljivi u kompleksnom organizacionom uređenju. Ako rezultati kao takvi mogu biti u svim klasama softverskog dobit - procesa, izrada, razvoj, korištenje i evolucija - predstavlja subjekt daljnjeg istraživanja. Slično tome, drugi istraživački problemi su bili identificirani za kako i gde unaprediti u softver proces modeliranju i simulaciji i kako mogu prednjačiti u daljnjim eksperimentalnim studijama i teoretskom razvoju u izvedbama redizajna softver procesa.

5. SOFTVERSKI PROCES (Software process)

Ravoj softvera predstavlja ciklus aktivnosti u razvoju, korišcenju i održavanju softvera. Tokom života, softver prolazi kroz više faza razvoja: od začetka, preko inicijalnog razvoja, produktivnog funkcionisanja, održavanja do povlačenja.

Softverski proces (ili životni ciklus softvera) je skup administrativnih i tehničkih aktivnosti koje se realizuju u toku prikupljanja, razvoja, održavanja i povlačenja softvera [Swebok], odnosno sve aktivnosti koje se sprovode da bi se proizveo neki softverski proizvod i svi rezultati koji se pri tom dobijaju.

Osnovne aktivnosti zajedničke za sve softverske procese su date u Tabeli 5.1.

Tabela 5.1.

Specifikacija softvera	*definicija funkcija i operacija softvera*
Razvoj softvera	*proizvodnja softvera prema specifikaciji*
Validacija softvera	*provera da li softver ispunjava zahteve korisnika*
Evolucija softvera	*softver evoluira shodno zahtevima korisnika*

Različiti softverski procesi sprovode ove aktivnosti na različite načine. Takođe, različite organizacije koriste različite procese da bi proizvele iste softverske proizvode. Neki procesi više odgovaraju nekim tipovima aplikacija. Ako se koristi loš proces proizvest će se manje kvalitetan softverski proizvod.

Postoji veliki broj modela ili paradigmi razvoja softvera. Svaki od njih ima sopstveni skup aktivnosti i način njihovog

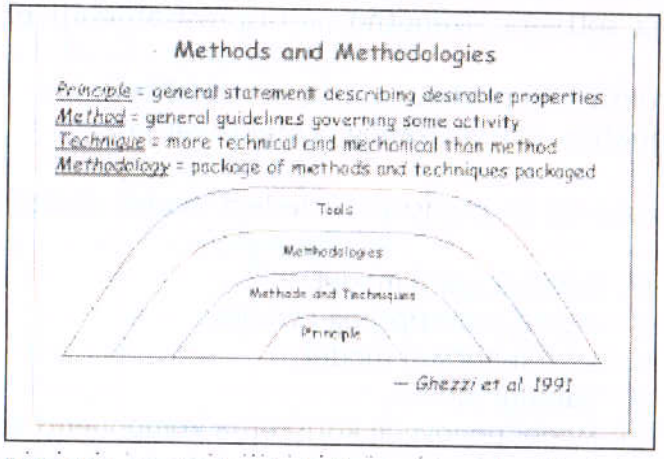
povezivanja. Najbitnije karakteristike softverskog procesa su date u Tabeli 5.2.

Tabela 5.2. Karakteristike softverskog procesa

Razumljivost	*da li je proces eksplicitno definisan i da li je razumljiva njegova definicija*
Vidljivost	*da li se napredovanje procesa može spolja videti*
Prihvatljivost	*da li proces mogu lako prihvatiti inženjeri koji ga sprovode*
Podržanost	*da li je proces podržan CASE alatima*
Pouzdanost	*da li proces štiti korisnika od grešaka*
Robusnost	*da li je proces sposoban da reaguje na neočekivane probleme*
Održavanje	*da li proces može evoluirati da bi reagovao na promene*
Brzina	*koliko se brzo dolazi do softverskog proizvoda*

5.1. Modeli softverskog procesa

Softverski inženjering čine skupovi koraka koji uključuju metode, alate i rocedure.Ti koraci su zasnovani na razvojnim principima i upućuju na modele razvoja softvera u softverskom inženjerstvu. Model razvoja se odabira u zavisnosti od prirode projekta i aplikacije, tehničke orijentacije, ljudi koji ce učestvovati u razvoju, metoda i alata koji će se upotrebljavati pri razvoju, načina kontrole i samih proizvoda koji se zahtevaju, Slika 5.1.



Slika 5.1. Principi, metode i tehnike,metodologija i alati za razvoj softvera

Razvojni principi su opšte važeći osnovni principi koji određuju način izvršenja rada i omogućavaju uopštavanje određenih specifičnosti i zakonitosti objektivne stvarnosti. Oni mogu biti osnovni principi načina izvršenja rada i principi razvoja obzirom na metodološke korake i redosled njihovog izvršenja.

Modeli razvoja se pojavljuju od vremena kada su se projektima razvijali veliki softverski sistemi. Osnovni razlog njihove pojave je bila želja da se obezbedi uopštena šema razvoja softvera, koja bi služila kao osnova u planiranju, organizovanju, snabdevanju, koordinaciji, finansiranju i upravljanju aktivnostima razvoja softvera. Modeli prikazuju različite poglede na proces razvoja softvera.

Uopšteno, modeli su ***apstrakcije*** koje pomažu u procesu razvoja softvera. Model softvera predstavlja komponente razvoja softvera i razvijen je na osnovu ideja konstruktora i njegove predstave šta je važno. Model može predstavljati: **model procesa razvoja, model softvera** ili **model upravljanja softverom**. Cilj kreiranja modela je da se obezbede softverski proizvodi koji odgovaraju zahtevima korisnika.

Prema redoslijedu izvršenja pojedinih faza-modula-etapa-koraka razvoja softvera, trenutno najrasprostranjeniji ***primenjeni principi*** su:

- **principi strukturnog projektovanja sa modalitetima i**
- **principi objektno -orijentisanog projektovanja.**

U nastavku će se predstaviti sledeći modeli životnog ciklusa softvera:

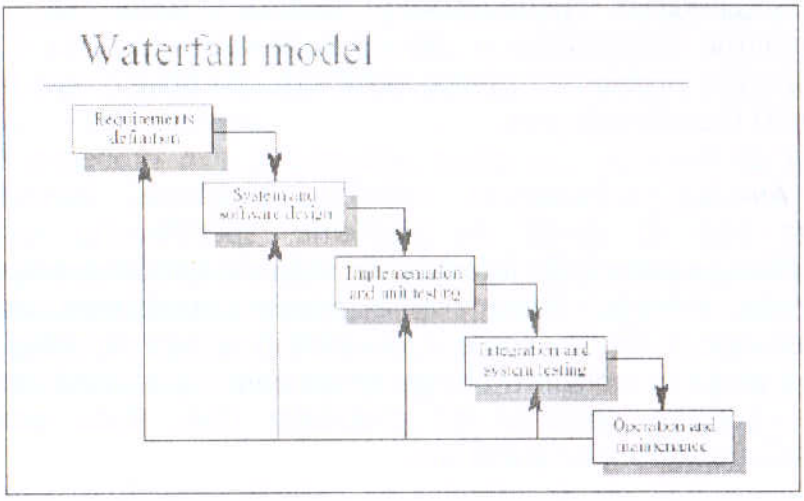
1. Sekvencijalni model,
2. Model prototipskog razvoja,
3. Inkrementalni model,
4. Spiralni model,
5. Model ponovnog korišcenja komponenti softvera,
6. Model razvoja tehnikama četvrte generacije,
7. Kombinovani modeli.

5.1.1. Sekvencijalni model

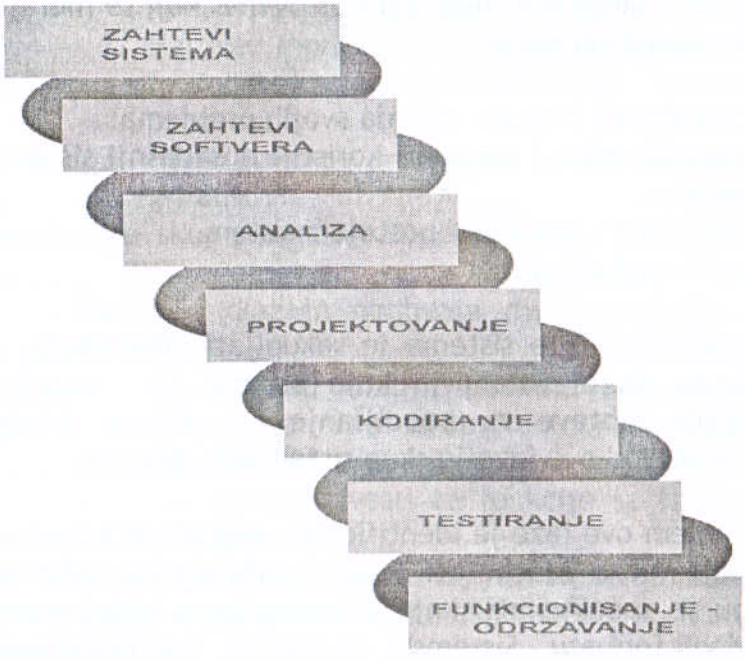
Sekvencijalni model je tradicionalna sistemska metoda za razvoj informacionih sistema, kao i samog softvera. Predstavlja strukturalni okvir koji se sastoji od sekvencijalnih procesa koji se koriste u razvoju informacionog sistema (IS). Model je uveo W.Royce 1970.godine.Ideja je uraditi sve dobro od početka, a u najgorem slučaju će se morati vratiti samo na prethodni korak. Sistemski razvoj projekta daje željene rezultate proporcionalne uloženom trudu.

Tim koji učestvuje u izgradnji IS-a tipično uključuje buduće korisnike, sistemske analitičare, programere i tehničke specijaliste. Korisnici su zaposleni u različitim sektorima i nivoima organizacije. Sistemski analitičari su profesionalci IS-a specializovani za analizu i dizajn IS-a. Programeri su profesionalci IS-a koji modifikuju postojeće programe ili pišu nove da bi se zadovoljila nova potraživanja korisnika. Tehnički specijalisti su eksperti ovog tipa tehnologije, kao što su baze podataka ili telekomunikacije.

Ovaj model je poznat i pod nazivom klasični model i model vodopada (engl.«waterfall»). Model vodopada sledi sledeće korake, Slika 5.2.a i 5.2.b:



Slika 5.2.a. Model vodopada - Waterfall model



Slika 5.2.b. Model vodopada - Waterfall model

Aktivnosti razvoja prema ovom modelu su sledeće:

SPECIFIKACIJA (ISTRAŽIVANJE) SISTEMA: Mora se obaviti preliminarno istraživanje u cilju razumevanja funkcija i ciljeva organizacije, problema i njenog okruženja, da bi se moglo ispravno pristupiti razvoju softvera.

ANALIZA podrazumeva definiciju problema, identifikaciju razloga koji su doveli do problema, specifikaciju rešenja i identifikaciju potrebnih informacija koje će dati zadovoljavajuća softverska rešenja. Razumevanje *bussines* problema uključuje razumevanje i drugih procesa uključenih u njemu, tako da se analiza može kompliciratí, mogu se pojaviti zavisnosti različitog tipa i potrebe povratka na prethodni nivo. Zato analitičari raspolažu različitim *case* alatima.

Znači, radi se o određivanju funkcija koje mora da obavlja softver jednog informacionog sistema, pod uslovima koji se moraju ispuniti prilikom obavljanja ovih funkcija i uslovima koji se moraju ispuniti nakon obavljene funkcije.

Organizacija ima tri moguća rešenja svojih problema:

1. Ne uraditi ništa i nastaviti koristiti dosadašnji sistem, bez promena.
2. Modificirati i poboljšati postojeći sistem.
3. Razviti novi sistem.

Zadatak analitičke faze sistema je sakupljati informacije o postojećem sistemu da bi ustanovili koju od ove tri solucije prihvatiti i utvrditi zahteve za poboljšanje ili zahteve novog softvera. Finalni proizvod ove faze je skup sistemskih zahteva.

Najteži problem ove faze je identificirati specifične informacione zahteve koje mora zadovoljiti sistem. Informacije koje se zahtevaju moraju se specificirati: koje informacije, u kojoj meri, za koga i u kojem formatu. Sistemski analitičari upotrebljavaju puno različitih tehnika da bi otkrili zahteve novih sistema. Ove tehnike uključuju struktuirane i nestruktuirane intervjue sa korisnicima i zapažanja.

Faza sistemske analize daje sledeće informacije:

1. Prednosti i nedostaci postojećeg sistema,

2. Funkcije koje mora imati novi sistem da bi rešio postojeće probleme,
3. Informacije koje zahtevaju korisnici od novog sistema.

Kada se poseduju ove informacije može se otpočeti sa fazom dizajna.

Dok su sistemski analitičari definisali šta *software* mora da radi, u DIZAJNU SISTEMA sistemski dizajneri rešavaju kako će se softver uklopiti u sistem. Potrebno je pronaći strukturu softvera koja će zadovoljiti potrebe i koja se može kreirati sa postojećim resursima.

Faza sistemskog dizajna je tehnički dizajn koji specificira sledeće:

1. Izlaze sistema (*outputs system*), ulaz (*inputs*) i korisnički interface (*user interface*).
2. Hardver (*hardware*), softver (*software*), baze podataka (*database*), telekomunikaciju, ljudske resurse i procedure.
3. Način kako su prethodno navedene komponente integrirane.

Rezultat je skup sistemskih specifikacija.

Sistemski diazajn obuhvata dva najvažnija aspekta novog sistema:

1. ***Logički dizajn sistema*** koji objašnjava ***šta*** će sistem raditi sa apstraktnom specifikacijom.
2. ***Fizički dizajn sistema*** koji objašnjava ***kako*** će sistem izvesti te funkcije sa fizičkom specifikacijom.

Specifikacija *Logičkog dizajna* uključuje dizajn output-a, input-a, procesiranja, baza podataka, telekomunikacija, kontrola, sigurnosti i poslova IS-a.
Specifikacija *Fizičkog dizajna* uključuje dizajn hardware-a, software-a, baza podataka, telekomunikacija i procedura.

Kada su oba ova aspekta specifikacije sistema prihvaćena od svih učesnika, oni će se i dalje menjati. Najčešće će korisnici tražiti da se doda još funkcija sistemu (*called scope creep* ili menjanje

opsega), jer kada korisnik bolje razume kako će sistem raditi i informacije koje će procesirati primetiće dodatne funkcije koje želi da sistem radi. *Business* okruženje se menja i korisnici žele sistem s novim funkcionalnostima. Pošto je *scope creep* dosta skup moramo kontrolisati zahteve klijenata da projekat ne bi postao nekontroslisan (*runaway projects*).

PROGRAMIRANJE (KODIFIKACIJA). Programeri će upotrebljavati specifikaciju dizajna za izradu potrebnog software-a koji će dati odgovarajuću funkcionalnost sistemu u skladu sa postavljenim ciljevima i rešiti poslovne probleme. Ova faza uključuje izradu koda kompjuterskih programa i kreiranje struktura podataka.

TESTIRANJE. Temeljito i kontinuirano testiranje se radi posle faze programiranja. Testiranjem se utvrđuje da li napisani kod postiže planirane i željene rezultate u stvarnim uslovima. Testiranje iziskuje dosta vremena i napora da bi se došlo do softvera koji ispravno funkcioniše.

Test se dizajnira tako da otkrije greške (*bugs*) u kompjuterskom kodu. Postoje dva tipa grešaka: *sintaksne i logičke greške*. Sintaksne greške (npr. pogrešno postavljen zarez) ne dozvoljavaju da se program izvrši (*run*). Logičke greške dozvoljavaju da se program izvrši, ali daje pogrešne rezultate (outputs). Logičke greške teže je pronaći, jer razlozi nisu vidljivi. Programer mora slediti logički tok programa da bi pronašao razlog pogrešnih rezultata. Kako *software* postaje kompleksniji, tako se povećava i broj grešaka i postaje nemoguće pronaći ih sve. Zbog ove situacije javlja se pojam *good-enough software*, softver koji pronalazi zaostale greške u kodu. Oni moraju pronaći *show-stopper bugs*, greške koje prouzrokuju pad sistema ili katastrofalne posledice po podatke. Verifikacija garantuje ispravnost (korektnost) softvera, a validacija da softver ispunjava postavljene zadatke i uslove. Verifikacija i validacija se rade nakon svake faze.

IMPLEMENTACIJA (UVOĐENJE) je proces prelaska sa starog na novi sistem. Organizacije koriste četri osnovna tipa prelaska: paralelni, direktni, pilot i u fazama.

U procesu paralelnog prelaska stari i novi sistem funkcionišu istovremeno (paralelno) jedan određeni period. Sistemi obavljaju

slične procese sa sličnim podacima i njihovi izlazi se upoređuju. Ovaj tip prelaza se najviše koristi, jer je najmanje rizičan.

U direktnom procesu prelaska stari sistem se odbacuje i novi sistem se pokreće. Ovaj sistem prelaska se najmanje koristi zbog visoke rizičnosti, jer novi sistem može da ne obavlja planirane zadatke. Malo sistema koristi ovaj tip prelaska zbog visokog rizika.

Pilot proces prelaska uvodi novi sistem u jednom delu organizacije i ocenjuje se njegov rad. Nakon što se utvrdi da sistem radi kako treba uvodi se i u ostalim delovima organizacije.

Procesom prelaska na novi sistem u fazama (fazni) uvode se delovi sistema po fazama. Svaki deo (modul) je ocenjen i kada radi ispravno, sledeći modul sistema je uveden, sve dok celokupan sistem ne postane operativan.

OPERATIVNA FAZA I ODRŽAVANJE. Pre prelaska na novi sistem on mora biti operativan jedno određeno vreme, sve dok se ne postignu zadati ciljevi . U operativnoj fazi sistem se stalno kontroliše i ocenjuje se njegov kapacitet i određuje se da li se može početi koristiti na ispravan način.

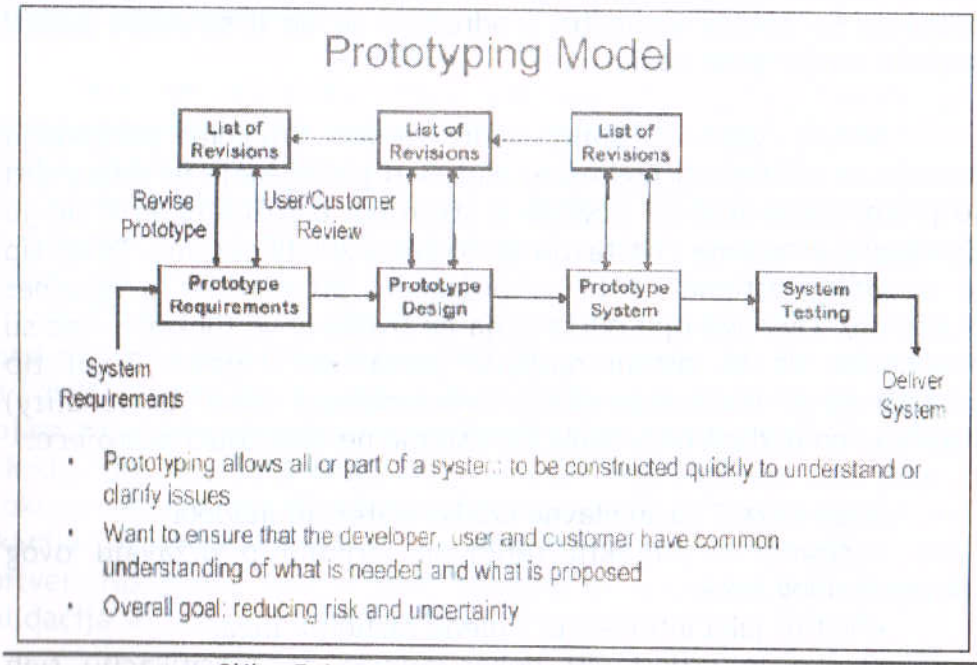
Sistem zahteva stalno održavanje. Prilikom održavanja nastoje se eliminisati pogreške, napraviti poboljšanja ili prilagoditi se promenama koje se dogode u stvarnom okruženju. Prvi tip je *debugging* programa i nastavlja se do kraja života sistema. Drugi tip je *updating* sistema da bi se prilagodio promenama u *bussines* okruženju. Ova dva tipa održavanja ne uvode nove funkcije, već su neophodne da bi sistem nastavio ostvarivati ciljeve. Treći tip održavanja je dodavanje novih funkcionalnost (*new funcionality*) sistemu i on dodaje nove funkcije sistemu ne ometajući postojeće.

Može se reći da su glavne odlike Waterfall modela:

- Učesnici u projektu retko se striktno pridržavaju ovog sekvencijalnog toka,
- Postoji jaka interakcija između različitih faza,
- Teško je uspostaviti dobar princip za specifikaciju svih zahteva,
- Konačni cilj je dobiti operativni proizvod.

5.1.2. Model prototipskog razvoja

Prototyping je alternativa razvoja softvera gde prvo ostvarujemo generalnu ideju o zahtevima njegovih budućih korisnika (klijent u početku definiše inicijalne potrebe, ali ne i detaljne zahteve). *Prototyping* je prethodna verzija nečega što ćemo raditi posle i ne planira razvoj sistema kao jedne celine, ali nam dozvoljava probati delove sistema koji su najinteresantniji za korisnike ili je model koji simulira kompletan sistem. Prototipi se konstruišu za korisnike da ih oni probaju i daju im viziju na koji će način moći koristiti sistem i nagoveštaj o poboljšanju uslova rada. Tim koji izgrađuje prototipe pregleda ih zajedno sa budućim korisnicima i koriste njihove sugestije za poboljšanje prototipa. Ovaj proces se obavlja sve dok korisnici ne ocene zadavoljavajućim jedan određeni prototip ili se ne zaključi da sistem ne može da ispuni zahteve korisnika. Ako je sistem moguć onda korisnici probaju prototip koji razvija kompletan sistem.

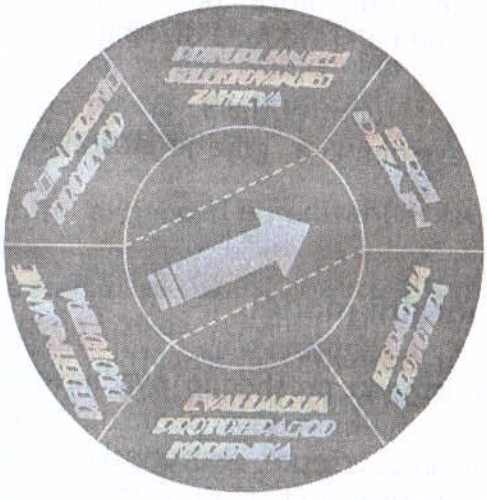


Slika 5.3.a.Model prototipskog razvoja

Prototyping omogućuje programeru kreiranje softverskog modela proizvoda koji će se konstruirati:

- Prototipi na papiru,

- Implementacija samo interfase-a,
- Adaptirani jedan deo već postojećeg programa.



Slika 5.3.b.Model prototipskog razvoja

Prednosti *prototyping*-a su sledeći:

- Ubrzava razvojni proces.
- Daje korisnicima mogućnost da shvate svoje informacijske potrebe i da provere interakciju sa novim sistemom putem operativne verzije koju dobiju jako rano.
- Utvrđuje ispravnost postavljenih zahteva od strane korisnika.

Prototyoing ima svoje nedostatake:

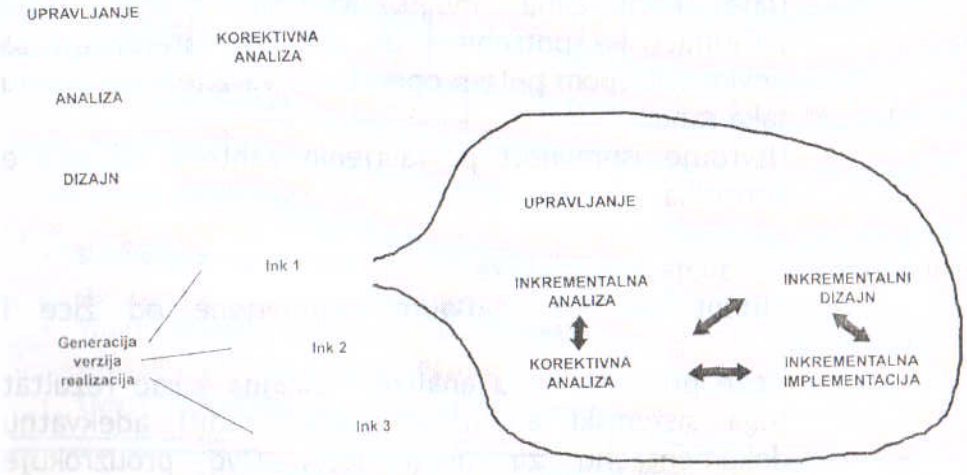
- Klijent ignoriše «figure napravljene od žice i plastelina»
- Može produžiti fazu analize i dizajna i kao rezultat toga sistemski analitičari neće uraditi adekvatnu dokumentaciju za programere. Ovo prouzrokuje poteškoće u operativnoj fazi i u fazi održavanja sistema.
- Postoji rizik da korisnici žele prerano zadržati jedan prototip koji nije dovoljno ispravan.

- Može se desiti da jedan element bez dovoljno kvalteta se uključi u definitivnu verziju.
- Razvojni tim može da ne uzime u obzir odluke donesene na kraju faze dizajna.

Prototyping se upotrebljava u razvoju DSS-a (*Decision support system*) i izvršnim IS-a, gdje je interakcija sa korisnikom od velikog značaja.

5.1.3. Inkrementalni model

Kompromis između prethodno dva navedena procesa jeste ovaj model. Sekvencijalni model podrazumeva linearni razvoj i uručenje projekta po završetku. Prototipski iterativni proces može pomoći budućim korisnicima i razvojnom timu u redefinaciji zahteva, a ne po predaji projekta. Inkrementalni model koristi «ubrzano» sekvencijalni model (prototipi) i u svakoj iteraciji proizvodi jednu verziju programa (sekvencijalni) koji nazivamo ***inkrement***.



Slika 5.4. Imkrementalni model

5.1.4. Spiralni model

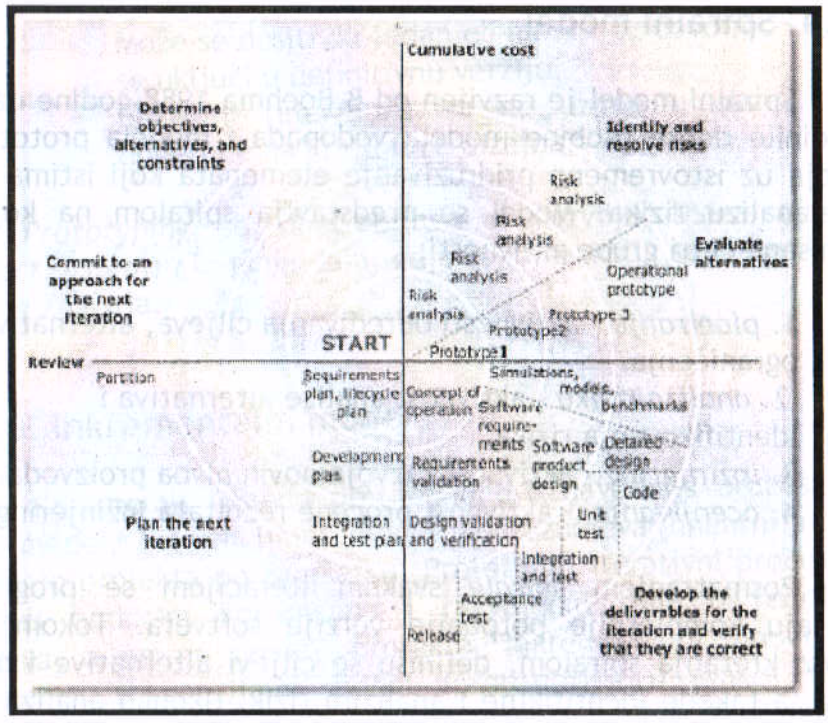
Spiralni model je razvijen od B.Boehma 1988.godine da bi se objedinile dobre osobine modela vodopada i modela prototipskog razvoja uz istovremeno pridruživanje elemenata koji istima nedostaje-analizu rizika. Model se predstavlja spiralom na kojoj su definisane četri grupe aktivnosti:

1. *planiranje* - aktivnosti određivanja ciljeva, alternativa i ograničenja.
2. *analiza rizika* - aktivnost analize alternativa i identifikovanja rizika.
3. *inžinjering* - aktivnost razvoja novih nivoa proizvoda.
4. *ocenjivanje* - aktivnost procene rezultata inžinjeringa.

Posmatranjem *spirale* svakom iteracijom se progresivno razvijaju kompletnije potpunije verzije softvera. Tokom *prvog ciklusa* kretanja spiralom, definišu se ciljevi alternative i ograničenja, a takođe identifikuje i analizira rizik. Ukoliko analiza rizika indicira neizvesnosti u zahtevima, tada se može upotrebiti prototipski razvoj da bi se zahtevi detaljnije spoznali. U iste svrhe, mogu se koristiti simulacija ili druge vrste modela.

Inžinjering se obavlja u *svakom ciklusu spirale* i to pristupom razvoja softvera po modelu životnog ciklusa ili modelu prototipskog razvoja. Broj aktvnosti u inžinjeringu raste, ukoliko se ciklusi udaljuju od centra spirale.

Korisnik *ocenjuje* inženjerski posao razvoja i daje sugestije za modifikaciju. Zasnovana na korisničkom *input*-u, javlja se sledeća faza *planiranja i analize rizika.* Svaki ciklus na spirali zahteva analizu rizika i donošenje odluke "nastaviti" ili ne nastaviti sa daljim razvojem. Ukoliko je rizik suviše velik, završava se svaki dalji rad. Svaki novozapočeti ciklus spirale donosi i kompletniji proizvod, ali i značajnije i više troškove.



Slika 5.5. Spiralni model

Model uključuje *multiplikovane iteracije* kroz cikluse koji analiziraju rezultate prethodnih faza određujući procenu rizika za buduće faze. U svakoj od faza razvijaju se alternative u skladu sa ciljevima i potrebama koje zajedno čine bazu za sledeći ciklus u spirali. Svaki ciklus završava ocenom. Model podrazumeva da se svaki deo proizvoda ili svaki nivo proizvoda na istovetan način provlači kroz ovu spiralu odnosno ocenu. Osnovna premisa modela je da se određeni redosled koraka ponavlja u razvoju i održavanju softvera. Koraci se prvo izvršavaju sa visokim stepenom apstrakcije. Svaka petlja spirale predstavlja ponovljene korake na nižem stepenu apstrakcije.

Prednost ovog modela je u njegovoj fleksibilnosti za upravljanje softverskim inženjeringom. Procena rizika se može izvesti u svakom trenutku i nivou apstrakcije. Model prilagođava svaku kombinaciju različitih pristupa u razvoju softvera.

Ovo je trenutno najrealniji pristup u razvoju softvera za

velike sisteme. Model omogućuje brzu reakciju na uočene rizike, a primenom prototipskog razvoja pruža mehanizam za njihovo smanjenje. Tako se primenom ovog modela rizici mogu smanjiti pre nego što izazovu veće probleme i pre svega veće troškove. Model podržava sistematski pristup preuzet iz modela vodopada uz mogućnost izvođenja iteracija. I pored velikog broja prednosti, model poseduje i nedostatke.

Nedostatak ovog modela je odsustvo veze prema postojećim standardima, odnosno ne postojanje standarda za ovaj način razvoja sistema. Takođe, model zahvata više uniformnosti i konzistentnosti u razvoju sistema. Velike probleme stvara situacija kada se na vreme ne otkriju rizici. Konačno, model je relativno nov i nije bio široko primenjivan. Stoga, biće potrebno još dosta vremena da se sa više sigurnosti i verovatnoće priđe njegovoj ozbiljnoj primeni.

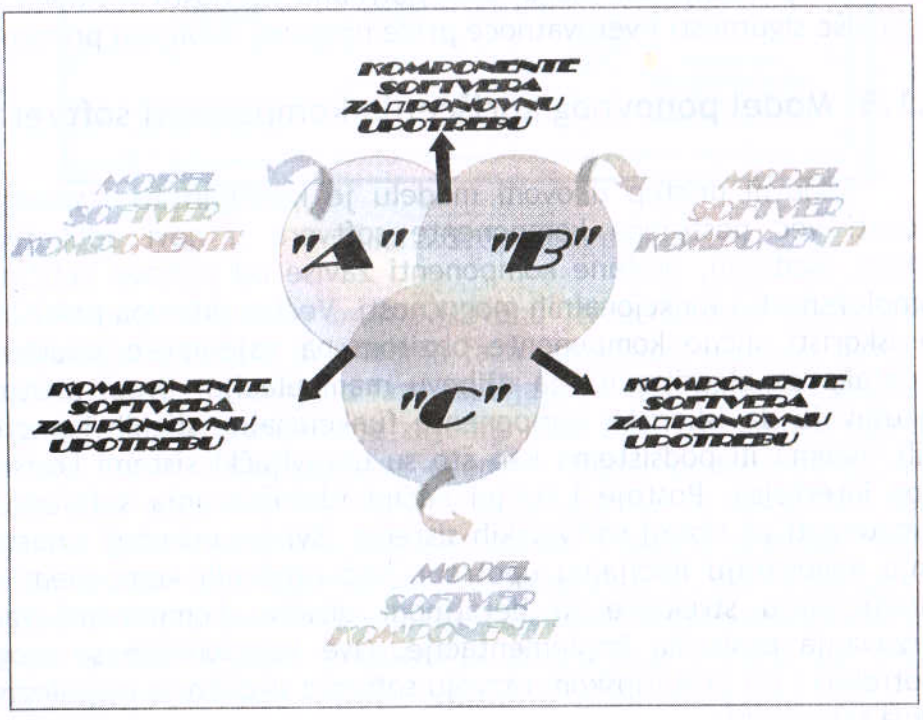
5.1.5. Model ponovnog korišćenja komponenti softvera

Osnovni pristup u ovom modelu je konfigurisati i specijalizovati već postojeće komponente softvera u novi aplikativni sistem. Međutim, osobine komponenti zavise od njihove veličine, kompleksnosti i funkcionalnih mogućnosti. Većina pristupa pokušava da iskoristi slične komponente obzirom na zajedničke strukture podataka sa algoritmima za njihovu manipulaciju. Drugi pristupi pokušavaju da iskoriste komponente funkcionalno sličnih kompletnih sistema ili podsistema kao što su upravljački sistemi korisničkog interfejsa. Postoje i brojni načini iskorišćavanja softverskih komponenti za razvoj softverskih sistema. Svi ovi pokušaji i nastojanja zagovaraju inicijalnu upotrebu već urađenih komponenti u specificiranju strukture ili detaljnom dizajnu komponenti radi ubrzavanja postupka implementacije. Ove komponente se mogu upotrebiti i pri prototipskom razvoju softvera ukoliko je raspoloživa takva tehnologija.

Višestruko korišćenje softvera je proces uključivanja u novi proizvod pojedinih komponenti: prethodno testiranog koda, prethodno proverenog dizajna, prethodno razvijene i korišćene specifikacije zahteva i prethodno korišćenih procedura za testiranje.

Prednosti koje sobom donosi ponovno korišćenje komponenti razvijenog softvera su sledeće:

- podiže robusnost softvera,
- povećava produktivnost izrade softvera,
- povećava kvalitet softvera,
- smanjuje troškove razvoja softvera,
- štedi odnosno skraćuje vreme izrade,
- zadovoljava ciljeve softverskog inžinjeringa,
- obezbeđuje adekvatnu dokumentaciju,
- olakšava održavanje softvera,
- modelira sistem za lakše razumevanje i dr.



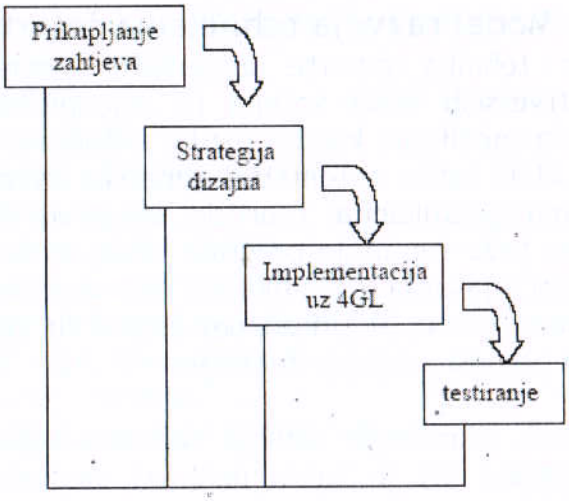
Slika 5.6.Model ponovnog korišćenja komponenti

5.1.6. Model razvoja tehnikama četvrte generacije

Pojam tehnika četvrte generacije podrazumeva široko područje softverskih alata kojima je zajedničko da omogućuju projektantu da specificira karakteristike softvera na najvišem nivou apstrakcije. Alati zatim automatski generišu izvorni kod na osnovama utvrđenih specifikacija. Teorijski, što je viši nivo specifikacije softvera to se brže izgrađuje program. Ovaj model se fokusira na mogućnosti sačinjavanja specifikacije softvera na nivou koji je blizak računaru datom u prirodnom jeziku ili putem korišćenja notacije koja prenosi značajne funkcije.

Trenutno, okruženje razvoja softvera koje podržava ovaj model sačinjavaju svi ili samo neki od sledećih alata: neproceduralni jezici, generatori izveštaja, alati za upravljanje podacima, alati za definisanje ekrana, generatori koda. Većina navedenih alata postoji, ali samo za domen veoma specifične upotrebe. Ne postoji okruženje tehnika četvrte generacije raspoloživo za upotrebu kod razvoja svih kategorija softvera.

Kao i kod ostalih modela, mogu se uočiti određene aktivnosti u razvoju softvera. One započinju prikupljanjem zahteva. Idealna situacija je ukoliko korisnik opisuje svoje zahteve i ovi se direktno transformišu u operativni prototip. Medutim, to je čak i nezamislivo. Korisnik može biti nesiguran u definisanju onoga šta traži, može biti nejasan i dvosmislen u specificiranju detalja koje poznaje i može biti u nemogućnosti ili ne želi da specifikuje informacije na način kako bi ih alati mogli koristiti. Alati 4GT još nisu dovoljno razvijeni da bi spoznali i prirodan jezik. Stoga dijalog korisnika i projektanta opisan u ostalim modelima predstavlja i deo ovog modela.



Slika 5.7. Model razvoja tehnikama IV generacije

Kod manjih aplikacija, moguće je direktno preći iz aktivnosti prikupljanja zahteva na aktivnost uvođenja upotrebom neproceduralnih jezika četvrte generacije (4GL). Međutim, kod ozbiljnijih poduhvata, neophodno je razviti strategiju dizajna sistema i pored upotrebe 4GT. Upotreba 4GT, bez aktivnosti dizajna, može uzrokovati poteškoće kakve su npr. nizak kvalitet, teško održavanje, teško prihvatanje proizvoda od korisnika i dr., koje se javljaju i kod ostalih modela softverskog inženjeringa.

Implementacija je sledeća aktivnost koja se obavlja primenom 4GT, koji omogućuju projektantu da automatski generiše i predstavi rezultate na osnovu automatskog generisanja koda. Za ove potrebe, moraju biti raspoložive strukture podataka sa informacijama kojima 4GL mogu pristupati. Da bi generisani kod pretvorio u proizvod, projektant mora proći sledeći put:

- Prikupljanje zahteva,
- Strategija dizajna,
- Implementacija uz 4GL,
- Testiranja, izrade potrebne dokumentacije i izvršiti sve ostale aktivnosti prevođenja, karakteristične i za ostale modele.

Oprečna su mišljenja pojedinaca koji zagovaraju ili negiraju primenu ovog modela. Pristalice modela tvrde da je postignuta

značajna redukcija vremena u razvoju softvera i visoko podignuta produktivnost onih koji razvijaju softver. Međutim, protivnici ovog modela tvrde da postojeći alati 4GT nisu baš mnogo lakši za upotrebu od programskih jezika, da je izvorni kod proizveden automatski neefikasan i da mogućnost održavanja velikih softverskih sistema razvijenih sa 4GT predstavlja veliku nepoznanicu i neizvesnost.

Činjenice koje treba istaći prilikom prikaza karakteristika ovog modela su sledeće:

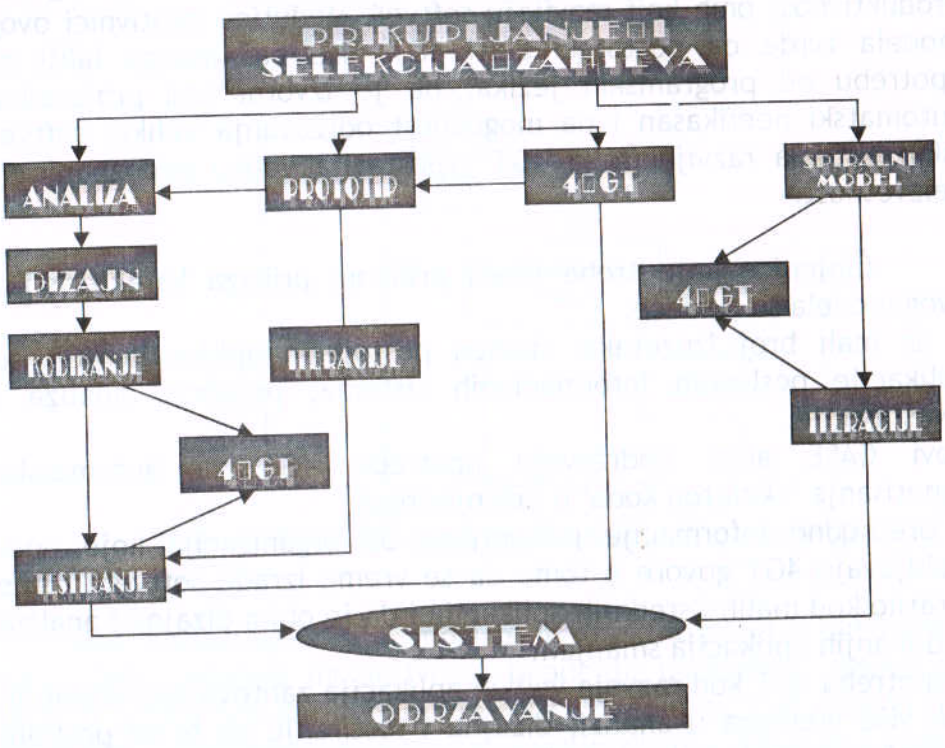
- uz mali broj izuzetaka, domen postojećih aplikacija 4GT su aplikacije poslovnih informacionih sistema, posebno, analiza i izveštavanje.

Novi CASE alati podržavaju upotrebu 4GT za automatsko generisanje "*skeleton* koda" u inženjeringu;

- prethodne informacije prikupljene od organizacija koje upotrebljavaju 4GT govore o tome da se vreme izrade softvera bitno skratilo kod malih i srednjih aplikacija i da je obim dizajna i analize kod manjih aplikacija smanjen;
- upotreba 4GT kod razvoja velikih aplikacija zahteva isto vreme ili čak više vremena u analizi, dizajnu i testiranju da bi se postigle značajnije uštede vremena eliminisanjem kodiranja.

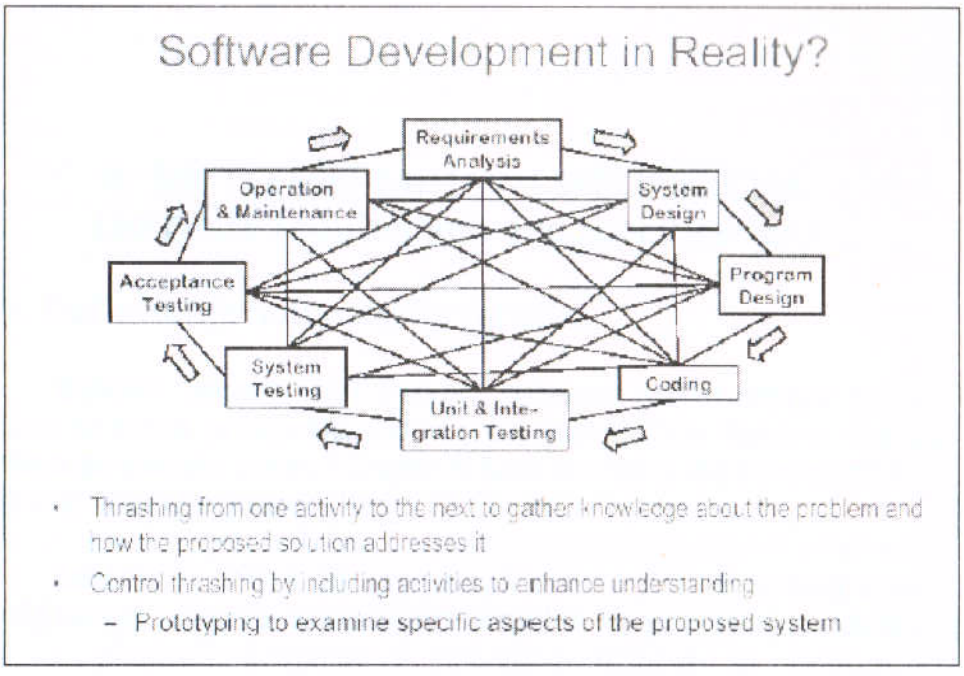
5.1.7. Kombinovani modeli

Napred opisani modeli su uglavnom prikazivani kao alternativni, a manje kao komplementarni modeli softverskog inženjeringa. Međutim, u mnogim situacijama modeli se mogu kombinovati tako da se postignu prednosti od svih na samo jednom projektu. Spiralni model je i sam primer dobre kombinacije dva modela, ali i drugi modeli mogu poslužiti kao osnova na koju će se integrisati neki modeli. Ne treba biti dogmatičan u izboru određenog modela u softverskom inženjeringu. Priroda aplikacije će diktirati model koji bi trebalo primeniti. Kombinovanjem modela, rezultat postignut u celini može biti povoljniji nego što bi to bio prosti zbir rezultata postignutih pojedinim modelima.



Slika 5.8. Kombinovani model

Na kraju ovog poglavlja, napomena: da postoji još mnoštvo modela razvoja softvera!



Slika 5.8. Razvoj softvera u stvarnosti?

Literatura:

1. Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin, 2001.
2. B.Jošanov,P.Tumbas, *Softverski inženjering*, VPŠ, Novi Sad, 2002.
3. www.fit.ba (e-predavanja iz Softverskog inžinjeringa)
4. MIT OpenCourseWare » *Civil and Environmental Engineering* » 1.264J Database, Internet, and Systems Integration Technologies, Fall 2002

6. SOFTVERSKI ZAHTEVI I SPECIFIKACIJA (Software requirement and specification)

6.1. Definisanje softverskih zahteva

Zahtevi jednog softverskog sistema predstavljaju uslove kojima se treba prilagoditi jedan softverski sistem. Zahtevi jednog sistema se moraju pronaći i specificirati na način da su razumljivi i prihvatljivi za dizajnere sistema i klijente.

Zahtevi su jedan od najvažnijih delova projekata koji imaju za cilj razvoj softvera. Svi učesnici projekta koriste zahteve da bi:

- planirali projekat i potrebne resurse za realizaciju projekta,
-odredili tip verifikacije sistema, npr.: kada se nastoji slediti određeni standard.
-planirali strategiju testiranja sistema, npr.: na osnovu zahteva se određuje da li je određeno testiranje sistema izvršeno uspešno ili ne.

Korisnički zahtevi su osnova životnog ciklusa projekta. Dokumentovani zahtevi su osnova za kreiranje dokumentacije sistema, a ispravno definisanje i adekvatno korištenje zahteva je veoma važan deo projekta.

6.1.1. Karakteristike zahteva

Zahtevi su važan deo softverskog sistema, pa je potrebno definisati i karakteristike koje moraju imati pravilno formulisani zahtevi.

Karakteristike zahteva:

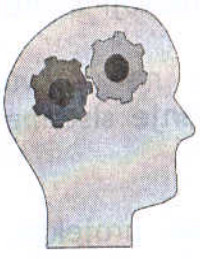
- **Zahtevi moraju biti specificirani u pisanoj formi**, kao i svaki ugovor.

- **Moguće ih je testirati i verifikovati.** Ukoliko ne možemo proveriti jedan zahtev, ne možemo ni znati da li smo ga ispunili ili ne.
- **Moraju biti opisani kao jedna od karakteristika sistema.** To znači: šta sistem treba da radi (a **ne kako** treba da radi).
- **Zahtevi moraju biti što je moguće jasniji,** da bi se izbegle neispravne interpretacije.

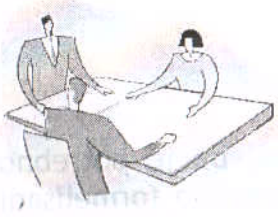
Analiza zahteva

Skup zahteva jednog softverskog sistema je kompleksan, jer ima mnoštvo elemenata koji se detaljno opisuju i povezani su međusobno. U toku definisanja zahteva, klasificiramo, struktuiramo i organizujemo zahteve (što sistem treba da radi?) u cilju specifikacije kompleksnog sistema na jednostavan način.

Analiza zahteva uključuje sledeće korake:



- prikupiti informacije o korisničkim zahtevima,
- klasificirati prikupljene informacije (zahteve),
- prepoznati nivoe hijerarhije u sistemu i početi smeštati već klasificirane zahteve u odgovarajuće nivoe u hijerarhiji.



Zahtevi su posledica dogovora između klijenta i projektnog tima razvoja softvera. Ovaj dogovor je potreban da bi smo zadovoljili potrebe klijenta. Zahtevi su usmereni opisu potreba klijenta i kao takve, logično je da ih uzimamo iz «prve ruke» putem intervjua sa klijentom ili proučavajući dokumentaciju koja opisuje na koji način klijent želi da sistem funkcionira.

Interviju

Interviju je tehnika sakupljanja informacija putem dijaloga na planiranom i formalnom sastanku između jedne ili više osoba koje su *izvor* informacija i jedne ili više osoba koje su *receptori* istih, (inženjeri sistema i analitičari zaduženi za realizaciju projekta). Dobijene informacije se transformišu i sistematizuju na način da budu korisni elementi za razvoj informacioih sistema u fazi analize.

Klasifikacija intervjua se vrši na sledeći način:

- *Struktuirani intervjui* se sastoje od pitanja unapred formulisanih i zapisanih, a prethodno definisanih. Tipovi odgovora struktuiranih intervjua su:
  - *otvoreni odgovori*: intervjuisani odgovara slobodno na pitanja.
  - *zatvoreni odgovori*: intervjuisani bira između ponuđenih odgovora.

- *Nestruktuirani intervjui* gde analitičari nemaju pripremljena pitanja već koriste dosadašnja iskustva da bi od korisnika saznali postojeće probleme u sistemu. Kako pitanja, tako i odgovori su otvorenog tipa.

- Na *mešovitiom intervju* su podjeljena pitanja u dve grupe, u grupu pitanja prvog tipa i grupu pitanja drugog tipa. Ovaj tip intervjua bi mogao rešiti probleme prethodna dva. U svakom slučaju potrebno je prilagoditi se potrebama prikupljanja informacija.

Tabela 6.1. Prednosti i nedostaci opisanih tipova intervjua:

	Struktuirani	Nestruktuirani
Fleksibilnost u pitanjima	NEDOSTATAK	PREDNOST
Fleksibilnost u sektorima informacija koje se istražuju	NEDOSTATAK	PREDNOST
Subjektivni uticaj intervjuisanih lica	PREDNOST	ZAVISI OD RECEPTORA
Pripremanje receptora	PREDNOST	NEDOSTATAK
Objektivno ocjnjivanje pitanja i odgovora	PREDNOST	NEDOSTATAK

Komfornost intervjuisanih	ZAVISI OD RECEPTORA	ZAVISI OD RECEPTORA
Trajanje	PREDNOST	NEDOSTATAK
Efikasnost intervjua (informacija/vrema)	PREDNOST	ZAVISI OD RECEPTORA
Administracija i ocenjivanje (zaključci)	PREDNOST	NEDOSTATAK
Ujednačenost među intervjuisanim licima	PREDNOST	ZAVISI OD RECEPTORA
Cena priprema	NEDOSTATAK	PREDNOST

Rezultati intervjua su odlučujući za razvoj projekta, jer na ovim informacijama će se zasnivati razvoj softverskog sistema.

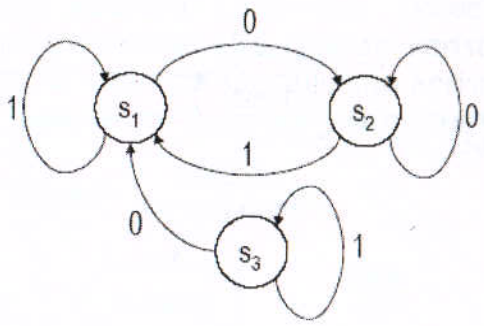
Posmatranjima prikupljamo informacije koje nismo dobili od korisnika i u slučaju potrebe nadopunjujemo ih novim intervjuima.

Potrebe i/ili zahtevi klijenta se povećavaju s vremenom, a svaka promena ima svoju cijenu. Potrebno je čuvati orginalnu dokumentaciju klijentovih zahteva, kao i svaku reviziju ili promenu ovog dokumenta.

Zahteve klijenta je potrebno klasificirati na one koji će biti ispunjeni softverom i one koje će ispuniti drugi delovi sistema.

6.2. Tehnike za specifikaciju zahteva

Postoje formalne, neformalne i semiformalne metode za specifikaciju zahteva. **Formalne metode** su regularni izrazi, dijagrami stanja (transition diagrams), mrežni dijagrami. Ove metode nisu u širokoj upotrebi, jer korisnici najćešće nisu upoznati sa tim notacijama i ne razume ih.



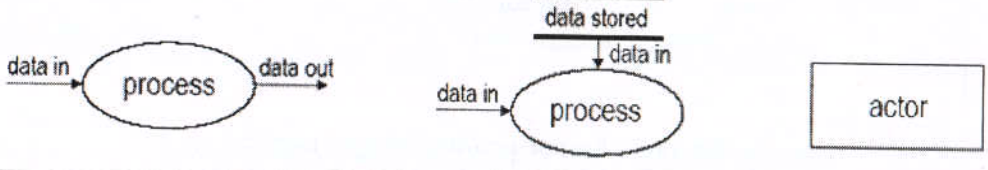
Slika 6.1.Primer dijagrama stanja

Semiformalne metode su tablice odlučivanja, dijagrami toka podataka i rečnici podataka.

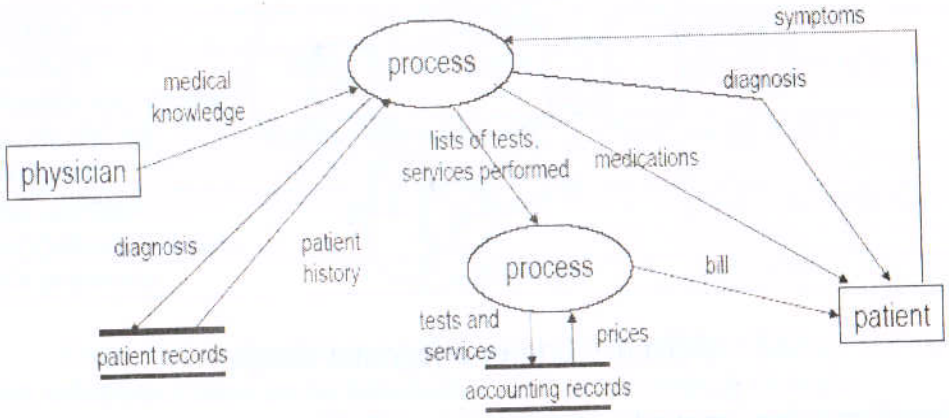
	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
High standardized exam scores	T	F	F	F	F
High grades	-	T	F	F	F
Outside activities	-	-	T	F	F
Good recommendations	-	-	-	T	F
Send rejection letter			X	X	X
Send admission forms	X	X			

Tabela 6.2. Primer tablice odlučivanja (Redovi su uslovi, a kolone su skupovi uslova)

Tok podataka u sistemu je prikazan na Slici 6.2 a.



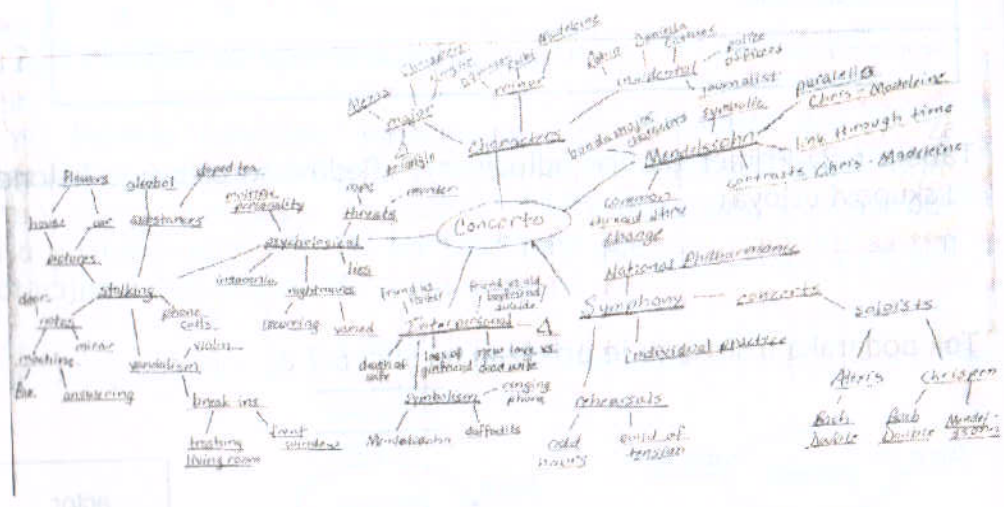
Poseta lekaru:



Slika 6.2.b. Primer dijagrama toka podataka

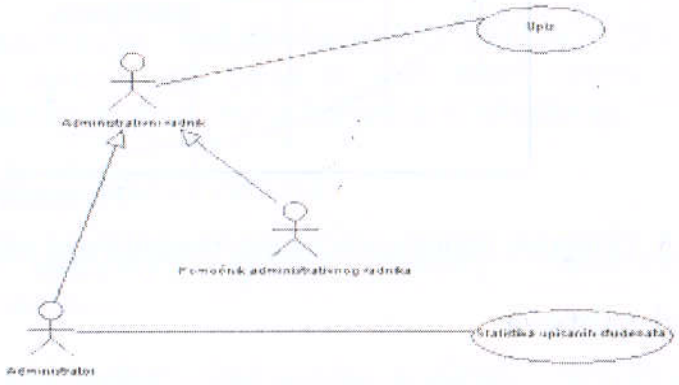
Neformalne metode su prirodni jezik i struktuirani prirodni jezik, kao i *brainstorming* dijagrami.

Brainstorming dijagram predstavlja kreativne ideje za rešenje određenog problema. Sesije su grupne i svaka ideja je prezentirana ostalim članovima ekipe. Ključna stvar je: «*environment free of criticism*» s ciljem ekploatacije različitih opcija rešenja problema.

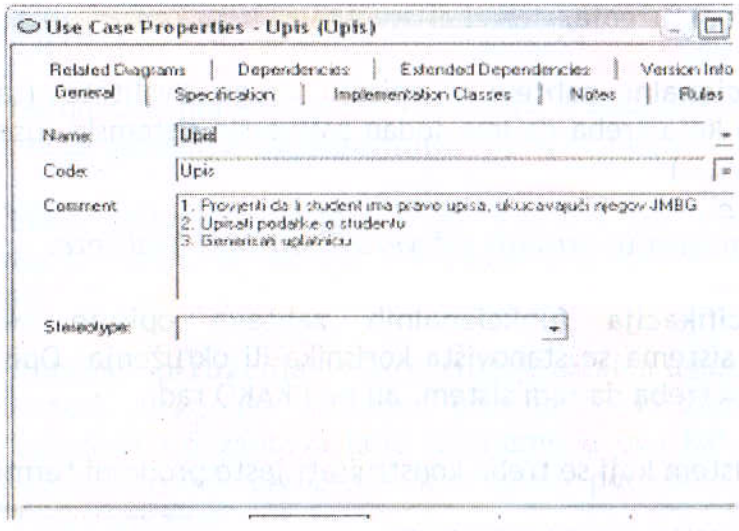


Slika 6.3. Primer *brainstorming* dijagrama

UML (*Unifed Modeling Language*) poseduje *uses cases* u *actors*-u za notaciju zahteva. *Use case* se koristi za specifikaciju ponašanja sistema, a *actors* za specifikaciju korisničkih uloga, Slika 6.4.a. Dijagram UML-a je propraćen struktuiranim tekstualnim opisom, Slika 6.4.b.

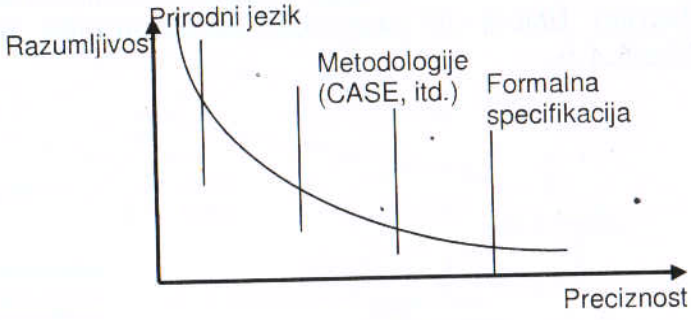


Slika 6.4.a. Primer *use case* dijagrama



Slika 6.4.b. Tekstuelni opis *use case* dijagrama

Odnos između preciznosti i razumljivosti zahteva predstavljen je dijagramom na Slici 6.5.



Slika 6.5. Dijagram odnosa preciznost/razumljivost zahteva

6.3. Tipovi zahteva

Klasifikacija zahteva je jedan od načina njihove organizacije. Različite karakteristike zahteva ne možemo posmatrati na isti način. Najčešće razlikujemo dve familije zahteva u softverskim sistemima:

- **Funkcionalni zahtevi** - opisuju na koji način sistem odgovara na događaje iz okruženja.

- **Nefunkcionalni zahtevi** - opisuju druge kvalitete (različita ponašanja) koja treba da ima jedan softverski sistemski (*usability, availability,...*)

Takođe se nazivaju: *atributi softvera* ili *atributi kvaliteta.*

Specifikacija funkcionalnih zahteva opisuje eksterno ponašanje sistema sa stanovišta korisnika ili okruženja. Opis treba da kaže ŠTA treba da radi sistem, ali ne i KAKO radi.

Primjer: Sistem koji se treba konstruisati jeste prodajni terminal.

Funkcionalni zahtevi: Prodavati proizvode
Klijent dolazi na kasu sa proizvodima koje želi kupiti. Prodavac na kasi koristi prodajni terminal da bi registrovao sve proizvode i izdao račun na kojem stoji cena svakog od proizvoda i konačna suma.

• Funkcionalni zahtevi se registruju u dokumentu Model slučajeva upotrebe

•Komplementarni dokumenti: rečnik (Glossary) i sažetak.

Specifikacija nefunkcionalnih zahteva opisuje interno ponašanje sistema.Opis treba da kaže KAKO sistem odgovara na zahteva postavljene od strane korisnika ili okruženja.

Primeri nefunkcionalnih zahteva:

• **Usability**: Ljudski faktor, interface, pomoć, documentacija...

• **Availability**: frekvencija grešaka, rekonstrukcija posle greške...

• **Efikasnost**: vreme odgovora, optimizacija upotrebljivih resursa, prèciziranje rezultata

• **Podrška**: internacionalizacija, adaptacija, lako održavanje...

Nefunkcionalni zahtevi se registruju u slučajevima upotrebe sa kojima se povezuju, i u dokumentu koji specificira suplement.

Svaki softverski projekt može koristiti svoju klasifikaciju zahteva, a ovde se navodi još jedna prihvaćena klasifikacija zahteva:

- **Zahtevi «okruženja»** - Okruženje je okolina u kojoj se nalazi sistem. Iako ne možemo promeniti okruženje, postoji određeni tip zahteva koje ubrajamo u ovu kategoriju, jer sistem koristi okruženje kao izvor potrebnih usluga za funkcionisanje.

  Primeri: operativni sistemi, sistem fajlova, baze podataka.

Sistem mora tolerisati greške koje se mogu dogoditi u okruženju, kao što su greške u ulaznim podacima i zbog toga je potrebno uzeti u obzir okruženje u okviru zahteva.

- **Zahtevi komunikacije sa sistemom** - Najpoznatiji zahtev ovog tipa je GUI (Graphic User Interface). Ovi zahtevi su način na koji korisnici komuniciraju sa sistemom.

- **Zahtevi interfejsa** - Interfejs je način kako sistem komunicira sa korisnicima ili sa drugim sistemima (inverzni zahtevi zahtevima komunikacije sa sistemom). Interfejs je formalna specifikacija podataka koje sistem prima ili šalje u eksterijer. Najčešće se specificira protokol, tip informacije, medij komunikacije i format podataka.

- **Funkcionalni zahtevi**

- **Zahtevi efikasnosti** - Ovi zahtevi nam opisuju karakteristike efikasnosti koje treba da ima sistem. Koliko brzo? Koliko sigurno? Koliko resursa? Koliko transakcija?

Ovaj tip zahteva je od posebne važnosti za *real time* sisteme, gde je efikasnost sistema važna koliko i njegova funkcionalnost.

- **Dostupnost (u određenom vremenskom periodu)** - Ovaj tip zahteva se odnosi na trajnost, degradaciju, portabilnost, fleksibilnost i kapacitet aktuelizacije. Takođe je veoma bitan u *real time* sistemima.

- **Obuka** - Ovaj tip zahteva je usmeren ka ljudskim resursima koji će koristiti sistem. Koji je tip korisnika? Koji priručnici će se priložiti i na kojem jeziku?

Zahtevi obuke su veoma bitni u procesu dizajna (iako neće biti deo koda), jer olakšavaju uvođenje i prihvaćanje sistema u okruženju u kojem će isti biti smešten.

- **Ograničenja dizajna** - Pravila koja nazivamo «ograničenja dizajna» su rešenja jednog softverskog sistema koja slede određene zakone i standarde.

Ovde specificiramo na kojem medijumu će se predati sistem i kako je „upakovan". Važan je za određivanje cene industralizacije sistema.

6.4. Proces definiranja zahteva metodom iterativnog razvoja

Neizbježno, zahtevi softverskog sistema evoluiraju u toku svoje konstrukcije. Cilj nije elaborirati dugačku listu nepromenjivih zahteva (proces *waterfall*).

Metode iterativnog razvoja podržavaju inkrementalnu specifikaciju i evaluaciju zahteva:

- Na početku treba predstaviti listu zahteva sistema i detaljno opisati najvažnije zahteve.
- Otpočeti analizu i dizajn funkcionalnosti vezano za napred navedene zahteve.
- Inkrementalno poboljšavati prethodno navedene zahteve potpomognute rezultatima analize i dizajna.

Dokumenti koji čine analizu zahteva

Dokumenti koji čine analizu zahteva su:

- **Model slučaja upotrebe**

  Sastoji se od:

  - *Lista slučajeva upotrebe* (jedan slučaj upotrebe registruje zahteve; specijalno, funkcionalne zahtjeve)
  - Sekvencijalni dijagrami sistema (Sekvencijalni dijagram sistema čini eksplicitnim i organizuje događaje kreirane od strane korisnika sistema)
  - *Operacije sistema*

- **Ostali dokumenti**

  - *Glossary* (Definisanje uvedenih termina koji se pojavljuju u zahtevima)

- *Sažetak* (Perspektiva koju imaju klijenti sistema koji se razvija)
- *Komplementarna specifikacija* (Lista nefunkcionalnih zahteva koji su uključeni u slučajeve upotrebe)

Rad sa zahtevima

U saglasnosti sa "Capability Maturity Model" (CMM) rad sa zahtevima jeste: "Uspostaviti i održati dogovor o zahtevima softverskog projekta sa klijentom. U dogovor su uključeni tehnički i netehnički zahtevi (kao datum završetka projekta) i služi kao osnova za planiranje i izvršenje razvoja projekta u toku celog životnog ciklusa." ... "Prema zahtevima projekta grupa koja radi sa zahtevima brine se da zahtevi budu dokumentovani i kontrolisani na adekvatan način."[5]

CMM je vodič za kontrolu zahteva: "Da bi smo postigli kontrolu nad zahtevima, grupa koja radi sa zahtevima proverava ih pre uvođenja u softverski projekat i svaki put kada se zahtevi promene, planovi i aktivnosti se prilagođavaju."[6]

Treba imati u vidu dve stvari:

1. Zahtevi moraju biti provereni.
2. Tehnički i netehnički zahtevi formiraju jedan skup zahteva. Ukoliko se promeni jedan od zahteva, takođe su promenjeni i ostali. To znači da: više tehničkog sadržaja, znači veću cenu ili niži kvalitet ili više potrebnog vremena. Sve tehničke promene moraju biti odobrene od strane grupe koja radi sa zahtevima i ova grupa proverava uticaje na vreme, cenu i kvalitet. Rezultati ove provere su faktori koju utiču na odluku da li prihvatiti promene ili ne.

---

[5] *Paulk,M.C.,Weber,C.V.,Garcia,S.,Chrissis,M.B.Bush,M., Key Practise of the Capability Maturity Model, Ver.1.1. Software Engineering Institute, CMU/SEI-93-TR-25, 1995.*

[6] *Paulk,M.C.,Curtis,B.,Chrissis,M.B.Weber,C,V., Capability Maturity Model for Software, Ver.1.1. Software Engineering Institute, CMU/SEI-93-TR-24, 1993.*

Literatura:

5. Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin, 2001.
6. B.Jošanov,P.Tumbas, *Softverski inženjering*, VPŠ, Novi Sad, 2002.
7. www.fit.ba

7. OCENA KVALITETA SOFTVERSKIH PROIZVODA (Software product grade and quality check)

Sa rastom polja upotrebe informacionih tehnologija porastao je i broj aplikacija tj. razvijenog softvera. Kvalitet softvera je postala važna karakteristika, jer ukoliko softver ne zadovoljava specifične zahteve u pogledu kvaliteta može doći do ozbiljnih posledica u sistemima čiji je on deo.

Određivanje kvaliteta i standardizacija alata koji se koriste u procesu razvoja informacionih sistema stvaraju mogućnost da kvalitet samog procesa kao i krajnjih proizvoda bude na željenom i očekivanom nivou. Posmatrano sa stanovišta složenih informacionih sistema u čijem razvoju i implementaciji učestvuje više strana, korištenje softverskih proizvoda, koji zadovoljavaju utvrđene kriterijume ne samo da stvara uslove za razvoj softverskog proizvoda odgovarajućeg kvaliteta nego omogućava i zajednički rad na projektima.

Da bi se ubrzao razvoj softvera i omogućilo donošenje upravljačkih odluka u fazama projektovanja informacionih sistema bilo je potrebno utvrditi atribute softvera, odnosno modela informacionog sistema koji će omogućiti razumevanje i kontrolu kvaliteta softverskog razvoja i softverskih proizvoda. U tom cilju razvijaju se pojedine oblasti softverskog inženjerstva, čija je svrha da uspostave operativne procedure za razvoj kvalitetnog softvera i vrše mnogobrojna istraživanja i eksperimente. Softverska metrika je jedna od njih, a njen zadatak je uvođenje kvantitativnih i kvalitativnih indikatora za softverske proizvode u svim fazama njihovog razvoja i korištenja, kao i za tehnike i metodologije koje se koriste u njihovom razvoju.

Metod određivanja kvaliteta, metrike određivanja kvaliteta i alati za podršku određivanja kvaliteta su suštinski deo tehnologije određivanja kvaliteta softverskih proizvoda. Da bi se određivanje kvaliteta obavilo, zahtevi kvaliteta moraju biti specificirani, proces

ispitivanja kvaliteta softvera planiran, implementiran i kontrolisan, mogu biti vrednovani i međuproizvodi i krajnji proizvod. Određivanje kvaliteta softverskih proizvoda mora biti objektivno i nepristrasno, kvantitativno, dokumentovano, sa mogućnošću reprodukovanja.

Karakteristike kvaliteta

ISO/IEC JTC1/SC7 je utvrdio standarde i smernice za softversko inženjerstvo koje predstavljaju osnovu za određivanje kvaliteta softvera. ISO/IEC 9126 je prvi internacionalni standard u području određivanja kvaliteta softvera i metrika i u njemu je definisano šest karakteristika kvaliteta softvera i dat model procesa određivanja kvaliteta. Definisani atributi određuju osnovu za dalje preciziranje i opis kvaliteta softvera.

Tabela 7.1. Osnovne karakteristike kvaliteta softvera

- funkcionalnost	Da li zadovoljava formulisane ciljeve?
- pouzdanost	Kako često ispada?
- upotrebljivost	Koliko lako se koristi?
- efikasnost	Koliko resursa koristi?
- pogodnost za održavanje	Koliko lako se popravlja?
- prenosivost	Koliko lako se prenosi?

Za definisanih 6 osnovnih karakteristika kvaliteta softvera definisan je skup podkarakteristika:

- **Funkcionalnost**, sposobnost softvera da obezbedi funkcije koje zadovoljavaju formulisane i implicitne potrebe kada se softver koristi pod specificiranim uslovima, posmatra se kroz podkarakteristike: **podesnost, tačnost, interoperativnost i zaštitu.**

- **Pouzdanost,** sposobnost softvera da održava nivo performansi sistema kada se koristi pod specificiranim uslovima, izražava se preko: **zrelosti** (sposobnost softvera da speči otkaz kao rezultat grešaka u softveru), **tolerantnost** na **greške, oporavljivost, raspoloživost.**

- **Upotrebljivost**, osobina softvera da bude shvaćen, naučen, korišten i prihvaćen od korisnika, kada se koristi pod specificiranim uslovima. Podkarakteristike upotrebljivosti su **shvatljivost, pogodnost za učenje, operativnost.**

- **Efikasnost**, sposobnost softvera da obezbedi zahtevane performanse u zavisnosti od iskorištenosti resursa, pod formulisanim uslovima (resursi mogu obuhvatiti druge softverske proizvode, hardverske karakteristike, i slično). Efikasnost se izražava preko: **vremenskog ponašanja i korištenja resursa.**

- **Pogodnost za održavanje**, sposobnost softvera da se modifikuje. Modifikacije mogu obuhvatiti korekcije, poboljšanja ili prilagođenja softvera promenama u okruženju, i u zahtevima i u funkcionalnim specifikacijama. Osnovne podkarakteristike pogodnosti za održavanje su: **mogućnost analize, izmenljivost, stabilnost mogućnosti testiranja.**

- **Prenosivost**, osobina softvera koja omogućava njegovo prenošenje iz jednog okruženja u drugo. Ona dopušta projektantima softvera da koriste iste izvorne kodove aplikacija na više platformi. Prema izjavama nekih projektanata, od 50% do 70% njihovih napora odlazi na prenos softvera sa jedne na drugu platformu, umesto da razvijaju nove projekte. Na taj način su softverske inovacije sputane potrebom prenosa na nova hardverska okruženja. Interoperativnost je, pre svega, briga korisnika, a prenosivost je prvenstveno briga projektanata. Standardi koji podržavaju napore za prenosivošću su okupljeni oko API-a (Application Programming Interfaces). Prenosivost se izražava kroz: **prilagodljivost, mogućnost instalacije, koegzistenciju i zamenljivost.**

Svaka od definisanih podkarakteristika izražava se preko utvrđenih indikatora.

Metodologija određivanja kvaliteta

Određivanje kvaliteta softvera je podskup aktivnosti softverskog inženjerstva i kao takvo se može smatrati sistemom određivanja kvaliteta. Sistem određivanja kvaliteta je sastavljen od elemenata:

- ulazni proizvodi,
- resursi,
- kontrolni podaci,
- proces određivanja kvaliteta i
- izlazni proizvod.

Ulazni proizvodi su:

- specifikacije zahteva,
- ciljni entitet određivanja kvaliteta,
- informacije koje pokreću sistem određivanja kvaliteta,
- kao i informacije koje obezbeđuju kriterijume za određivanje kvaliteta.

Izlazni proizvod je rezultat procesa npr.:

- plan projekta,
- izveštaj,
- uputstvo,
- kriterijum određivanja kvaliteta.

Kontrole ili kontrolni podaci se koriste kao kriterijumi od strane procesa. Resursi su ulazi u proces. Primeri resursa su tehnologije, računarski programi, vreme i dr.

Entitet određivanja kvaliteta je objekat koji se vrednuje ili meri. Identifikuje se kada se definiše cilj određivanje kvaliteta.

Entitet određivanja kvaliteta je softverski proizvod koji može biti:

- operativni sistem,
- CASE alat i
- Softver za upravljanje bazama podataka, itd.

Osnovu procesa određivanja kvaliteta predstavljaju zahtevi koji se odnose na kvalitet softvera i oni se mogu podeliti u dve osnovne kategorije:

- funkcionalni zahtevi i
- zahtevi u odnosu na karakteristike kvaliteta.

Funkcionalni zahtevi se izvode iz funkcionalnih potreba koje treba da zadovolji softverski proizvod i zahteva u pogledu performansi. Radi jasnijeg i lakšeg korištenja preporučuje se da se iskažu putem hijerarhijske ček liste koja omogućava:

- grupisanje zajedno sličnih, ili povezanih funkcija,
- variranje nivoa detalja u definiciji potreba za različite funkcije,
- detaljizacija potreba tokom procesa analize.

Karakteristike kvaliteta softvera se daju preko karakteristika i podkarakteristike softvera definisanih u JUS ISO/IEC 9126.

Zbog činjenice da karakteristike softvera ne mogu biti kompletno specificirane, tj formalno povezane sa podkarakteristikama i sa metrikama, zahtevi određivanja kvaliteta mogu sadržati nivoe određivanja kvaliteta za izabrane karakteristike kvaliteta.

Nivoi određivanja kvaliteta su povezani sa važnošću pridodatoj svakoj karakteristici. Izabrani nivo treba da je značajan u odnosu na pretpostavljeno korištenje i okruženje softverskog proizvoda (npr. sigurnosni uslovi, ekonomski rizik, ograničenja primene).

Nivoima određivanja kvaliteta se isto tako definiše dubina ili temeljnost. Kao što određivanje kvaliteta, kao posledicu na različitim nivoima, daje različite nivoe poverenja u kvalitet softverskog proizvoda, tako nivo određivanja kvaliteta može biti

izabran nezavisno za svaku karakteristiku. Nivoi čine hijerarhiju tako da se na najvišem nivou primenjuju najstrožije tehnike određivanje kvaliteta. Idući prema nižim nivoima, postupno, koriste se manje strogi metodi i kao posledica toga obično se ulaže manji napor u određivanje kvaliteta.

Nivo određivanja kvaliteta za svaku karakteristiku softvera se može menjati kao i za njegove različite komponente (npr. komponente sa velikim zahtevima pouzdanosti mogu se odvojiti od drugih komponenata sistema).

Proces određivanje kvaliteta je skup zadataka ili aktivnosti koji transformiše ulazne proizvode u izlazne proizvode trošeći ljudske resurse i resurse računarskog sistema, koristeći metodologije i računarske programe. Proces određivanja kvaliteta se može dalje dekomponovati u podprocese:

- analiza zahteva određivanja kvaliteta,
- specifikacija određivanja kvaliteta,
- projektovanje određivanja kvaliteta,
- izvršenje određivanja kvaliteta i
- izveštavanje.

Ulazi u proces određivanja kvaliteta su:

- specifikacija zahteva za određivanje kvaliteta,
- ciljni entitet određivanje kvaliteta,
- informacije koje obezbjeđuju kriterijume.

U procesu određivanja kvaliteta se na osnovu utvrđenog cilja, a u saglasnosti sa standardima, obrađuju ulazi. Nakon toga, uspostavljaju se metrike, definišu karakteristike i podkarakteristike, stvaraju moduli određivanja kvaliteta na osnovu kojih se generiše plan određivanja kvaliteta. Na osnovu plana određivanja kvaliteta sprovodi se određivanje kvaliteta čiji se rezultati analiziraju i prave izvještaji.

Osnovu procesa određivanja kvaliteta čine zahtevi određivanja kvaliteta koji mogu biti iskazani preko:

- funkcionalnih zahteva: liste potrebnih funkcija, ukazivanje na njihovu važnost (koja se može dati dodeljivanjem težina ili klasa funkcijama),
- sadržaja korištenja softverskog proizvoda:
- proizvod dodeljen jednom zadatku ili za nekoliko zadataka,
- broja korisnika: jedan/više, odeljenje, organizacija, itd.
- profila korisnika: nivo eksperta, iskustvo, obučenost,
- definisanih zahteva kvaliteta njihovih rangova (dodeljivanjem težina ili klase svakoj karakteristici kvaliteta).

Proces 'Analiza zahteva određivanja kvaliteta' je složen proces koji u sebi sadrži procese:

- Razvoj i utvrđivanje načina akvizicije.
- Definisanje funkcionalnih zahteva.
- Identifikacija korištenja.
- Definisanje zahteva sistema kvaliteta.

Pomoću njih obezbeđujemo informacije o načinu akvizicije podataka, listi potrebnih funkcija sa njihovom važnošću - ček liste, identifikujemo korištenje na osnovu koga biramo vrstu određivanje kvaliteta. Izlazi su jasno precizirani zahtevi kvaliteta. Preciznost zahteva kvaliteta je u velikoj zavisnosti od tipa procesa određivanje kvaliteta koji će se koristiti. Ponekad je neophodno ponavljanje da bi se obezbedio potrebàn nivo detalja u zahtevima.

Specifikacija određivanja kvaliteta na osnovu obrađenih zahteva, zahteva sistema kvaliteta, entiteta određivanja kvaliteta, opšte prihvaćenih standarda obezbeđuje preko svojih podprocesa:

- Izbor metrike i indikatora,
- Izbor karakteristika,
- Definisanje nivoa ranga,
- Definisanje kriterijuma ocenjivanja.

Isto tako, oni obezbeđuju izlaze u vidu odabranih metrika i indikatora, specificiranih karakteristika i podkarakteristika ciljnog entiteta, definisanog nivoa ranga kao i kriterijuma ocenjivanja.

Proces 'Projektovanje određivanja kvaliteta' čine procesi:

- Specifikacija modula određivanja kvaliteta,
- Ugradnja modula određivanja kvaliteta u biblioteku.

Izlaz je plan određivanja kvaliteta koji se formira na osnovu definisanih metrika i indikatora za karakteristike, podkarakteristike, atomske karakteristika i kriterijuma ocenjivanja, dobijenih u prethodnom procesu.

Proces 'Primena određivanja kvaliteta' daje rezultate određivanja kvaliteta dobijene kao izlaz iz podprocesa:

- Merenje pomoću metrika,
- Ocenjivanje poređenjem,
- Izveštavanje merenja ocenjivanja,

a u skladu sa planom mjerenja definisanim u prethodnom procesu.

Proces 'Izveštavanja' sastavljen je od svojih podprocesa:

- Analiza rezultata određivanje kvaliteta,
- Generisanje izveštaja.

Na osnovu rezultata dobijenih kao izlaz iz procesa 'Primena određivanja kvaliteta' i analize tih rezultata, formiraju se razne vrste izveštaja koje su izlazi iz kompletnog sistema određivanja kvaliteta softverskog proizvoda.

Literatura:

1. Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin, 2001.,str.70-77
2. B.Jošanov,P.Tumbas, *Softverski inženjering*, VPŠ, Novi Sad, 2002.,str.14-18

8. POJAM I KOMPONENTE CASE ALATA (CASE tools concept and components)

Računarske nauke i tehnologije su se u poslednje dve decenije XX veka razvijale vrlo intenzivno u više različitih pravaca. Može se reći, kako za danas tako i za proteklo vreme, da impuls tom brzom razvoju daju svakim danom sve veće tehničke mogućnosti i bolje performanse računarskih i komunikacionih uređaja. Uslovi savremenog načina poslovanja, s jedne strane, a mogućnosti hardvera, s druge strane, diktiraju da krajnji korisnici računarskih sistema imaju sve složenije informacione zahteve. Za takve korisničke zahteve treba realizovati i odgovarajuće ***programske proizvode***, koji treba da ispunjavaju sledeće, međusobno konfliktne zahteve:

- odgovarajuća ***sveobuhvatnost***, ***aktuelnost*** i ***funkcionalnost*** s obzirom na domen primene,
- visoka ***pouzdanost*** u radu,
- dovoljno ***brz odziv*** pri davanju odgovora na informacioni zahtev korisnika,
- ***jednostavnost za upotrebu***,
- ***jednostavnost za održavanje*** i
- da budu dovoljno ***brzo realizovani***, s obzirom na trenutak identifikacije informacionog zahteva.

Vremenom se pokazalo da su se mogućnosti hardverskih uređaja povećavale prilično "paralelno", ili s manjim zaostajanjem u odnosu na porast obima i složenosti informacionih zahteva korisnika, dok su karakteristike postojećih programskih proizvoda bitno zaostajale za iskazanim potrebama korisnika i mogućnostima hardvera. Fenomen zaostajanja realizovanih programskih proizvoda, s obzirom na aktuelne potrebe i mogućnosti hardvera, dosta je rano identifikovan, a poznat je pod nazivom ***softverska kriza***.

Identifikovani su sledeći problemi, kroz koje se softverska kriza prelamala:

- najveći deo programerskog vremena je odlazio na održavanje postojećih programskih proizvoda, što je blokiralo dalji razvoj informacionih sistema i
- programiranje upotrebom jezika III generacije je bilo neefikasno i dugotrajno.

Činjenica da je najveći deo programerskog vremena odlazio na održavanje postojećih proizvoda se može potkrepiti statističkim podacima, prema kojima se 64% grešaka pri razvoju informacionog sistema pravilo u toku analize korisničkih zahteva i projektovanja informacionog sistema, dok se preostalih 36% grešaka pojavljivalo tokom realizacije informacionog sistema. Od pomenutih 64% "ranih" grešaka, svega 30% je otklanjano pre same isporuke softvera. Pri tome, kasno otkrivanje i otklanjanje grešaka iz početnih faza razvoja programskog proizvoda dovodi do eksponencijalnog rasta troškova uvođenja u upotrebu i održavanja proizvoda. Tako, na primer, otklanjanje strateške greške u fazi održavanja košta i do 100 puta više, nego ako se greška otkrije na početku rada na projektu. Ovo je dovelo do jedne "neprirodne" raspodele finansijskih sredstava, uloženih u razvoj informacionog sistema, prema kojoj preko 70% ukupnih sredstava uloženih u razvoj informacionog sistema odlazi na njegovo održavanje.

Poskupljenje održavanja i neefikasno programiranje su dovodili do velikih zakašnjenja u realizaciji projekata informacionih sistema (prema nekim podacima, veliki projekti u Sjedinjenim američkim državama su 1985. godine "kasnili" od 30 do 45 meseci). Saglasno ovim činjenicama, vremenom su identifikovani sledeći ***uzroci krize softvera***, u oblasti razvoja informacionih sistema.

1. ***Ad hoc projektovanje*** informacionog sistema, bez primene odgovarajuće metodologije, što dovodi do lošeg projekta, pojave velikog broja grešaka i prekoračenja zadatih vremenskih rokova.
2. Nepostojanje softverskih alata, koji bi ***podržali projektovanje*** informacionih sistema, ili ***automatizovali*** postupke projektovanja s dovoljnim nivoom ekspertskog znanja. Ovo, takođe, vodi ka nekvalitetnom projektu, usled otežanog rukovođenja projektom, fragmentiranog i nekonzistentnog dokumentovanja i neusaglašenosti delova projekta.

3. Nepostojanje odgovarajućih softverskih alata *za razvoj aplikacija* informacionog sistema, što vodi ka neefikasnoj realizaciji i održavanju informacionog sistema.

Rešenje krize softvera je trebalo tražiti u otklanjanju ova tri uzroka. To je vodilo ka:

- formalizaciji metodologija i tehnika projektovanja informacionih sistema i
- pojavi CASE proizvoda i jezika IV generacije, kao podrške odgovarajućim metodologijama i tehnikama.

8.1. Pojam CASE proizvoda

Drugi i treći uzrok krize softvera (nepostojanje odgovarajućih softverskih alata za podršku razvoja programskih proizvoda), kao i kompleksnost zadataka i tehnika koje se u metodologiji životnog ciklusa i u strukturiranom pristupu primenjuju, predstavlja motiv za pojavu *CASE proizvoda*. CASE predstavlja skraćenicu od engleskih reči "Computer Aided Software Engineering", što bi u prevodu značilo "računarom podržano softversko inženjerstvo" ili, slobodnije, "razvoj programskih proizvoda uz pomoć računara". CASE proizvod je bilo koji programski proizvod, namenjen za podršku, ili automatizaciju makar jednog zadatka u okviru životnog ciklusa drugog programskog proizvoda, ili je namenjen za kompletnu podršku projektovanju i realizaciji drugog programskog proizvoda.

Osnovni ciljevi primene CASE proizvoda su:

- obezbeđenje zadovoljavajućeg *kvaliteta projekta i projektne dokumentacije*,
- obezbeđenje zadovoljavajućeg *kvaliteta samog programskog proizvoda*,
- *povećanje produktivnosti* projektanata i programera,
- *skraćenje vremena* projektovanja i realizacije programskog proizvoda i
- obezbeđenje *jednostavnog i jevtinog održavanja* programskog proizvoda.

Primena metodologije životnog ciklusa i strukturiranog pristupa znače upotrebu većeg broja manje ili više formalnih tehnika i crtanja različitih dijagrama i matrica zavisnosti na različitim nivoima detaljnosti. Pri tome, izmena na jednom nivou detaljnosti često zahteva izmene i na drugim nivoima detaljnosti. Izmena na jednom dijagramu može značiti i potrebu sprovođenja izmena na više drugih dijagrama. Ukoliko je reč o projektu većeg obima, ručno projektovanje i sprovođenje ovih izmena, održavanje konzistentne projektne dokumentacije i kontrola kompletnosti projekta postaju naporan posao, s neizvesnim ishodom. Tako, na primer, o manuelnom projektovanju baze podataka ima smisla govoriti samo ako broj tipova entiteta i poveznika konceptualne šeme ne prelazi nekoliko desetina, odnosno, ako broj identifikovanih obeležja ne prelazi sto. Kada veličina konceptualne šeme prelazi ove granice, pokazuje se da problem, zbog kompleksnosti, prouzrokovane obimom, prevazilazi čovekove moći percepcije i koncentracije. Tada vreme i napor, potrebni za izradu projekta, postaju teško prihvatljivi, a kvalitet rezultata nepredvidiv. U projektu se javljaju greške u vidu: sinonima, homonima, protivrečnih ograničenja i, generalno, greške, učinjene u ranijim fazama projekta, uočavaju se tek u kasnijim fazama, kada ih je teže otkloniti. Iterativno vraćanje na ranije faze projekta u cilju otklanjanja grešaka, može dovesti do pojave novih grešaka.

Pored toga, strukturirani pristup zahteva od projektanta i programera da poseduju visoki nivo ekspertskog znanja iz oblasti softverskog inženjerstva i zadovoljavajući nivo znanja iz predmetne oblasti za koju se pravi programski proizvod, što u praksi ne mora biti uvek obezbeđeno.

Saglasno navedenim razlozima, od CASE proizvoda se očekuje da obezbede što ***viši stepen automatizacije***, prilikom izvođenja sledećih zadataka:

- vođenje dokumentacije,
- izrada dijagrama i matrica,
- konceptualno i implementaciono projektovanje šeme baze podataka,
- projektovanje programskih specifikacija aplikacija,
- izrada (generisanje) programskog koda,
- sprovođenje izmena,

- integracija parcijalnih rezultata projektovanja u jedinstvenu celinu,
- kontrola konzistentnosti, kompletnosti i kvaliteta projekta, itd.

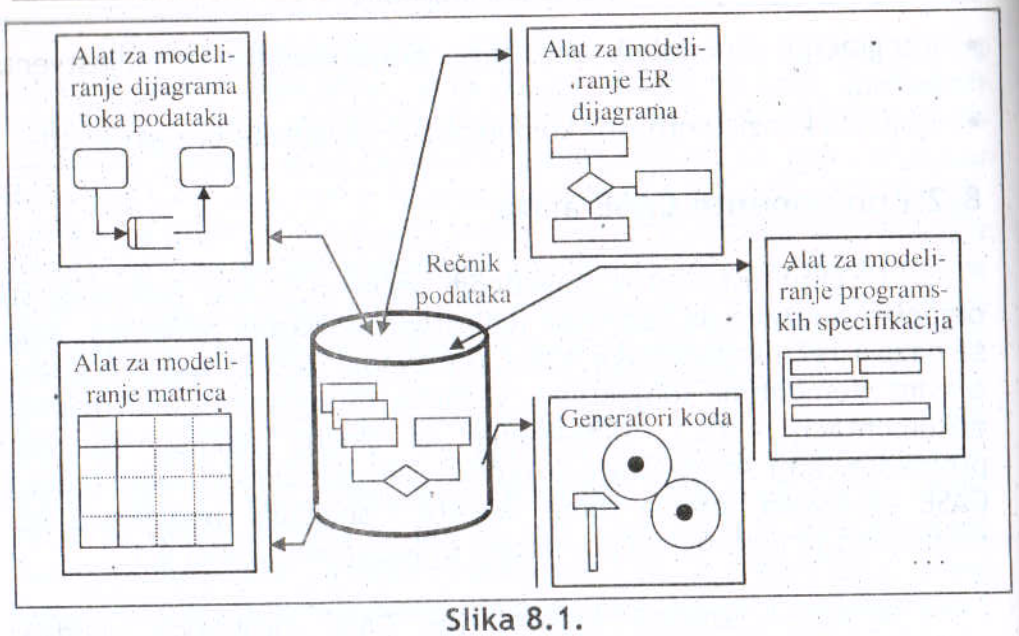
8.2. Komponente CASE alata

U cilju ostvarenja navedenih zahteva, CASE proizvodi su organizovani tako da rade nad jedinstvenom bazom podataka, koja se naziva ***rečnik podataka***[7] CASE proizvoda. Rečnik sadrži podatke o svim elementima (objektima, vezama, dijagramima, matricama, dokumentaciji, itd.), definisanim u okviru jednog, ili više projekata, koji se smeštaju u rečnik. Svi pojedinačni alati jednog CASE proizvoda, prema tome, koriste i smeštaju podatke u isti rečnik, što je ilustrovano primerom, prikazanom na slici 8.1.

Postoje različite klasifikacije CASE proizvoda. Jedna, uobičajena i ne suviše selektivna, je izvršena saglasno fazama životnog ciklusa koje CASE proizvod pokriva. Saglasno toj klasifikaciji, razlikuju se:

- ***projektantski CASE*** [8] proizvodi - namenjeni za podršku prve tri faze životnog ciklusa (strategija, snimanje i analiza i projektovanje), odnosno za podršku projektovanju programskog proizvoda,
- ***programerski CASE*** [9] proizvodi - namenjeni za podršku poslednje tri faze životnog ciklusa (programiranje, uvođenje u upotrebu, eksploatacija i održavanje), odnosno za podršku realizacije programskog proizvoda i
- ***integrisani CASE*** proizvodi - integrisani projektantski i programerski CASE proizvodi, namenjeni da podrže svih šest faza životnog ciklusa, odnosno kompletan život programskog proizvoda.

[7] Na engleskom: data dictionary, ili repository.
[8] Na engleskom: Upper CASE.
[9] Na engleskom: Lower CASE.



Slika 8.1.

Projektantski CASE proizvodi

Projektantski CASE proizvodi treba da podrže prve tri faze životnog ciklusa. U domenu projektovanja informacionih sistema, CASE proizvod koji podržava fazu strategije, može da sadrži alate za podršku:

- planiranja projekta (izbora metodologije i tehnika razvoja informacionog sistema, načina i standarda za primenu izabrane metodologije i tehnika),
- upravljanja projektom (detaljnog planiranja i izdavanja zadataka i vremenskog terminiranja projekta),
- planiranja i upravljanja resursima (materijalnim, kadrovskim i finansijskim),
- praćenja realizacije projekta i
- sprovođenja postupaka kontrole kvaliteta.

Kada je u pitanju podrška u fazi snimanja i analize, CASE proizvod može da sadrži alate za izradu:

- strukturnih modela sistema (model funkcionalne, organizacione i prostorne strukture),
- modela procesa koji se odvijaju u realnom sistemu,
- dijagrama toka podataka,

- konceptualne šeme baze podataka i
- matrica, kojima se iskazuju međuzavisnosti između elemenata konceptualne šeme baze podataka, kao i funkcionalne, organizacione, ili prostorne strukture sistema.

Za fazu projektovanja, CASE prozivod može sadržati alate za:

- prevođenje konceptualne šeme baze podataka u implementacionu šemu,
- implementaciono projektovanje šeme baze podataka, koje se može sprovoditi direktno, bez prethodne izrade i prevođenja konceptualne šeme, ili putem modifikacija prevedene konceptualne šeme,
- generisanje programskih specifikacija aplikacija (struktura menija, opisa ekranskih ili štampanih formi, podšema i standardnih procedura za upite i ažuriranje baze podataka) i
- implementaciono projektovanje programskih specifikacija aplikacija, koje se može sprovoditi direktno, bez prethodnog generisanja programskih specifikacija, ili putem modifikacija prethodno izgenerisanih programskih specifikacija.

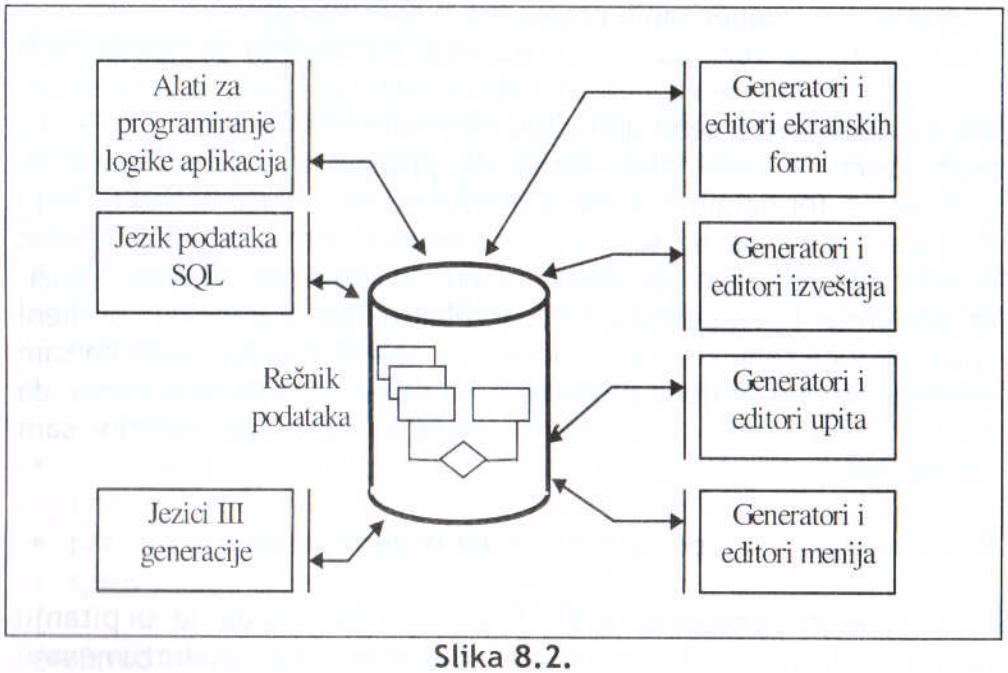
Može se reći da je pri razvoju savremenih projektantskih CASE proizvoda, sve naglašeniji zahtev da CASE sadrži "inteligentne" alate i alate koji u sebe uključuju elemente ekspertskog znanja. To prvo znači da sami alati treba da primoravaju projektanta na poštovanje formalnih pravila upotrebe odgovarajuće tehnike, koju dati alat podržava. Na taj način, projektant dobija tehničku pomoć u radu. Pored toga, na višem nivou, očekuje se da alat "pruži" projektantu i ekspertsku, tj. intelektualnu pomoć u primeni odgovarajuće tehnike. Takva pomoć se ogleda u mogućnosti da sam alat pruža odgovarajuća projektantska rešenja, ili da je u stanju da analizira, vrednuje i ocenjuje rešenja, koja je sačinio sam projektant.

Programerski CASE proizvodi i jezici IV generacije

Pod ***programerskim CASE*** proizvodima, kada je u pitanju razvoj informacionih sistema, najčešće se podrazumevaju ***generatori koda***, koji su u mogućnosti da:

- na osnovu opisa implementacione šeme baze podataka izgenerišu DDL[10] opis šeme baze podataka za konkretan sistem za upravljanje bazama podataka i
- na osnovu programskih specifikacija izgenerišu 4GL[11] programe aplikacija informacionog sistema.

Jezici IV generacije (*4GL*) predstavljaju dalju nadgradnju jezika III generacije u smislu povećanja nivoa deklarativnosti, preglednosti i lakoće programiranja. Teško je dati preciznu definiciju pojma jezika IV generacije, jer on podrazumeva široki spektar programerskih ili korisničkih alata, različitih namena i mogućnosti - od jednostavnih alata do razvijenih jezika. Zbog toga se često govori o pojmu *okruženja IV generacije*. Na slici 8.2 su prikazani mogući elementi jednog 4GL okruženja. Treba zapaziti da u takvo okruženje ulaze i jezici III generacije, što znači da ova vrsta jezika i dalje ima svoje mesto u postupku realizacije programskog proizvoda.



Slika 8.2.

[10] DDL je skraćenica za Data Definition Language, [1].
[11] 4GL je skraćenica za Fourth(4) Generation Language.

Treba napomenuti da su svi alati iz okruženja IV generacije, iz istih razloga kao i CASE proizvodi, oslonjeni na jedinstveni rečnik podataka. Šta više, težnja je da ovi alati budu integrisani s CASE proizvodima, u smislu korišćenja zajedničkog rečnika podataka, čime se obezbeđuje jedinstveno razvojno okruženje programskih proizvoda.

Generatori koda i 4GL okruženje su razvijani s ciljem da se prevaziđe neproduktivno i dugotrajno programiranje uz upotrebu jezika III genracije. Direktni pozitivni efekti primene generatora koda i 4GL okruženja se ogledaju u sledećem:

- ubrzava se i olakšava proces izrade programskog proizvoda i
- smanjuju se troškovi održavanja aplikacije, pošto je olakšano otkrivanje, pronalaženje i ispravljanje uočenih grešaka.

Posredni pozitivni efekat jeste mogućnost primene tzv. prototipskog pristupa razvoju programskih proizvoda.

Negativni efekti primene generatora koda i jezika IV generacije se ogledaju u sledećim činjenicama:

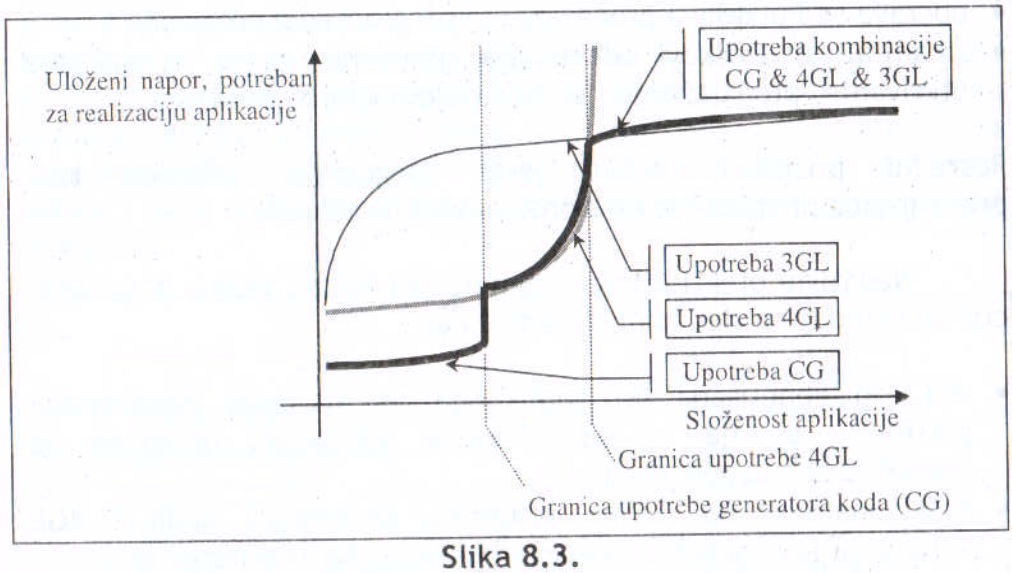
- 4GL, ili generisana 4GL aplikacija je, na istoj hardverskoj platformi, sporija od odgovarajuće aplikacije, razvijane uz pomoć jezika III generacije i
- funkcionalnost (tj. širina primene) generatora koda i 4GL okruženja je manja od funkcionalnosti jezika III generacije.

Navedene prednosti i nedostaci upotrebe generatora koda i 4GL okruženja, ukazuju da se pravilnim izborom:

- generatora koda i 4GL okruženja,
- jezika III generacije i načina povezivanja s 4GL okruženjem i
- odgovarajuće ("jače") hardverske platforme,

mogu obezbediti sve prednosti upotrebe generatora koda i 4GL okruženja, uz očuvanje dovoljno dobrih performansi izvršavanja razvijene aplikacije i dovoljno dobre funkcionalnosti za rad. To znači da se dodatnim ulaganjima u hardver i alate za razvoj aplikacija mogu postići velike uštede pri realizaciji i održavanju aplikacija informacionog sistema.

Dijagram na Slici 8.3 ilustruje problematiku funkcionalnosti generatora koda, 4GL okruženja i jezika III generacije. Pokazuje se da se manje složene aplikacije mogu direktno dobiti upotrebom generatora koda. Za složenije aplikacije je, nakon generisanja koda potrebno izvršiti dodatna prilagođavanja, upotrebom alata IV generacije, dok se vrlo složeni i dominantno proceduralni delovi aplikacija, mogu uspešno realizovati samo upotrebom jezika III generacije. Zbog toga je prethodno i naglašena potreba kombinovane upotrebe alata IV generacije i jezika III generacije.



Slika 8.3.

Uz već navedene, upotreba generatora koda ima još jedan nedostatak: ponovno generisanje aplikacije, nakon već izvršenog prilagođavanja generisanog koda putem alata IV generacije, može značiti "uništavanje" prethodno izvršenih prilagođavanja. Savremeni trendovi razvoja generatora koda idu ka tome da se ovaj nedostatak ublaži, na dva načina:

- pomeranjem granice upotrebljivosti generatora koda, tako da se putem generatora mogu napraviti i složenije aplikacije (cilj je da se pređe granica od 80% ukupno generisanog koda, koji bi bez daljih dorada bio spreman za upotrebu) i
- sistematičnim evidentiranjem dorađenih delova generisanog koda u okviru rečnika podataka, tako da svako sledeće regenerisanje uzme u obzir i postojeća prilagođenja koda.

CASE proizvodi za projektovanje šeme baze podataka

Postoje samostalni CASE proizvodi koji su isključivo namenjeni za projektovanje šeme baze podataka. Kao takvi, oni pretežno pripadaju klasi projektantskih CASE proizvoda. Ukoliko sadrže i generatore opisa šeme baze podataka, prilagođene konkretnim sistemima za upravljanje bazama podataka, tada pripadaju i klasi programerskih CASE proizvoda. Integrisani CASE proizvodi, namenjeni za razvoj informacionog sistema, obavezno moraju sadržati alate za projektovanje konceptualne, implementacione i interne šeme baze podataka.

Kada su u pitanju postupci projektovanja šeme baze podataka, CASE proizvodi za projektovanje konceptualne, implementacione i interne šeme, na današnjem nivou razvoja, najčešće omogućavaju projektovanje konceptualne šeme u ER modelu i automatsko prevođenje ER konceptualne u implementacionu šemu, zasnovanu na relacionom modelu podataka. Neki CASE proizvodi omogućavaju i projektovanje fizičke organizacije relacione baze podataka. Konačni rezultat takvog projektovanja treba da predstavlja automatski izgenerisani opis implementacione i interne šeme u jeziku podataka SQL. Obično se može generisati opis implementacione šeme, specifičan za nekoliko rasprostranjenijih relacionih sistema za upravljanje bazama podataka.

Literatura:

1. P. Mogin, I. Luković, *Principi projektovanja baza podataka*, Univerzitet u Novom Sadu, Edicija Univerzitetski udžbenik 107, Novi Sad, 2000.
2. Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin2001.
3. B.Jošanov,P.Tumbas, *Softverski inženjering*, VPŠ, Novi Sad,2002.
4. Chris Billings,*Rapid Application Development With Oracle Designer/2000*, Addison-Wesley Pub Co,1996.
5. *PROGRESS User Interface Builder*, Developers Guida, Release V8.2C

6. *PROGRESS Language Reference* : Widgets, Attributes & Methods, Events & Indexes, Release V8.2C
7. *ERwin Reference Guide*, Version V3.5.2, PLATINUM tehnology

9. CASE ALATI ZA RAZVOJ SOFTVERA U OKRUŽENJU - PRAVCI INTEGRACIJE (CASE tools development environment and directions)

CASE tehnologije obezbeđuju integralno razvojno okruženje koje je nezavisno od metodološkog pristupa u razvoju sistema i pruža podršku scim aktivnostima od definisanja, preko razvoja do održavanja softvera. Pri tome, omogućuje optimalno funkcionisanje sistema uz najmanji procenat grešaka.

CASE tehnologije se po [1,92-94], mogu na različite načine integrisati za razvoj softverskog proizvoda u okruženju, ali su najčešći sledeći pravci:

- Data Excange - Razmena podataka, je slučaj kada CASE alat izlaze koje kreira prevodi u oblik nestruktuirane datoteke u formatu štampe. To omogućuje da se ostvari zaštita podataka alata, eliminiše potreba ponovnog unošenja elemenata specifikacije dizajna i onemoguće štamparske greške. Nedostatak ovog načina je što se samo deo podataka može koristiti u drugom alatu, dok se ne definiše kompatibilnost. Takođe, stalni razvoj softvera zahteva da se i nakon manjih izmena datoteke prevode i prenose, te razne verzije mogu dovesti do odsustva sinhronizacije.
- Common Tool Access - Zajednički pristup alatima, predstavlja sledeći nivo integracije, koji omogućuje korisniku da poziva veći broj alata na isti način. U ovom okruženju, razmena podataka iz alata u alat može biti uprošćena uvođenjem procedure prevođenja običnim izborom menija ili izborom makro funkcije.
- Common Data Management - Zajedničko upravljanje podacima, omogućuje da se podaci iz različitih alata održavaju u jednoj logičkoj bazi podataka, koja može biti centralizovana ili decentralizovana. Svaki alat ima stalni i trenutni pristup uvek ažurnim podacima. Prava pristupa se mogu kontrolisati, a omogućeno je i upravljanje verzijama

alata. Postoji funkcija integrisanja podataka koja omogućuje projektantima softvera da generišu različite delove softvera i kombinuju poslove. Ako alat poseduje i karakteristiku provere, moguće je utvrditi nekonzistentnost između rezultata pojedinih projektanata. Mada se podacima iz različitih alata upravlja zajedno na zajedničkom nivou, alati ne poznaju međusobno interne strukture podataka i semantiku predstavljanja u dizajnu. I dalje je neophodan korak prevođenja koji se manuelno aktivira da bi se u jednom alatu omogućilo korišćenje izlaza drugog alata.

- Data Sharing - Podela podataka, jeste način integracije u kojem alati poseduju kompatibilne strukture podataka i semantiku i mogu se direktno upotrebljavati bez ikakve transformacije. Za realizaciju Data Sharing-a ključnu ulogu su odigrali standardi koji su osnova integracija CASE alata.
- Interoperability - Međusobna operabilnost, predstavlja najviši nivo integracije pojedinačnih alata i javlja se u slučaju kada je realizovan zajednički pristup i podela podataka. Da bi se postigla puna integrisanost CASE okruženja, neophodna su još dva elementa: upravljanje meta podacima i sredstva kontrole. Meta podaci su produkt pojedinih CASE alata i uključuju: definicije objekata, veze i zavisnosti između objekata proizvoljne detaljnosti, pravila dizajna softvera, tokove procesa, procedure i događaje. Sredstva kontrole omogućuju pojedinačnim alatima da obaveste ostatak okruženja o značajnim događajima i pošalju zahteve za akcijom ostalim alatima i servisima putem trigera. Sredstva kontrole pomažu u održavanju integriteta okruženja i obezbeđuju sredstva za automatizaciju standardnih procesa i procedura.

Literatura:

1. B.Jošanov,P.Tumbas, *Softverski inženjering*, VPŠ, Novi Sad,2002.
2. Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin2001.

10. UPRAVLJANJE PROJEKTOM (Project management)

Pre nego što se definiše pojam upravljanja projektom (projekt management), potrebno je definisati pojam projekta. Definisanjem pojma projekta olakšat će se razumevanje metodologije upravljanja projektima. Postoji mnoštvo definicija projekta:

Projekt je (vremenski ograničen) privremeni napor koji se poduzima u svrhu kreiranja jedinstvenog produkta, servisa (usluge) ili rezultata ...[12]

Projekt je sekvenca jedinstvenih, kompleksnih, i povezanih aktivnosti koje imaju zajednički cilj ili svrhu i koji mora biti završen u određenom vremenskom periodu, u okviru planiranog budžeta, i u skladu sa specifikacijama.[13]

Podvučene riječi su ključne za razumevanje koncepata upravljanja tj. rukovođenja projektima:

- svaka metodologija za razvoj sistema definiše niz (sekvencu) aktivnosti koje mogu biti obavezne ili opcionalne;
- svaki projekt je jedinstven, tj. drugačiji od svakog drugog realizovanog ili planiranog projekta;
- aktivnosti od kojih se sastoji projekt su relativno kompleksne; zahtevaju veštine i znanja koja će vam omogućiti da se prilagodite promenama uslova u kojima se izvodi projekt kao i da unapred predvidite događaje;
- svaki projekt se pokreće s određenom svrhom; najčešće je potrebno ostvariti nekoliko ciljeva kako bi se ostvarila svrha;

---

[12] *PMBOK®, 2000; "Project Management Body of Knowlwdgwe"- temeljni dokument za edukaciju i certifikaciju profesionalnih project managera; izdao.ga je Project Management Institute; a postao je i IEEE standard;*

[13] *Robert K:Wysocki, Robert Beck Jr i David B. Crane; Effective Project Management: How to Plan, Manage and Deliver Projects on Time and within Budget; New York, John Wiley & Sons, 1995.*

- zbog tržišnog pritiska da se skrati životni ciklus produkata i poslovnih procesa neophodno je da projekt bude završen u predviđenom vremenskom periodu;
- prekoračenje budžeta projekata je neprihvatljivo, bez obzira na uzroke;
- softverski produkt zbog kojeg se projekt i pokreće mora zadovoljiti očekivanja korisnika, *manager*-a kao i poslovnog okruženja;

Za razvoj bilo kojeg projekta, neophodno je učinkovito upravljanje projektom kako bi se osiguralo da projekat bude završen na vreme, u okviru prihvatljivog budžeta, ta da u potpunosti zadovolji očekivanja i zahteve korisnika. Različiti autori različito definišu upravljanje projektom:

Upravljanje projektom podrazumeva primenu znanja, veština, alata i tehnika na projektne aktivnosti radi ostvarenja cilja projekta.[14]

Upravljanje projektom je proces određivanja opsega, planiranja, organizovanja, usmeravanja i kontrolisanja razvoja prihvatljivog sistema uz minimalne troškove u specificiranom vremenskom okviru.[15]

Upravljanje projektom je aktivnost koja zalazi u sve faze svake razvojne metodologije. Većina korporacija i organizacija su u današnje vreme odbacile rigidnu hijerarhijsku strukturu i „stalne" timove. Današnja razvojna metodologija podrazumeva timove koji uključuju i tehničko i ne-tehničko osoblje, menadžere i IT stručnjake usmerene na projektni cilj. Ovakvi dinamički timovi zahtevaju rukovođenje projektom. Različite organizacije različito pristupaju upravljanju projektima (project managementu). Jedna od pristupa podrazumeva izbor menadžera projekta (project managera) iz redova osoblja već formiranog tima. Drugi, daleko popularniji pristup, podrazumeva izbor menadžera projekta (project managera) iz redova vrhunskih profesionalaca.

---

[14] PMBOK®, 2000

[15] J.L. Whitten, L.D. Bentley, K.C. Dittman "System Analysis and Desugn Methods"McGraw-Hill, 2004.

Preduslov za dobar *project management* je dobro definisan sistem razvoja procesa, stoga je iznimno važno razlikovati *process management* od *project managementa.*

Process management je aktivnost koja za cilj ima dokumentovanje, upravljanje upotrebom, te poboljšanje izabrane metodologije za razvoj sistema.

Zapravo, *process management* ima za cilj uspostavljanje kvalitativnih standarda koji će biti primenjivi na sve projekte. Drugim rečima, opseg *process management* su svi potencijalni projekti, a opseg *project management*-a je samo jedan konkretan projekt.

Uzroci neuspeha projekata

Upravljanje projektom (project management) se smatra uspešnim ako:

- rezultira produktom prihvatljivim za krajnjeg korisnika;
- ako je sistem (softver) završen u okvirima predviđenog budžeta;
- ako proces razvoja softvera ima minimalan uticaj na tekuće poslovne operacije.

Ne zadovoljavaju svi projekti ove kriterijume, a kao posledica tog nisu ni svi projekti uspešni. Broj neuspešnih ili projekata s ograničenim uspehom daleko nadmašuje broj uspešnih projekata. Najveći broj neuspeha projekata može se pripisati nedovoljno edukovanim rukovodiocima tj. menadžerima projekta. Da bi se razvila svest o važnosti *project management*-a proučimo neke od problema *project management*-a kao i posledice tih problema:

- nemogućnost motivisanja višeg menadžmenta za projekt: tokom izvođenja projekta motivisanost rukovodstva se menja; neretko više rukovodstvo u potpunosti izgubi interes za dati projekt;

- izostanak organizacijske posvećenosti metodologiji razvoja sistema;
- nepoštovanje svih koraka koje propisuje metodologija razvoja: projektni timovi često izostavljaju pojedine korake u razvojnom metodologiji zbog kašnjenja projekta, prekoračenja budžeta ili zbog nedostatka veština i znanja pojedinih članova tima;
- neopravdana očekivanja: i korisnici i *management* imaju određena očekivanja od projekta; tokom vremena ova očekivanja se mogu promeniti, bilo da očekivanja od sistema rastu pa se time utiče na prekoračenje budžeta i vremenskih rokova, bilo da dođe do nekontrolisanog dodavanja nove tehnike i tehnologije;
- preuranjeno usvajanje budžeta i rasporeda: prve procene troškova i neophodnog vremena za završetak projekta su retko ispravne; ove procene je potrebno izvesti tek nakon detaljne analize problema i zahteva;
- neadekvatne tehnike procene: mnogi sistem analitičari rade procene tako da izvrše proračune, a zatim dobiveni rezultat pomnože sa 2, što se teško može smatrati naučnim pristupom;
- neopravdani optimizam: sistem analitičari i menadžeri projekta su skloni neopravdanom optimizmu;
- *mythical man-month*: ukoliko projekt kasni, voditelji projekta često uključuju nove ljude u tim; ne postoji linearna relacija između vremena i broja osoblja; dodatno osoblje sa sobom donosi nove probleme - otežanu komunikaciju, što dovodi do još većeg kašnjenja;
- neadekvatne menadžerske veštine: menadžeri su skloni uživati prava koja sa sobom donosi položaj menadžera, ali nisu spremni preuzeti i obaveze koje takav položaj donosi; ovaj problem je lako uočiti - naizgled niko nije odgovoran, korisnici ne znaju kako projekt napreduje, tim ne održava redovne sastanke, članovi tima ne komuniciraju međusobno...
- nemogućnosti prihvatanja promena u poslovanju: ukoliko se značaj projekta tokom njegovog izvođenja promeni, ili ukoliko dođe do reorganizacije menadžmenta ili poslovanja neophodno je prilagoditi raspored po kojem se izvodi projekt;

- nedostatak resursa: ovo može biti posledica neadekvatne tehnike procene ili posledica toga što raspoređeno osoblje ne raspolaže neophodnim veštinama i iskustvima.

I na posletku, najčešći razlog neuspeha projekata je **nedostatna edukovanost *project manager*-a**. Prepoznajući ovaj problem Project Management Institute, profesionalno društvo za vođenje razvoja i sertifikaciju profesionalnih *project manager*-a je kreirao "Project Management Body of Knowlwdgwe" (PMBOK) koji je temeljni dokument za edukaciju i sertifikaciju profesionalnih *project manager*-a. PMBOK je postao i IEEE standard 1998. godine.

Uspešan *project manager* bi trebao posedovati veliki broj osobina. Neke od tih osobina se stiču edukacijom, a neke iskustvom. Međutim, sve te osobine se temelje na dve osnovne premise:

- prvo: menadžer ne može upravljati procesima s kojima se susreće prvi put;
- drugo: menadžeri moraju razumeti kontekst projekta kojim upravljaju (oblik poslovanja i poslovna kultura);

Osobine neophodne uspešnom *project manager*-u se mogu grupisati u sledeće kategorije:

- poslovna postignuća,
- sposobnost rešavanja problema,
- sposobnost vršenja uticaja,
- sposobnosti upravljanja ljudima,
- doslednost.

Funkcije project managemenet-a

Različiti teoretičari proučavaju i redefiniraju osnovne funkcije *project management*-a već dugi niz godina. Funkcije *project management*-a uključuju određivanje opsega, planiranje, procenjivanje, organizovanje, pravljenje rasporeda, usmeravanje, kontrolisanje i zatvaranje projekta.

- određivanje opsega: definisanjem opsega definiraju se okviri projekta; menadžer projekta mora odredit opseg očekivanja učesnika, odrediti troškove i upravljati očekivanjima;
- planiranje: planiranjem se identifikuju neophodne zadatke u cilju ostvarenja projekta;
- procenjivanje: potrebno je proceniti svaki zadatak unutar projekta, Koliko će biti potrebno vremena? Koliko ljudi će biti uključeno? Koje su neophodne veštine koje osoblje mora posedovati? Kojim redom je potrebno izvršavati zadatke? Koliko će koštati? Preklapaju li se neki zadaci?
- pravljenje rasporeda: menadžer projekta je odgovoran za raspoređivanje aktivnosti; pri pravljenju rasporeda potrebno je voditi računa o neophodnim zadacima, trajanju tih zadataka i preduslovima koje je neophodno ispuniti;
- organizovanje: menadžer projekta se mora pobrinuti da svaki član tima razume svoju ulogu i odgovornosti u okviru projekta, kao i neophodnost redovnog izveštavanja;
- usmeravanje: nakon što je projekt započeo menadžer projekta mora koordinirati, savetovati, ocenjivati i nagrađivati članove projektnog tima;
- kontrolisanje: kontrolisanje je verojatno jedan od najtežih funkcija menadžera projekta, zato što mora nadzirati proces napretka, poštovanje rasporeda i troškova, te praviti prilagođavanja kada je to nephodno;
- zatvaranje: svaki dobar menadžer projekta po završetku projekta nešto nauči, bilo iz svog uspeha bilo iz svog neuspeha;

Project management alati i tehnike

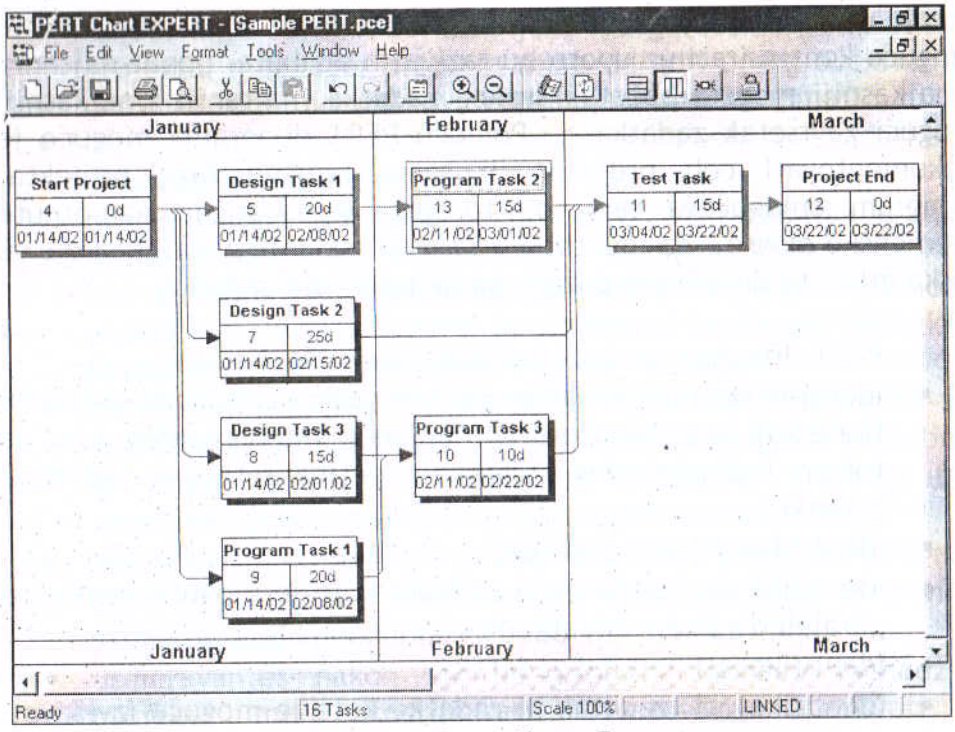
PMBOK između ostalog definiše i alate i tehnike koju služe kao podrška *project manager*-u. Dva najčešće spominjana alata su PERT i Gantt dijagram.

PERT (Project Evaluation and Review Technique) je razvijen 1950 - tih za potrebe planiranja i kontrolisanja velikog projekta razvoja oružja za US mornaricu. PERT dijagram je grafički mrežni model koji prikazuje projektne zadatke, te njihove međusobne veze. Osim toga, dijagram definiše kritičnu putanju koja se sastoji od kritičnih zadaaka koje je potrebno izvršiti na

vreme da bi se projekt okončao u predviđenom roku. Dijagram je moguće konstruirati uz upotrebu različitih atributa, poput najraniji i najkasniji mogući početak svakog zadatka, najraniji i najkasniji mogući završetak zadatka ... Pomoću PERT dijagrama moguće je dokumentovati celi projekat ili samo ključne faze projekta. Dijagram omogućava timu da izbegne nerealistične nametnute vremenske okvire, da identificira i skrati trajanje zadataka koje su "uska grla", te da usmere pažnju na najkritičnije zadatke.

PERT dijagram se može kreirati u nekoliko koraka:

- identifikovati sve zadatke: pri tom vodite računa da uključite ljude koji raspolažu relevantnim znanjima o projektu tako da tokom *brainstorming* sastanka identifikujete sve potrebne zadatke;
- identifikovati prvi zadatak;
- identifikovati zadatak ili zadatke koje je moguće izvršavati paralelno s prvim zadatkom;
- identifikovati naredni zadatak;
- identifikovati zadatak ili zadatke koje je moguće izvršavati paralelno s drugim zadatkom;
- nastaviti proces identifikovanja dok ne pobrojite sve zadatke;
- odrediti trajanje zadataka: u konsultaciji s članovima tima predvidite vreme trajanja za svaki zadatak;
- konstruisati PERT dijagram: numerisati sve zadatke, povezati ih strelicama i dodati karakteristike svakog zadatka (predviđeni datum početka i predviđeni datum završetka)
- određivanje kritične putanje: kritična putanja uključuje one zadatke koje se moraju početi ili završiti u određenom roku kako bi se izbeglo kašnjenje celog projekta; kritična putanja je obično označena crvenom bojom;
- softver: većina komercijalnih *project manegement* softvera rutinski generiše PERT dijagram;



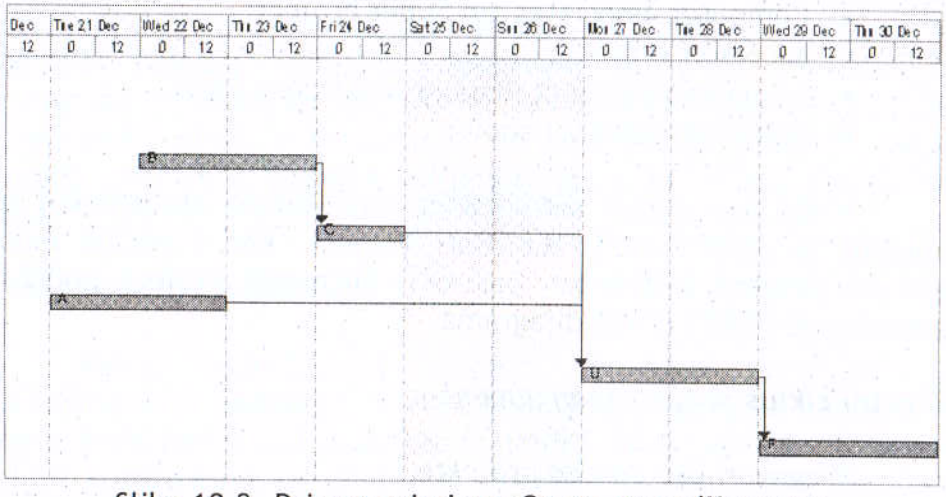
Slika 10.4. Pert Chart Expert

Gantt dijagram

Gantt dijagram, kojeg je 1917. godine koncipirao Henry L. Gantt je najčešće korišten alat za pravljenje rasporeda aktivnosti i evaluaciju napretka projekta. Gantt dijagram omogućava procenu trajanja projekta, prikazuje redosled kojim će se zadaci izvršavati, olakšava upravljanjem međusobnim zavisnostima među zadacima, determiniše neophodne resurse i pruža trenutni uvid u postignuća.

Gantt dijagram se sastoji od horizontalne osi koje predstavlja vreme (podeljeno u manje vremenske odsečke: dan, sedmica, mesec) i vertikalne ose koja predstavlja zadatke od kojih se sastoji projekt. Pojedini zadaci se mogu "preklapati". Vertikalna crta se koristi za predstavljanje datuma podnošenja izveštaja.

Task Name	Duration	Text Above	Start	Finish	Predecessors	Resource Names
B	2 days	Yes	Wed 22.12.0<	Thu 23.12.0<		
C	1 day	Yes	Fri 24.12.0<	Fri 24.12.0<	2	
A	2 days	Yes	Tue 21.12.0<	Wed 22.12.0<		
D	2 days	Yes	Mon 27.12.0<	Tue 28.12.0<	4;3	
F	2 days	Yes	Wed 29.12.0<	Thu 30.12.0<	5	



Slika 10.2. Primer primjene Gantt-ovog dijagrama

Gantt dijagram i PERT dijagram se ne isključuju međusobno. Gantt dijagram je efikasniji ako tražite raspored izvođenja i dužinu trajanja pojedinih zadataka, PERT dijagram je efikasnije oružje ako proučavate međusobne zavisnosti zadataka.

10.1. Upravljanje životnim ciklusom softvera (Software lifecycle management)

Project management softver

Softver za upravljanje projektom se rutinski koristi kao podrška menadžerima projekta u planiranju projekata, pravljenju rasporeda, koncipiranju budžeta, nadgledanju napretka i troškova, generisanju izveštaja i praćenju promena.

Najčešće korišteni softverski paketi su:

- Applied Business Technologies' Results Management Including Project Workbench;
- Artemis Management Systems' 7000 and 9000;
- Computer Associates' CA-Super Project;
- Microsoft Project;
- Primavera's Project Planner and Monte Carlo
- Scitor's Project Scheduler

Među navedenim softverskim proizvodima studentskoj populaciji je najdostupniji Microsoft Project. Kao i većina *management project* softverskih paketa i Microsoft Project podržava generisanje PERT i Gantt dijagrama.

Životni ciklus project mangement-a

1. *Pregovori oko opsega projekta*

Pregovori oko opsega projekta su verojatno najvažniji preduslov za uspešan projekat. Sve strane uključene u projekt se moraju složiti oko opsega projekta i to pre nego pokušaju identifikovati i rasporediti zadatke ili dodeliti resurse. Opseg definiše očekivanja od projekta, a očekivanja određuju stupanj zadovoljenja i stupanj uspešnosti. Pregovori oko opsega projekta su neophodna aktivnost u životnom ciklusu *project managemnet*-a. Što je opseg? Opseg definiše okvire (granice) projekta - delove poslovanja koje treba prostudirati, analizirati, dizajnirati, konstruisati i naposletku poboljšati.

Odgovori na pet osnovnih pitanja su ključni za određivanje opsega projekta:

1.1. Proizvod - Šta želimo?
1.2. Kvalitet - Koliko kvalitetan proizvod želimo?
1.3. Vreme - Kada želimo da proizvod bude gotov?
1.4. Trošak - Koliko želimo platiti za proizvod?
1.5. Resursi - Koje resurse ćemo ili možemo ponuditi?
1.6. Usmeravanje timskih napora
1.7. Nadgledanje i kontrola progresa
1.8. Procena rezultata projekta

Pregovori o navedenim pitanjima su aktivnosti koja treba rezultirati dokumentom - *statement of work* (SOW)- koji opisuje celi posao koji će se obaviti tokom projekta. Ovaj dokument je postao uobičajeni ugovor između konsultanta i klijenta. SOW potvrđuje da je *project manager* shvatio ko je odgovorna osoba, ko kontoliše, kakva je formalna i neformalna organizacija s kojom će se projekt razvijati, ko su ključne osobe, kao i ostale netehničke pojedinosti. Ovim dokumentom se uspostavlja poslovni odnos između *project manager*-a, klijenta, ali i proširenog projektnog tima.

2.Identifikacija zadataka

Nakon određivanja opsega projekta pristupa se identifikovanju zadataka. Identifikovanjem zadataka zapravo se identifikuje posao koji je potrebno uraditi. Faze razvoja softvera su suviše kompleksne za planiraje i pravljenje rasporeda, te ih je potrebno razlagati na manje celine, na aktivnosti i zadatke, sve dok se ne dobiju zadaci kojima je lako upravljati. Neki eksperti preporučuju da se aktivnosti razlažu sve dok se ne dobije količina posla koju je moguće obaviti za dve sedmice ili manje. Aktivnosti i zadaci nisu čvrste sturkture, tj. većina metodologija dopušta naknadne izmene. Vrlo popularan alat za identifikaciju o dokumentovanju projektnih aktivnosti je *work breakdown* struktura (WBS). WBS je grafički alat kojim se predstavlja hijerarhijska dekompozicija projekta u faze, aktivnosti i zadaci. Moguće je u WBS uključiti specijalne zadatke - *milestones*. *Milestones* su događaji koji značajno doprinose ili zaokružuju najveće isporuke tokom projekta.

3.Procena trajanja zadataka

Na temelju WBS *project manager* mora proceniti trajanje svakog zadatka. Trajanje svakog zadatka zavisi od niza faktora kao što su veličina tima, broj korisnika, sklonosti korisnika, kompleksnost poslovnog sistema, arhitektura informacijske tehnologije, zauzetost članova tima na nekim drugim projektima, te iskustva članova tima. Većina metodologija za razvoj sistema ne samo da definiše zadatke već nudi i osnovne smernice za procenu trajanja zadataka. *Project manager* mora na temelju ovih smernica koje daje metodologija izvršiti prilagođavanje procene.Npr. u Microsoft Project-u sve faze, aktivnosti i zadaci se nazivaju jednostavno zadaci. Međutim, razlikuje sumarne i primitivne zadatke. Sumarni zadaci se sastoje od više drugih zadataka. Primitivni zadatak ne sadrži u sebi niti jedan drugi zadatak.

Pri proceni trajanja zadatka važno je uzeti u obzir dva faktora:

- Efikasnost - niti jedan radnik nije 100% efekasan; potrebno je uračunavati pauze za ručak, pauze za kafu, učestvovanje u drugim projektima, vreme provedeno u proveri email-a, u neobaveznom ćaskanju u kancelariji i sl.
- Prekidi - radnike prekidaju telefonski pozivi, posetioci i drugi neplanirani zadaci koji ometaju koncentraciju i oduzimaju vreme koje bi se provelo na radu na projektu.
  Na taj način dobijamo definiciju da radnici rade u proseku 75% na projektu, a 25% vremena protekne u ostalim aktivnostima.

Postoji mnoštvo tehnika za procenu trajanja zadataka:

- Procena minimalne količine vremena potrebne za obavljanje zadatka: tzv. optimistična procena pri kojoj se ne uzimaju u obzir nikakvi prekidi niti ometanja;
- Procena maksimalne količine vremena potrebne za obavljanje zadatka: tzv. pesimistična procena pri kojoj se polazi od pretpostavke da će sve krenuti loše;
- Procena očekivanog trajanja zadatka: ova tehnika nije naprosto sredina između optimistične i pesimistične procene;

pokušava se identifikovati (ne sve, već) najčešći uzroci kašnjenja;

- Najverovatnije trajanje: koje se računa po formuli:

$$D = \frac{(1 \cdot optimisti\ čnaprocena) + (4 \cdot očekivana\ rocena + (1 \cdot pesimisti\ čnaprocena)}{6}$$

4. Specificiranje međusobne zavisnosti zadataka

Nakon procene trajanja zadataka počinje se pravljenje rasporeda. Raspored ne zavisi samo od trajanja već i od međusobne zavisnosti zadataka. Postoje četiri oblika međusobne zavisnosti zadataka:

- *Finish-to-start* - završetak jednog zadatka znači početak sledećeg zadatka;
- *Start-to-start* - početak jednog zadatka znači početak drugog zadatka u isto vreme;
- *Finish-to-finish* - dva zadatka se moraju završiti u isto vreme;
- *Start-to-finish* - početak jednog zadatka podrazumeva završetak drugog zadatka.

Nakon određivanja datuma početka projekta, zadatak koji je tokom projekta potrebno izvršiti, međusobne zavisnosti zadataka, te trajanja pojedinih zadataka, pristupa se pravljenju rasporeda. Postoje dva pristupa za pravljenje rasopreda:

- *Forward*: odredi se datum početka projekta, a zatim se pravi raspored za aktivnosti koje slede; datum završetka projekta se izračunava na temelju procene trajanja pojedinih zadataka, njihove međusobne zavisnosti, te raspoloživosti resursa potrebnih za izvršenje predviđenih zadataka;
- *Reverse*: odredi se datum završetka projekta (dedline), a zatim se od tog datuma unatrag pravi raspored aktivnosti;

Poput većine projekt mangement alata, Microsoft Project može generisati raspored na temelju procene trajanja zadataka i njihove međusobne zavisnosti.

5. Dodeljivanje resursa

Nakon pravljenja rasporeda vreme je da se razmišlja o neophodnim resursima. Resurse delimo u sledeće kategorije:

- Ljudi - uključujući vlasnike sistema, korisnike, analitičare, dizajnere, spoljne saradnike, ljude koji rade na razvoju i asistente;
- Usluge - pregled kvaliteta rada i sl.
- Kapaciteti i oprema - prostorije i tehnologije koje se koriste u projektu;
- Sredstva i materijali - papir, olovke, laptopi, printeri i sl.;
- Novac - uključuje izražavanje svih gore navedenih stavki u novcu.

Raspoloživost resursa, osobito ljudi, može značajno izmeniti projekat. Većina metodologija identifikuje *resurs-ljudi* u obliku uloga. Uloga nije isto što i posao. Najčešće se uloga objašnjava poređenjem sa "šeširom" kojeg neko nosi zbog svojih veština. Svaki pojedinac može nositi više "šešira" , ali i više pojedinaca može posedovati veštine neophodne za nošenje "šešira". Posao projekt managera je da dodeli odgovarajuće ljude odogovarajućoj ulozi. Neke, od mogućih uloga su: revizor, poslovni analitičar, administrator baze podataka, mrežni administrator, projekt manager, programer....

6. *Usmeravanje timskih napora*

Sve prethodne faze mogu se smatrati pripremom za izvođenje projekta. U ovoj fazi se pristupa izvršavanju planiranih aktivnosti. Većina autora se slaže u konstataciji da su najteži deo posla *management*-a - ljudi, te većina literature nudi savete za *project manger*-e[16]:

- budite dosledni,
- osigurajte podršku,
- ne dajte obećanja koja ne možete održati,
- javno hvalite, kritikujte nasamo,
- budite svesni moralnih konsekvenci svojih postupaka,
- postavljajte realistične rokove,
- postavljajte dostižne ciljeve,

[16] Keith London, The People Side of System

- ohrabrujte timki duh.

7.Nadgledanje i kontrola progresa

Pri izvođenju projekta obaveza je *project manager*-a da kontroliše projekt, tj. da nadgleda napredak i poštovanje rasporeda, budžeta i opsega projekta. Menadžer mora raditi izveštaje o napretku, te ako je potrebno prilagođavati se situaciji. **Izveštaji o napretku** se moraju praviti dovoljno često kako bi se uspostavila kontrola, ali ne prečesto kako ne bi postali kontraproduktivni. Izveštaji mogu biti u usmenoj ili pismenoj formi. Izveštaji moraju biti iskreni i precizni čak i kada vesti nisu povoljne. Osim toga, u izveštajima je neophodno identifikovati uspeh, ali i probleme pre nego što eskaliraju u katastrofu. Osim redovnog izveštavanja u ovoj fazi često dolazi do nekontrolisane promene opsega projekta. Promene u opsegu su najčešće razlog nesporazuma između klijenata i organizacije koja izvodi projekt. Neizbežnost promena opsega iziskuje formalnu strategiju za rešavanje ovog problema. ***Change management*** je formalna strategija za uspostavljanje procesa koji će olakšati upravljanje promenama u toku projekta. **Očekivanja management-a** su još jedan od problema s kojima se projekt manager mora nositi. Svaki projekt ima cilj i svoja ograničenja kada su u pitanju troškovi, kvalitet, opseg i raspored. U idealnim uslovima navedeni parametri bi bili optimizirani. Neretko, *managemenet* upravo to i očekuje: optimalne uslove i rezultate. Kao pomoć *project manager*-ima razvijena je **matrica očekivanja *management*-a** - alat koji pomaže razumevanje dinamiku i uticaje promena parametara na krajnji ishod projekta. Kašnjenja u izvršavanju projekta nisu neobična pojava. Međutim, neki zadaci su osjetljiviji na kašnjenja. Kao posledica tog *project manger*-i su prinuđeni razmatrati kritičnu putanju i vremenske prekide koji se pojavljuju u toku izvođenja projekta. Potrebno je naglasiti zadatke kritične putanje i ako je potrebno privremeno preusmeriti resurse sa zadataka koji su (iz bilo kojih razloga) trenutno obustavljeni na kritične zadatke kako bi se što je moguće više poštovao raspored.

8. Procena rezultata projekta

Project manager-i moraju učiti na svojim greškama! Posljednja faza životnog ciklusa *project manager* prikuplja povratne informacije od članova tima i klijenata o njihovim iskustvima rada na projektu i

predlozima o poboljšanju, tj. pravi prikaz projekta. Prikaz projekta bi trebao sadržavati odgovore na sledeća pitanja:

- Je li proizvod zadovoljio očekivanja korisnika?
- Je li je projekt završen na vreme?
- Je li projekt izvršen u okviru budžeta?

Na temelju odgovora na ova pitanja prave se izmene u metodama razvoja i *project management*-a koje će se primenjivati u budućnosti.

Literatura:

1. L. Whitten, L.D. Bentley, K.C. Dittman; *"System Analysis and Design Methods"*, McGraw-Hill, 2004;
2. http://www.mapnp.org/library/plan_dec/project/project.htm#anchor440783
3. http://www.swebok.org/
4. http://www.columbia.edu/~jm2217/
5. http://www.netmba.com/operations/project/pert/
6. http://egweb.mines.edu/eggn491/Information%20and%20Resources/pmbok.pdf

11. POUZDANOST SOFTVERA (Software reliability)

Optimisti misle da jednom nakon što pokrenu softver i on radi ispravno, da će raditi ispravno zauvek. Serije tragedija i haos izazvanih softverskom nepouzdanošću potvrđuju suprotno:

- Softver može imati male neprimetne greške koje mogu kulminirati u katastrofu. 25. februara, 1991.god. tokom *Golf War*, sakupljajući greške koje su propuštale 0.000000095 sekundi precizno svake 10-e sekunde, akumulirajući 100 sati, načinio je Patriot projektil grešku u prekidanju izbacivanja projektila. 28 života je bilo izgubljeno.
- 1991. god., nakon izmene 3 linije koda u signalizirajućem problemu koji sadrži milione linija koda, lokalni telefonski sistem u Kaliforniji se zaustavio.
- Jednom perfektno pokrenuti softver može se takođe pokvariti ako se pokrene u promenjenom okruženju. Nakon uspešnosti rakete *Ariane 4*, novi let *Ariane 5* je završen u vatri dok su greške bile otkrivene u kontrolnom softveru.

Ovo čini da se zapitamo ako je softver iole pouzdan, trebamo li ga koristiti u *safety-critical* ugrađenim aplikacijama. Vi možete uništiti svoju odeću ukoliko je softver u mašini pogrešan; 50% šansi je da budete sretni ukoliko ATM mašina pogrešno izračuna vaš novac; ali u avionima, srčanim *pace-maker*ima, mašinama za radioaktivno zračenje, softver može lako uzeti ljudske živote. Sa procesorima i softverom prožimajući *safety-critical* ugrađeni softver, pouzdanost softvera je jednostavno pitanje života i smrti. Jesmo li ugradili potencijalnu katastrofu kad smo ugradili softver u sisteme?

Ključni koncepti

Prema ANSI, ***Software Reliability*** (SR - pouzdanost softvera) je definisana kao: ***verovatnoća softver operacija bez greške za određen vremenski period u određenom okruženju.*** lako je *Software Reliability* definisana kao funkcija verovatnoće i dolazi u

11. POUZDANOST SOFTVERA (Software reliability)

Optimisti misle da jednom nakon što pokrenu softver i on radi ispravno, da će raditi ispravno zauvek. Serije tragedija i haos izazvanih softverskom nepouzdanošću potvrđuju suprotno:

- Softver može imati male neprimetne greške koje mogu kulminirati u katastrofu. 25. februara, 1991.god. tokom *Golf War*, sakupljajući greške koje su propuštale 0.000000095 sekundi precizno svake 10-e sekunde, akumulirajući 100 sati, načinio je Patriot projektil grešku u prekidanju izbacivanja projektila. 28 života je bilo izgubljeno.
- 1991. god., nakon izmene 3 linije koda u signalizirajućem problemu koji sadrži milione linija koda, lokalni telefonski sistem u Kaliforniji se zaustavio.
- Jednom perfektno pokrenuti softver može se takođe pokvariti ako se pokrene u promenjenom okruženju. Nakon uspešnosti rakete *Ariane 4*, novi let *Ariane 5* je završen u vatri dok su greške bile otkrivene u kontrolnom softveru.

Ovo čini da se zapitamo ako je softver iole pouzdan, trebamo li ga koristiti u *safety-critical* ugrađenim aplikacijama. Vi možete uništiti svoju odeću ukoliko je softver u mašini pogrešan; 50% šansi je da budete sretni ukoliko ATM mašina pogrešno izračuna vaš novac; ali u avionima, srčanim *pace-maker*ima, mašinama za radioaktivno zračenje, softver može lako uzeti ljudske živote. Sa procesorima i softverom prožimajući *safety-critical* ugrađeni softver, pouzdanost softvera je jednostavno pitanje života i smrti. Jesmo li ugradili potencijalnu katastrofu kad smo ugradili softver u sisteme?

Ključni koncepti

Prema ANSI, ***Software Reliability*** (SR - pouzdanost softvera) je definisana kao: ***verovatnoća softver operacija bez greške za određen vremenski period u određenom okruženju.*** lako je *Software Reliability* definisana kao funkcija verovatnoće i dolazi u

notaciji vremena, moramo zapaziti da različito od hardverske pouzdanosti, softverska pouzdanost nije direktna funkcija vremena. Elektronski i mehanički delovi mogu postati „stari" i vremenom i korištenjem „izaći iz mode", ali softver neće „zahrđati" ili „izaći iz mode" tokom svog životnog veka. Softver se neće vremenom promeniti izuzev ako se hotimično promeni ili nadogradi.

SR je važan atribut kvaliteta softvera, zajedno sa funkcionalnošću, korisnošću, performansama, mogućnostima, instalacijom, izdržljivošću i dokumentacijom. SR je težak za realizaciju, jer kompleksnost softverske težnje je visoka. Dok bilo koji sistem sa visokim stepenom kompleksnosti, uključujući i softver, teško dostiže izvesni nivo pouzdanosti, sistem programeri guraju kompleksnot u softverske *layere* sa rapidnim porastom veličine sistema i lakoće nadogradnje softvera.

Na primer, velika sledeća generacija aviona će imati preko jedan milion *source* linija softvera na ploči, sledeća generacija kontrolnog sistema vazdušnog saobraćaja će sadržati između jednog i dva miliona linija; nadolazeći internacionalni *Space Station* će imati preko dva miliona linija na ploči i preko deset miliona linija osnovnog podržavajućeg softvera; nekoliko glavnih *life-critical* odbrambenih sistema će imati preko pet miliona *source* linija softvera. Dok je kompleksnost softvera indirektno vezana sa SR, direktno je vezana sa drugim važnim faktorima u kvalitetu softvera, naročito funkcionalnosti, mogućnosti i slično. Naglašavanje tih prednosti vodi ka višoj kompleksnosti softvera.

Mehanizmi softverskih grešaka

Softverske greške mogu biti sigurne greške, dvosmislenosti, propusti ili nedostatak interpretacije specifikacije. Softver bi trebao ublažiti nehat ili nekompetentnost u pisanju koda, u neadekvatnim testovima, u netačnoj ili neočekivanoj upotrebi softvera ili drugih nepredvidivih problema.

Dok je primamljivo crtati analogiju između hardverske i softverske pouzdanosti, softver i hardver imaju osnovne razlike koje ih čine različitim u *failure* mehanizmima. Hardver greške su

najčešće fizičke greške, dok su softver greške, *design* greške koje su teže za vizuelizaciju, klasifikaciju, otkrivanje i ispravak.

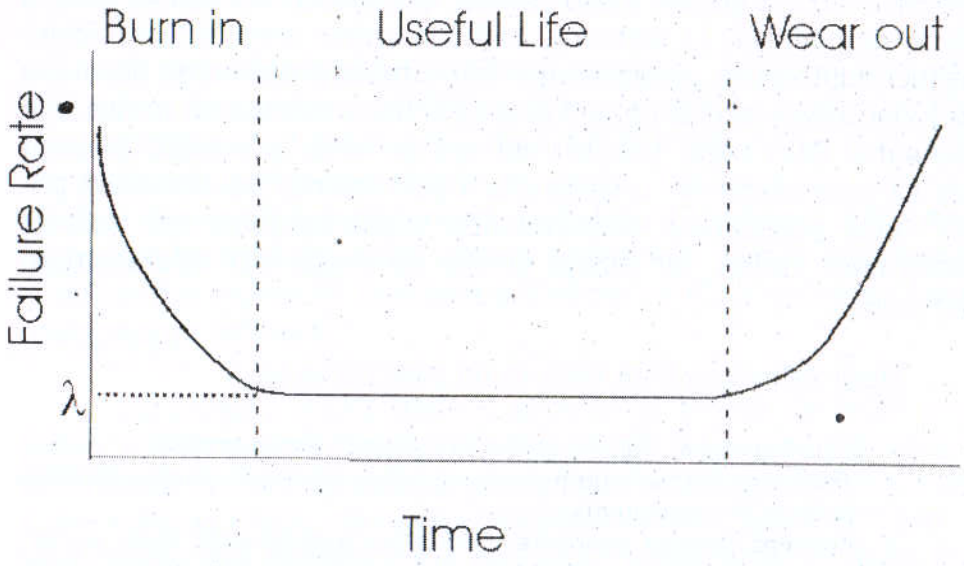
Dizajn greške su usko vezane za nejasne ljudske faktore i dizajn procese, za koje mi nemamo solidno razumevanje. U hardveru, dizajn greške mogu takođe egzistirati, ali fizičke greške obično dominiraju. U softveru, teško možemo naći striktni odgovarajući duplikat za „proizvodnju" kao proces proizvodnje hardvera ako jednostavna akcija upload-ovanja softver modula na mesto nije uračunata. Radi toga, kvalitet sofvera se neće promeniti jednom kada ga upload-ujemo u spremište i pokrenemo. Pokušavajući postići veću pouzdanost jednostavnim dupliciranjem, isti softver moduli neće raditi, jer dizajn greške ne mogu biti oglašavanjem otkrivene.

Sledi delimična lista faktora od značaja za SR:

- **Uzrok grešaka**: Softver greške su uglavnom dizajn greške.
- **Trošivost**: Softver nije povezan sa fazom trošivosti. Greške se mogu pojaviti bez upozorenja.
- **Koncept sistema oporavka**: Periodično restartovanje može pomoći pri ispravci softverskih problema.
- **Vrijeme nezavisnosti i životni ciklus**: Softverska pouzdanost nije funkcija radnog (operativnog) vremena.
- **Faktori okruženja**: Ne deluju na SR osim ukoliko može delovati na programske inpute.
- **Predviđanje pouzdanosti**: Softverska pouzdanost ne može biti predviđena ni iz kakve fizičke osnove, sve dok je u potpunosti zavisan od ljudskih faktora u dizajnu.
- **Redundantnost**: Ne može se poboljšati SR ako se koriste identične softverske komponente.
- **Interfejs**: Softverski interfejsi su čisto konceptualni - ne vizuelni.
- **Failure rate motivator**: Obično nije predvidljiv iz analiza podeljenih činjenica.
- **Izgradnja sa standardnim komponentama**: Dobro razumevanje i prošireno testiranje standardnih delova pomaže poboljšanju održavanja i pouzdanosti. Ali u softver industriji, nemamo te trendove vidljive. Kod će se ponovo koristiti nakon nekog vremena, ali u veoma limitiranom okruženju. Striktno govoreći, nema standardnih delova za softver, osim nekih standardiziranih logičkih struktura.

Kriva softverske pouzdanosti

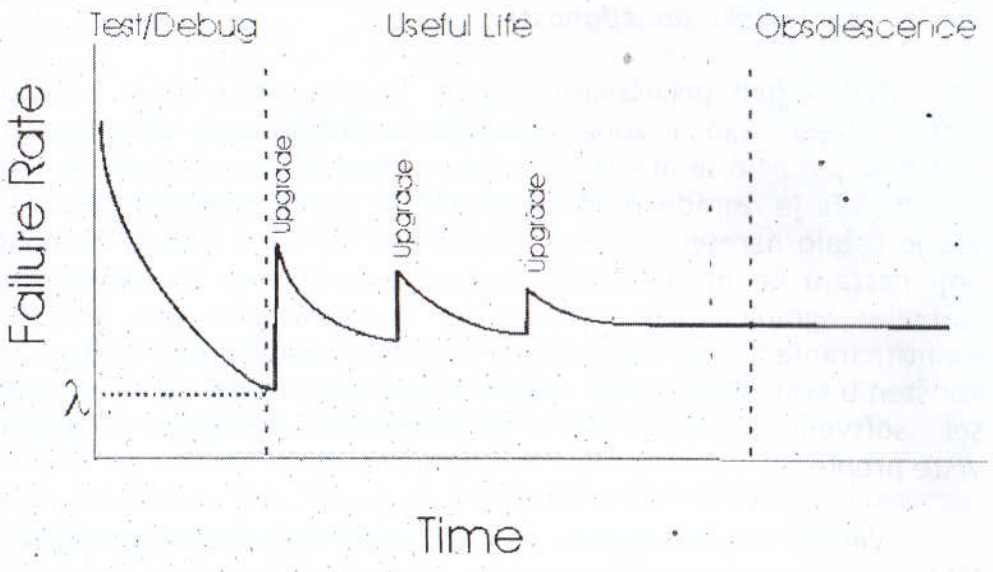
Kroz vreme, prikazani su odnosi hardver grešaka na Slici 11.1, poznatoj kao „*bathtub*" (kada) kriva. Period A, B i C stoje u fazama „burn-in", „useful life" i „end-of-life".



Slika 11. 1. „*Bathtub*" kriva za pouzdanost hardvera

Softverska pouzdanost, bilo kako, ne prikazuje iste karakteristike kao hardverska. Moguća kriva je prikazana na slici 11.2., ukoliko projektujemo softversku pouzdanost na istim osama. Tu su dve glavne različitosti između hardverske i softverske krive:

- Jedna je razlika da u poslednjoj fazi, softver nema povećavajući odnos grešaka, kao hardver. U toj fazi, softver se približava starosti; tada nema motivacije za bilo kakvu softversku nadogradnju ili promenu.
- Druga razlika je u „useful-life" fazi - softver će doživeti drastično povećanje u odnosu grešaka svaki put kada se napravi neka nadogradnja.



Slika 11.2. Revidirana „*bathtub*" kriva za softversku pouzdanost

Nadogradnje na Slici 11.2. pokazuju osobine nadogradnje, ne-nadogradnje za pouzdanost. Za prednosti, kompleksnost softvera bi se mogla povećavati dok se funkcionalnost softvera povećava. Čak i *bug fix* može biti uzrok softverskim greškama, ako one uzrokuju druge defekte softvera. Za nadogradnje pouzdanosti, moguće je nastajanje "spuštanja u odnosu softver grešaka", ako su ciljevi nadogradnje povećanje pouzdanosti softvera, kao što su redizajn ili reimplementacija nekih modula, koristeći bolje inženjerske pristupe kao što je *clean-room* metod.

11.1. Modeli, tehnike i metrike SR-a

Dok je SR jedan od najvažnijih aspekata kvaliteta softvera, *Reliability Engineering* zahtevi su takođe prakticirani u softver polju. *Software Reliability Engineering* (SRE) u opsežnim studijama operacionog ponašanja sistema zasnovanih na softveru, sa uvažavanjem korisničkih zahteva odnosi se na pouzdanost, [IEEE95].

Modeli softverske pouzdanosti

Kako ljudi pokušavaju shvatiti karakteristike kako i zašto softver greši, kao i zbog pokušaja kvantificiranja pouzdanosti softvera, pojavilo se mnoštvo modela softverske pouzdanosti. Preko 200 modela je izgrađeno od ranih 1970-ih, ali kako kvantificirati SR još je ostalo nerešeno. Mnogi modeli kao što su ti modeli i mnogi koji nastaju ne mogu postići zadovoljavajući nivo kompleksnosti softvera; ograničenja i pretpostavke su napravljene radi procesa kvantificiranja. Zbog toga, nema ni jednog modela koji može biti korišten u svim situacijama. Jedan model može raditi dobro za neki set softvera, ali može biti u potpunosti neupotrebljiv za druge vrste problema.

Većina softver modela sadrži sledeće delove: pretpostavke, faktore i matematičke funkcije koje povezuju pouzdanost sa faktorima. Matematička funkcija je obično većeg reda, eksponencijalna ili logaritamska.

Tehnike softverskog modeliranja mogu biti podeljene u dve podkategorije: modeliranje predviđanja i modeliranje procene. Oba tipa tehnika su zasnovane na posmatranju i akumuliranju pogrešnih podataka i analiziranje statističkih zaključaka.

Glavne razlike ova dva modela su prikazane u Tabeli 11.1.

Tabela 11.1. Razlike između modela predviđanja i modela procene

Teme	Modeli predviđanja	Modeli procene
Odnos podataka	Koristi istorijske podatke	Koristi podatke iz tekućeg softver razvojnog okruženja.
Kada je korišten u krugu razvijanja	Obično pravi prioritet za izgradnju ili test fazu; može biti korišten ranije kao koncept faza.	Obično se koristi kasnije u životnom krugu (nakon što neki podaci budu prikupljeni); ne tipična je upotreba u konceptu ili fazi izgradnje.

Okvirno vreme	Predviđa softversku pouzdanost u neko buduće vreme.	Procenjuje SR u neko skoro ili buduće vreme.

Koristeći modele predviđanja, SR može biti ranije predviđena u razvojnoj fazi i poboljšanjima može biti započeto usavršavanje pouzdanosti.

Reprezentativni modeli procene uključuju eksponencijalne distribucione modele, Weibull distribucione modele, Thompson i Chelson modele i td. Eksponencijalni modeli i Weibull distribucioni modeli su obično znani kao klasični *broj grešaka/odnos grešaka* modeli procene, dok Thompson i Chelson modeli pripadaju *Bayesian* modelima procene grešaka.

Područje je doseglo tačku kada softver modeli mogu biti primenjivani u praktičnim situacijama i davati značajne rezultate, i drugo, nema nijednog modela koji je najbolji u svakoj situaciji. Zbog kompleksnosti softvera, bilo koji model mora imati ekstra pretpostavke.

Mnogi SR modeli ignorišu softver razvijački proces i fokusiraju se na rezultate - uočene greške i/ili kvarove. Radeći tako, kompleksnost je reducirana i apstrakcija je postignuta, bilo kako, modeli teže da budu prihvaćeni u delovima situacija i eventualnim problemima. Mora se pažljivo odabrati pravi model u specifičnom slučaju. Dalje, rezultatima modeliranja ne može se slepo verovati i ne mogu se slepo prihvatiti.

SR merenja

Merenje je uobičajeno na drugim inženjerskim poljima, ali ne i u softver inženjeringu. Iako frustrirajuće, pitanje kvantifikacije SR nikada nije odbačeno. Do sada, još nije nađen dobar način merenja softverske pouzdanosti

Merenje SR ostaje težak problem jer nemamo dobro razumevanje prirode softvera. Ne može se naći odgovarajući način

za merenje softverske pouzdanosti. Čak i očiti podaci za merenje kao softverska veličina nemaju jedinstvenu definiciju.

Primamljivo je merenje nekih srodnih pouzdanosti ako se pouzdanost ne može meriti direktno. Tekuća praksa mjerenja SR može biti podeljena u 4 kategorije:

1. Podaci za merenje

Veličina softvera je uslovljena kompleksnošću, razvijačkim naporom i pouzdanošću. Linije koda (LOC) ili linije koda u hiljadama (KLOC), su zahtevane za merenje veličine softvera. Ali, ne postoji standardni način merenja. Tipično, *source* kod je korišten i tumačen, a drugi ne-izvršni iskazi nisu ni izračunati. Ovaj metod ne može uverljivo uporediti softvere koji nisu napisani u istom jeziku. Pojava novih tehnologija ponovnog korištenja koda i tehnika generisanja koda takođe uliva sumnju na jednostavne metode.

Function point metric je metod merenja funkcionalnosti predložene softver izgradnje zasnovan na brojanju ulaza, izlaza, master fajlova, istraživanja i interfejsa. Metod može biti korišten za procenu veličine softver sistema pre nego što te funkcije mogu biti identifikovane. To je mera funkcionalne kompleksnosti programa. Meri funkcionalnost poslatu korisnicima i nezavisna je od programskog jezika. Primarno korištena za biznis sisteme; nije dokazana u naučnim ili *real-time* aplikacijama.

Kompleksnost je direktno vezana za softversku pouzdanost i kao takva veoma je važna. *Complexity-oriented metrics* je metod determinisanja kompleksnosti programske kontrolne strukture pojednostavljujući kod u grafičkoj reprezentaciji. Reprezentativna mera je *McCabe's Complexity Metric.*

Test coverage metrics su načini procene grešaka izvodeći testove na softver proizvodima, bazirani na pretpostavkama da je softverska pouzdanost (SR) funkcija dela softvera koji je uspešno verificiran ili testiran.

2. Podaci za projekt menadžment

Istraživači su shvatili da dobar menadžment može rezultirati boljim produktima. Istraživanje je pokazalo da postoji veza između procesa izgradnje i sposobnosti komplementiranja projekata na vreme i unutar željenih stvarnih kvaliteta. Troškovi se povećavaju kada kreatori softvera koriste neadekvatne procese. Veća pouzdanost može biti postignuta koristeći bolje procese za kreiranje, *risk* menadžment procese, konfiguracijske menadžment procese i td.

3. Procesi merenja

Bazirani na pretpostavkama da je kvalitet produkata u direktnoj vezi sa procesom, procesi merenja mogu biti za procenu, nadgledanje i poboljšanje pouzdanosti i kvaliteta softvera. ISO-9000 certifikat ili „standardi menadžment kvaliteta" je generička referenca za familiju standarda razvijenih od strane *International Standards Organization* (ISO).

4. Mere grešaka i neuspeha

Cilj prikupljanja mera grešaka i neuspeha je biti u mogućnosti odrediti kada je softver blizu izvršavanja bez grešaka. Minimalno, oba broja i grešaka, nađenih kroz testiranja, i neuspeha (ili drugih problema) dobivenih od strane korisnika nakon dostavljanja, su sakupljeni, sumirani i analizirani kako bi se postigao taj cilj. Test strategija se odnosi na delotvornost mera grešaka jer ako scenario testiranja ne pokriva punu funkcionalnost softvera, softver može proći sve testove i biti podložan greškama. Obično, mere neuspeha su bazirane na informacijama mušterija zahvaljujući krahovima nađenim nakon pokretanja softvera. Sakupljeni podaci o neuspehu su zbog toga korišteni za kalkulisanje veličine neuspeha.

Tehnike poboljšanja softverske pouzdanosti

Dobre inženjerske metode mogu poboljšati softversku pouzdanost.

Pre razvijanja softverskih proizvoda, testiranje, verifikacija i validacija su potrebni koraci. Softver-testiranje se koristi za lociranje i otklanjanje softver-kvarova. Testiranje je još u svojoj početnoj fazi. Različiti alati za analize kao što su trend analize, *fault-tree* analize, formalne metode i td. mogu takođe biti

korištene za minimiziranje mogućnosti pojave kvara nakon što se softver pokrene i nakon toga poboljša softverska pouzdanost.

Nakon razvijanja softver produkata, polje podataka može biti analizirano u studiji ponašanja softver-kvarova. Tolerancija grešaka ili tehnike predviđanja grešaka/neuspeha bit će korisne tehnike i imaće vodeće uloge u minimiziranju pojave grešaka ili krahiranja u sistemu.

Softverska pouzdanost i kvalitet softvera

SR je deo softverskog kvaliteta. To se odnosi na mnoga područja gde je važan kvalitet softvera:

1. Tradicionalna/hardverska pouzdanost

Inicijalno pitanje u studiji SR zasnovano je na analogiji tradicionalne i hardverske pouzdanosti. Mnogi koncepti i analitičke metode koje se koriste u tradicionalnoj pouzdanosti mogu biti takođe korištene za procenu i poboljšanje SR. Bilo kako, SR se fokusira na dizajn perfekciju više nego na proizvođačku perfekciju kako je to kod tradicionalne/hardverske pouzdanosti.

2. Tolerancija grešaka softvera

Tolerancija grešaka softvera je važan deo sistema sa visokom pouzdanošću. To je način rukovanja nepoznatim i nepredvidivim softver (ili hardver) krahovima (ili greškama), nudeći set funkcionalnih ekvivalentnih softver modula razvijenih od strane različitih i nezavisnih produkcijskih timova. Pretpostavka je različit dizajn softvera koji je težak za ostvarivanje.

3. Testiranje softvera

Testiranje softvera je način merenja i poboljšanja SR. Ima važnu ulogu u dizajnu, implementaciji, validaciji i fazi realizacije. Ovo područje još nije potpuno usavršeno. Napredak u tom polju će imati veliki uticaj u softverskoj industriji.

4. Socijalne i legalne zabrinutosti

Kako softver ulazi u svaku poru svakodnevnog života, problemi vezani za softver i kvalitet softver proizvoda mogu uzrokovati ozbiljne probleme kao što je nesreća Therac-25. Greške u softveru su značajno drukčije nego što su u hardverskim ili softverskim

komponentama sistema: one su obično dizajn greške i mnoge od njih su vezane za probleme u specifikaciji. Neizvodivost kompletnog testiranja softver modula komplikuje problem jer softver bez grešaka ne može biti garantovan. Nebitno koliko se mi trudili, softver produkti ne mogu biti ostvareni bez grešaka. Gubici uzrokovani softver greškama uzrokuju sve više socijalnih i legalnih zabrinutosti.

SR je ključni dio kvaliteta softvera. Studija SR može biti kategorizirana u tri dela:

- Modeliranje,
- Merenje,
- Poboljšanje.

SR modeliranje je doseglo tačku kada značajni rezultati mogu biti dobiveni prihvatanjem prikladnih modela u problemu. Postoji mnogo modela, ali nijedan model ne može prihvatiti važnu vrednost softverskih karakteristika. Pretpostavke i apstrakcije moraju biti načinjene da pojednostave problem. Nema nijednog modela koji je univerzalan za sve situacije.

Merenje SR je prirodno. Merenje je odavno uobičajeno u softveru kao i na drugim inženjerskim poljima: „Koliko je dobar softver, kvantitativno". Jednostavno pitanje, ali još nema dobrog odgovora. SR ne može biti direktno merena, tako da se drugi srodni faktori mere za procenu SR i poređenje među proizvodima. Proces razvijanja, nađene greške i kvarovi su faktori vezani za softversku pouzdanost.

Poboljšanje SR je teško. Problem dolazi zbog nedovoljnog razumevanja SR i generalno, nerazumevanja karakteristika softvera. Do sada, nije bilo načina savladavanja kompleksnosti problema softvera. Kompletno testiranje kompleksnih softverskih modula je nemoguće. Softver produkt bez grešaka ne može biti garantovan. Realne stege vremena i budžeta pojedinačno ograničavaju napore da se SR poboljša.

Kako sve više softver polako ulazi u ugrađene sisteme, moramo ih osigurati od ugrađenih katastrofa. Garantovanje

softverske pouzdanosti nije lak zadatak. Kako je problem težak, obećavanje progresa je još uvek prema povećanju softverske pouzdanosti. Mnoge standardne komponente i procesi su predstavljeni na polju softverskog inženjerstva.

Literatura:

1. http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/#reference#reference

12. IZRADA SOFTVERSKOG PROIZVODA - DEMO PRIMER
(Making software product - study example)

U ovom poglavlju je dat jedan kompletan seminarski rad, kao demo primer.

Primer[17]:

Dizajn poslovne logike slučaja korišćenja studentske službe - /Prijem izveštaja o raspodeli predmeta po profesorima za tekuću školsku godinu/

Struktura rada:

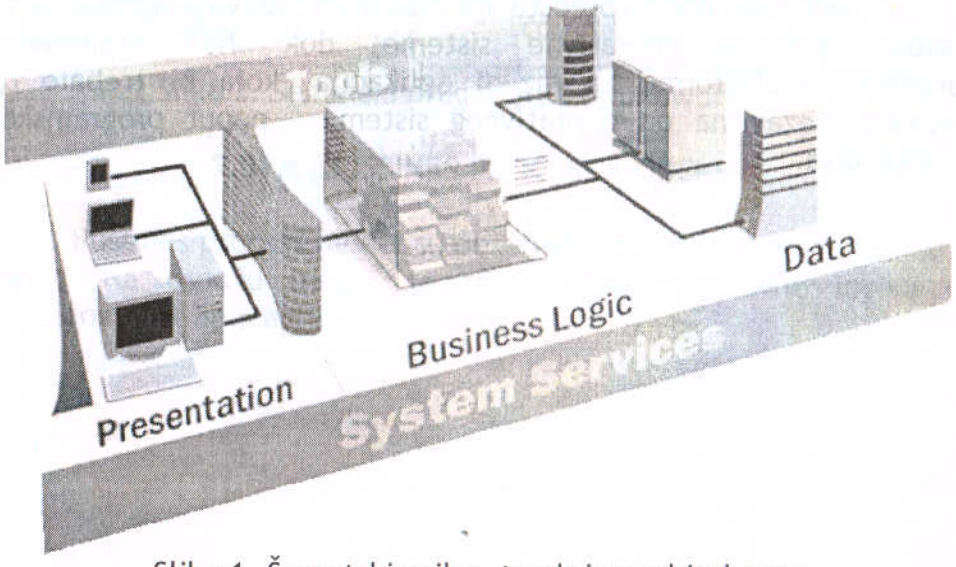
[17] *Seminarski rad studenta Aleksandra Jankovića, predmet: Softversko inženjerstvo, rađen na Tehničkom fakultetu u Zrenjaninu 2004. godine; pod mentorstvom Prof.dr Dragice Radosav.*

UVOD

Savremene tendencije načina dizajniranja i implementacije kako informacionih sistema, tako i software-a uopšte, definitivno se kreću u pravcu primenjivanja troslojnog koncepta realizacije aplikacija.

Dok je prvi , odnosno prezentacioni sloj okrenut korisniku, to jest korisničkom interface-u, drugi - srednji sloj predstavlja kičmu software-a. On zapravo izvršava poslovnu logiku sistema u celosti, obuhvata sve funkcije sistema i oslanja se na treći sloj, bazu podataka.

Ovakav savremeni pristup dizajniranja i implementacije informacionog sistema upotrebljen je, upravo u ovom radu prilikom realizacije slučaja korišćenja studentske službe, pod nazivom "Prijem izveštaja o raspodeli predmeta po profesorima za tekuću školsku godinu".



Slika 1. Šematski prikaz troslojne arhitekture

1. ZADATAK

Treba relaizovati dizajn poslovne logike slučaja korišćenja studentske službe, pod nazivom "Prijem izveštaja o raspodeli predmeta po profesorima za tekuću školsku godinu". Po realizaciji

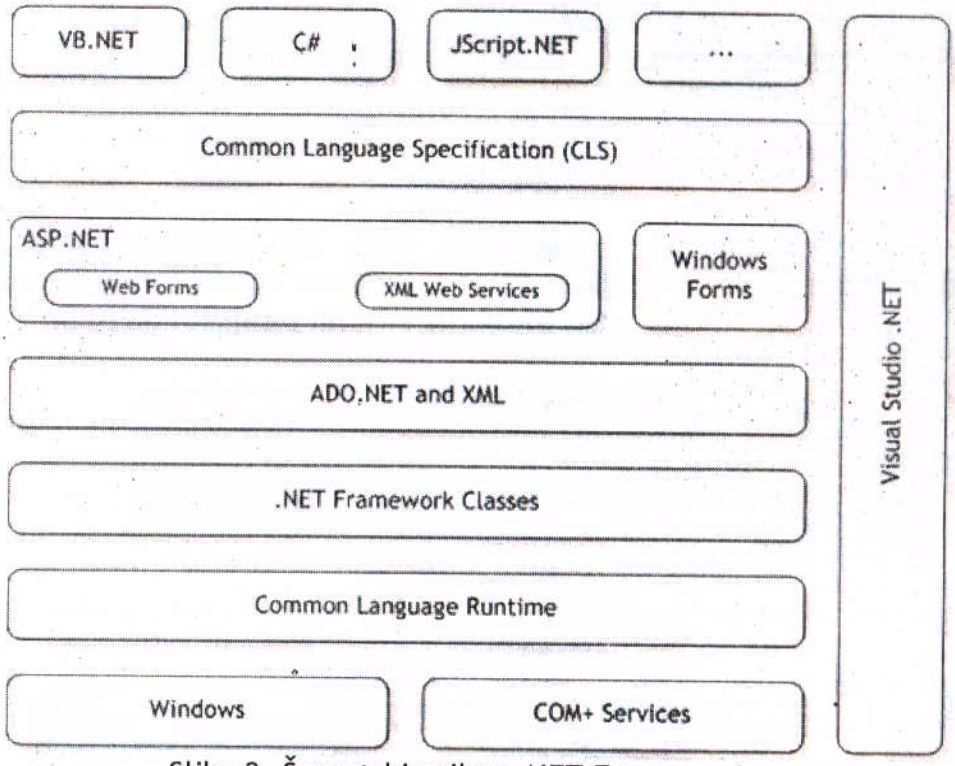
dizajna treba napraviti aplikaciju koja će raditi na osnovu urađene poslovne logikê. Svi elementi pristupa bazi podataka biće sadržani u srednjem sloju. kojem i aplikacija treba da se obraća u komunikaciji sa bazom podataka.

Za dizajn poslovne logike korišćen je alat Power Designer 9.5.

Preporuka za realizaciju srednjeg sloja i aplikacije je da se koristi nova Microsoft - ova tehnologija koju ozančava naziv ".net" i Microsoft-ov Visual Studio 2003, kao alat za razvoj. Takođe se preporučuje da programski jezik bude C#. Fakultet poseduje licencu za korišćenje istih.

Tehnologija ".net", direktno se oslanja na framework (nova verzija je 1.1) koji omogućava izvršavanje, kako poslovne logike, tako i aplikacije.

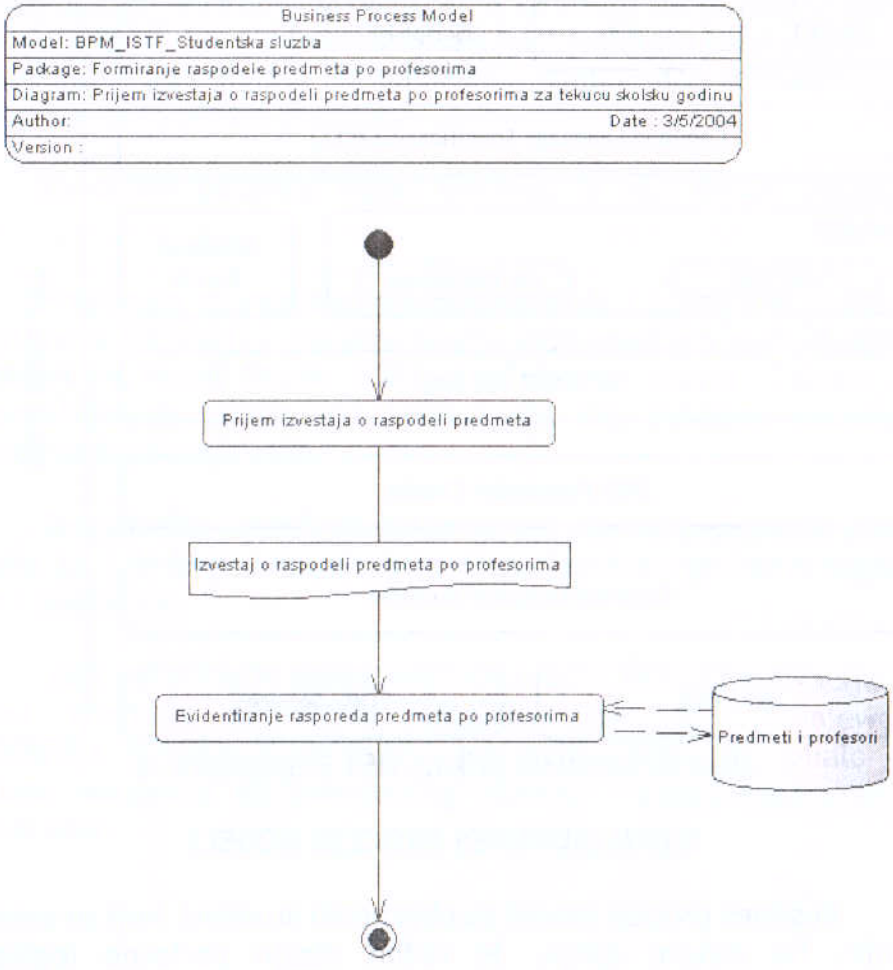
.NET tahnologija predstavlja novi način razvoja aplikacija za (sada) Windows operativne sisteme, dok .NET Framework predstavlja platformu za razvoj aplikacija, koja bi trebalo da postane nezavisna od operativnog sistema - poput programskog jezika Java.



Slika 2. Šematski prikaz .NET Framework -a

3.BPM (BUSSINES PROCESS MODEL)

Bussines process model su obezbedili studenti koji su snimali stanje. Na osnovu njega, je rađen dizajn poslovne logike i funkcionalnost aplikacije.



Slika 3. Dijagram Bussines Process Model

Procesi na dijagramu:

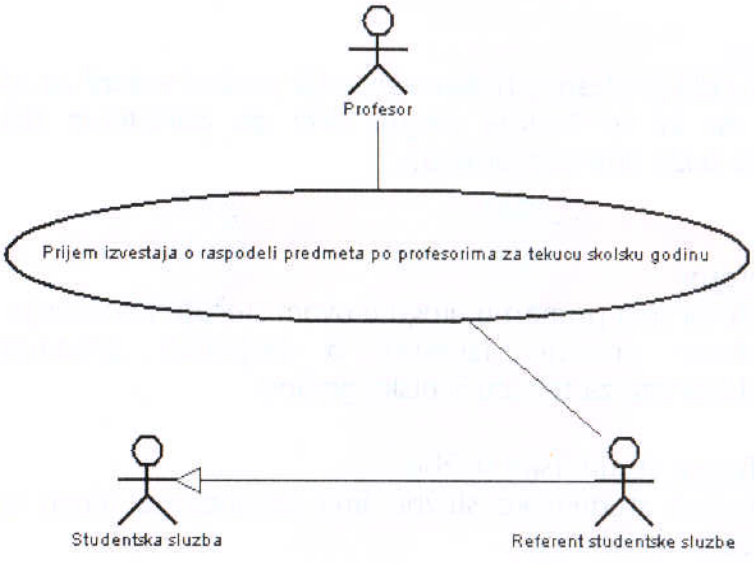
- Prijem izveštaja o raspodeli predmeta:
  U studentsku službu od strane profesora stiže izveštaj o raspodeli predmeta po profesorima za tekuću školsku godinu.
- Izveštaj o raspodeli predmeta po profesorima:
  Ovaj izveštaj sadrži zaduženja profesora za tekuću školsku godinu.
- Evidentiranje rasporeda predmeta po profesorima:

Referent studentske službe unosi podatke o rasporedu predmeta po profesorima u bazu "Predmeti i profesori".

4. USE CASE

Slučaj korišćenja (use case) služi za opis ponašanja, odnosno za definisanje funkcije sistema sa tačke gledišta korisnika sistema, pri čemu se ne razmatra interna struktura sistema.

Object-Oriented Model	
Model: OOM_ISTF_Studentska sluzba	
Package: Osnovne studije (Raspodela predmeta po profesorima)	
Diagram: Prijem izvestaja o raspodeli predmeta po profesorima za tekucu skolsku godinu	
Author: Aleksandar Jankovic, Vladimir Jelic	Date : 3/4/2004
Version :	



Slika 4. Dijagram Use Case
/Slučaj korišćenja Prijem izveštaja o raspodeli predmeta po profesorima za tekuću školsku godinu/

Specifikacija:

U studentsku službu od strane profesora stiže izveštaj o raspodeli predmeta po profesorima za tekuću školsku godinu.

Na osnovu rasporeda po predmetima koji prave profesori, referent studentske službe unosi podatke o rasporedu predmeta po profesorima u bazu "Predmeti i profesori".

Ukoliko neki profesor predaje iste predmete kao i ranijih godina onda se u bazu ne unosi ništa. Registruju se samo predmeti koje neki profesor po prvi put predaje, novouvedeni predmeti na fakultetu koji se povezuju sa profesorima, i novi profesori. Ukoliko je neki profesor otišao u penziju podaci vezani za njega se ne brišu.

Napomena:
Ukoliko je neki profesor otišao u penziju podaci vezani za njega se ne brišu da bi se kasnije moglo doći do određenih podataka (brisanje iz baze nije dozvoljeno).

Uloge:

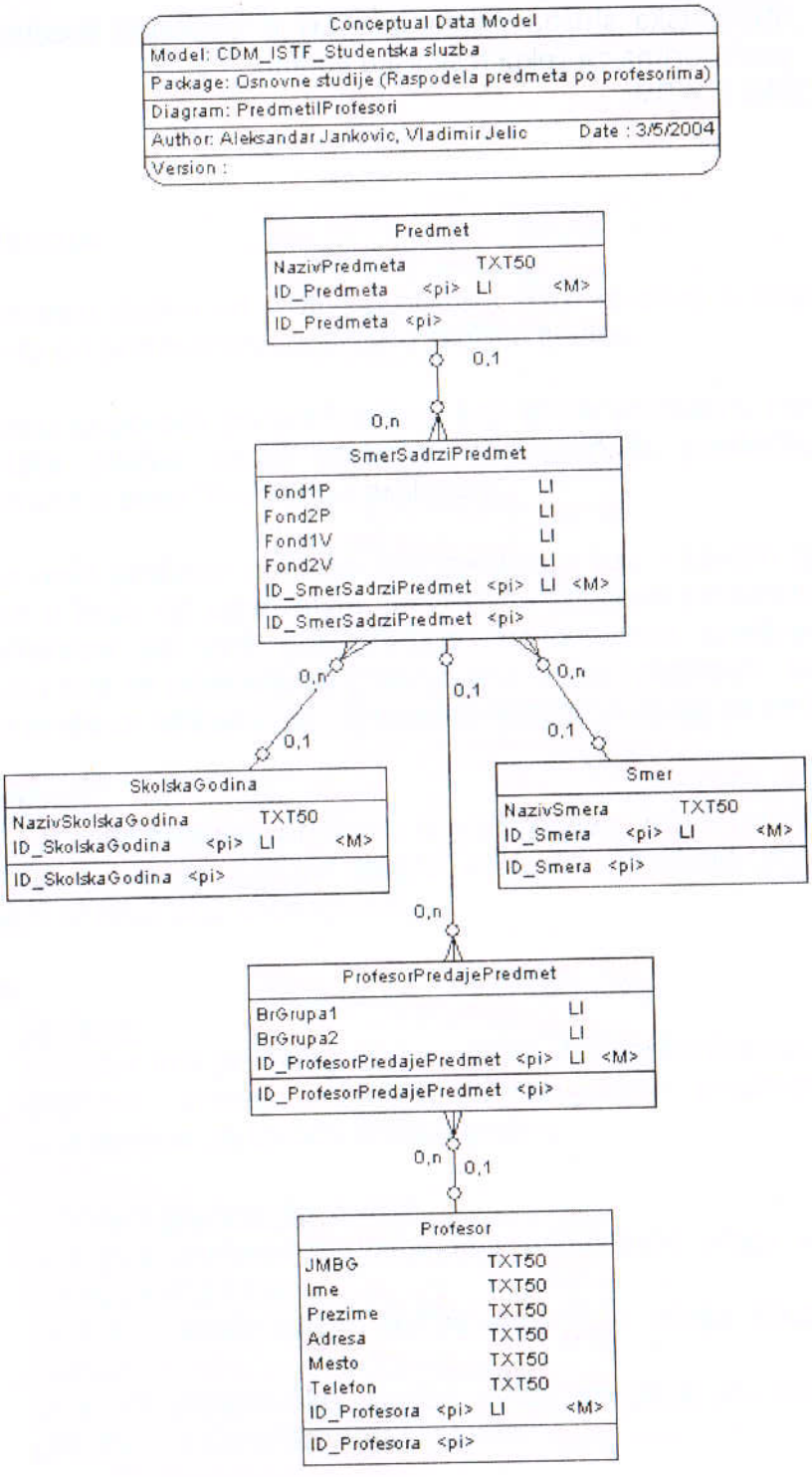
- Profesor
  Profesor ima primarnu ulogu u ovom slučaju korišćenja.
  Profesor predaje izvestaj o raspodeli predmeta po profesorima za tekuću školsku godinu.

- Referent studentske službe
  Referent studentske službe ima sekundarnu ulogu u ovom slučaju korišćenja.
  Referent studentske službe nasleđuje ulogu Studentska služba.
  Referent studentske službe unosi podatke o rasporedu predmeta po profesorima u bazu.

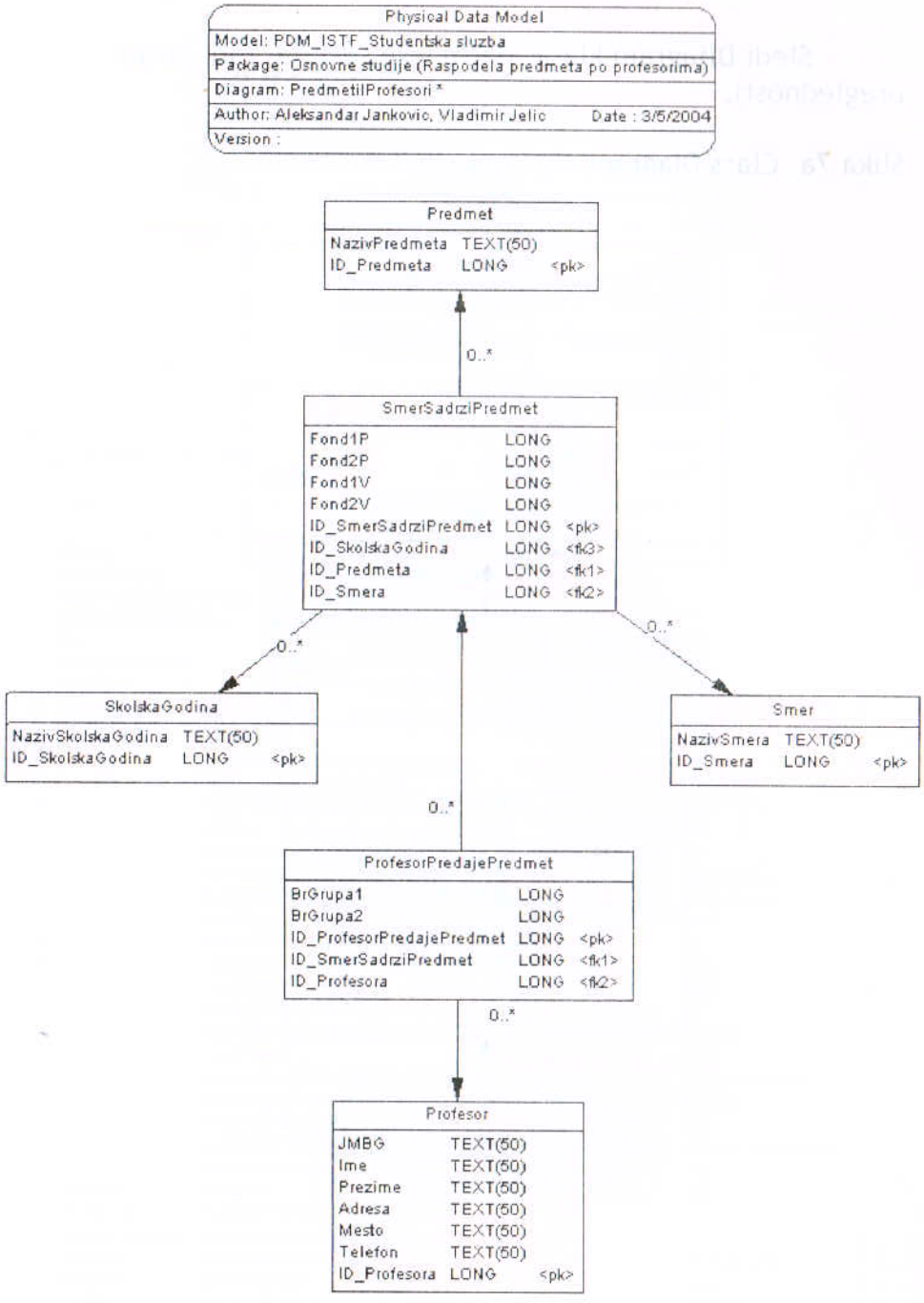
- Studentska služba
  Studentska služba prima izveštaj o raspodeli predmeta po profesorima za tekuću školsku godinu

Sledi Slika 5. CDM:

5. CDM (CONCEPTUAL DATA MODEL)



6. PDM (PHYSICAL DATA MODEL)

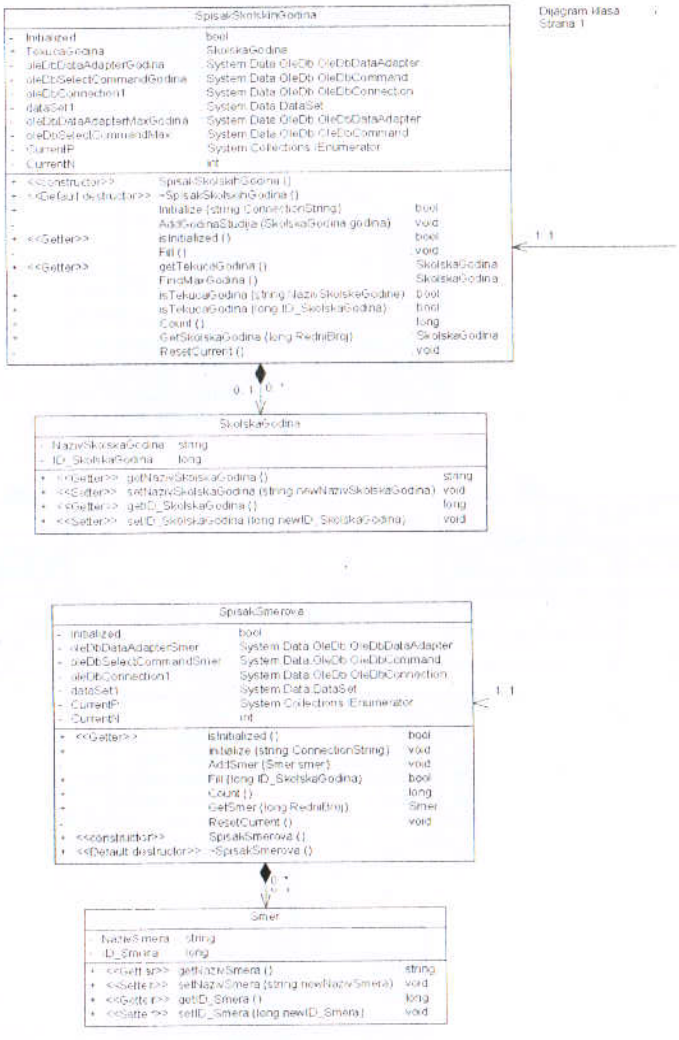


Slika 6. PDM

7. DIJAGRAM KLASA (CLASS DIAGRAM)

Sledi Dijagram klasa, dat u više sekcija, radi bolje preglednosti.

Slika 7a. Class Diagram /iz 5 sekcija/



Slika 7.b.

Object-Oriented Model	
Model: OOM_ISTF_Studentska sluzba	
Package: SSluzba	
Diagram: Prijem izvestaja o raspodeli predmeta po profesorima za tekucu skolsku godinu	
Author: Aleksandar Jankovic, Vladimir Jelic	Date: 3/4/2004
Version:	

Dijagram klasa
Strana 2

1..1

1..1

RasporedPredmetaPoProfesorima

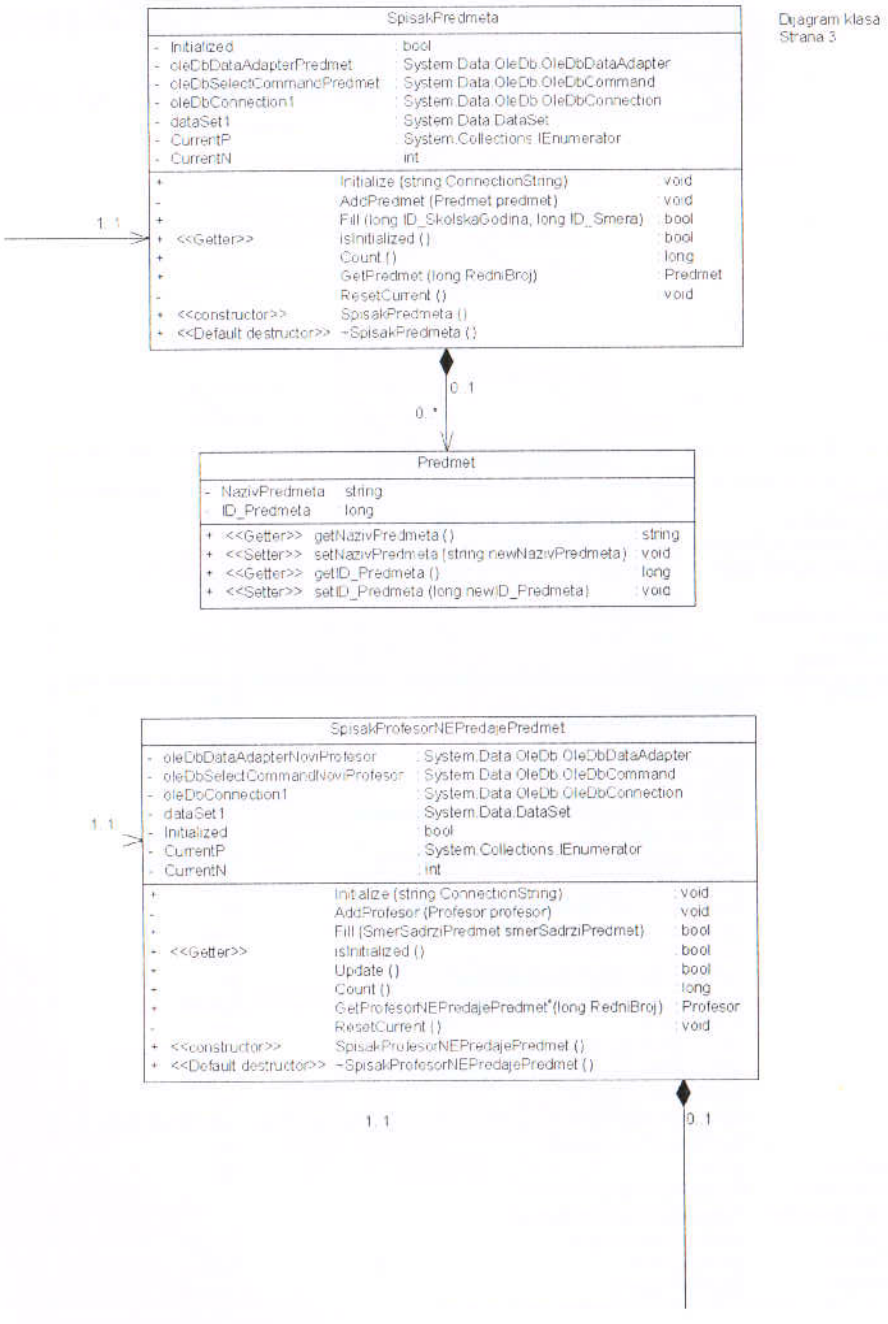
-	Initialized	: bool
-	oleDbConnection1	: System.Data.OleDb.OleDbConnection
-	oleDbCommandObrisiProfesora	: System.Data.OleDb.OleDbCommand
-	oleDbCommandNoviProfesor	: System.Data.OleDb.OleDbCommand
-	State	: byte
-	CurrentSkolskaGodina	: SkolskaGodina
-	CurrentSmer	: Smer
-	CurrentPredmet	: Predmet

+	<<Getter>>	isInitialized ()	: bool
+		Fill3BR (long RedniBrojObjektaPredmet)	: int
+		Initialize (string ConnectionString)	: bool
+		DeleteProfesor (long ID_Profesora)	: int
-		Update ()	: int
+		InsertProfesor (long ID_Profesora, int BrGrupa1, int BrGrupa2)	: int
+		GetTekucaSkolskaGodina ()	: SkolskaGodina
+		GetSkolskaGodina (long RedniBrojObjektaSkolskaGodina)	: SkolskaGodina
+		CountSkolskaGodina ()	: long
+		SmeroviFill1BR (long RedniBrojObjektaSkolskaGodina)	: int
+		GetSmer (long RedniBrojObjektaSmer)	: Smer
+		CountSmer ()	: long
+		PredmetiFill2BR (long RedniBrojObjektaSmer)	: int
+		GetPredmet (long RedniBrojObjektaPredmet)	: Predmet
+		CountPredmet ()	: long
+		getCurrentSmerSadrziPredmet ()	: SmerSadrziPredmet
+		GetProfesorNEPredajePredmet (long RedniBrojObjektaProfesor)	: Profesor
+		CountProfesorNEPredajePredmet ()	: long
+		GetProfesorPredajePredmet (long RedniBrojObjektaProfesorPredajePredmet)	: ProfesorPredajePredmet
+		CountProfesorPredajePredmet ()	: long
+	<<Getter>>	getState ()	: byte
+	<<constructor>>	RasporedPredmetaPoProfesorima ()	
+	<<Default destructor>>	~RasporedPredmetaPoProfesorima ()	
+	<<Getter>>	getCurrentSkolskaGodina ()	: SkolskaGodina
+	<<Getter>>	getCurrentSmer ()	: Smer
+	<<Getter>>	getCurrentPredmet ()	: Predmet

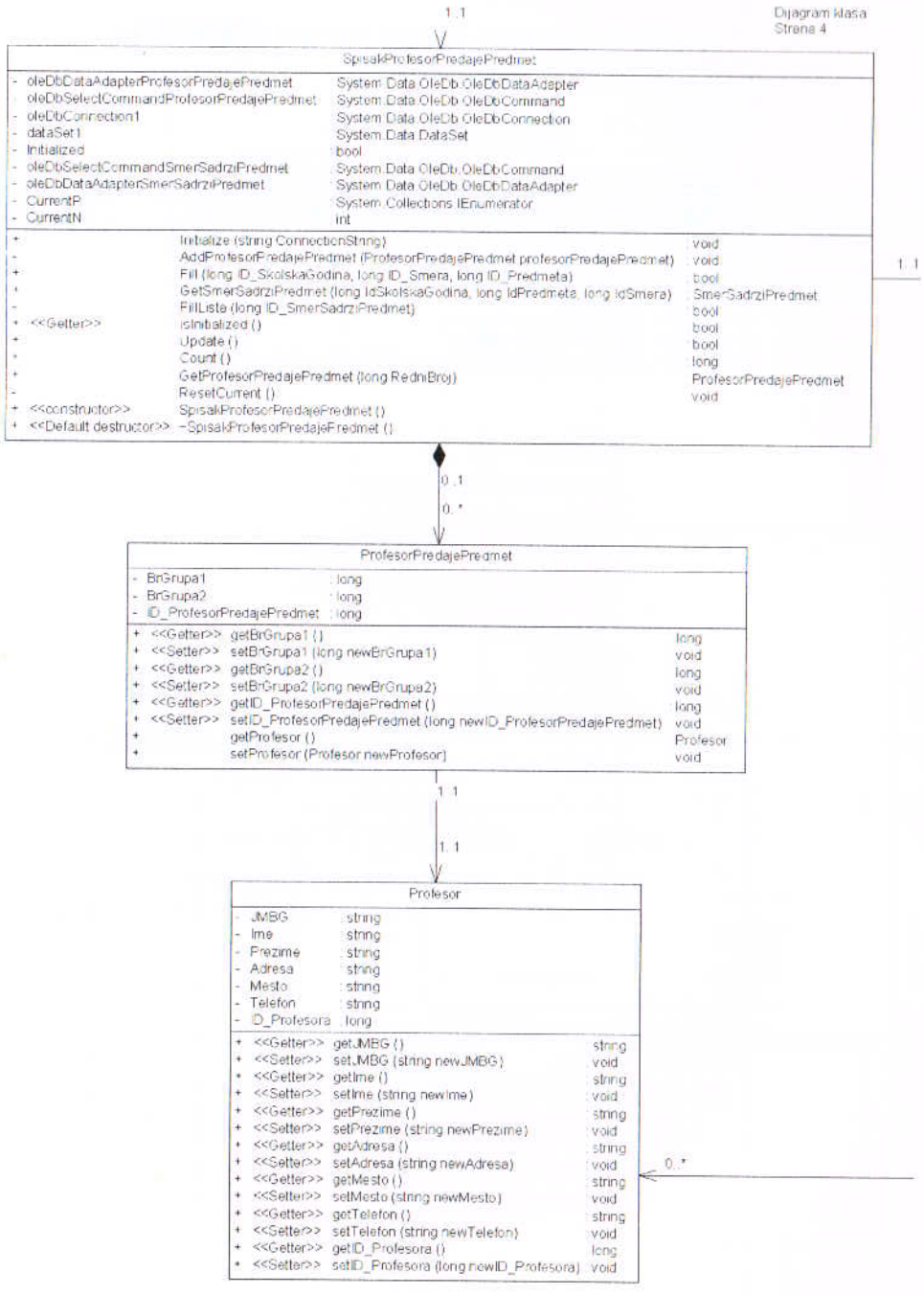
1..1

1..1

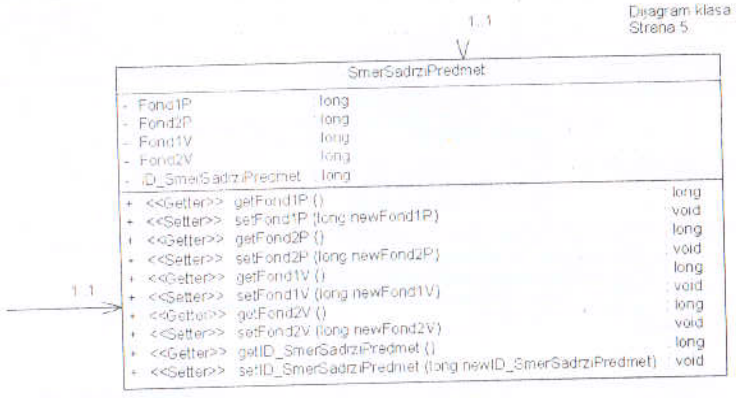
Slika 7.c.


SpisakPredmeta
- Initialized : bool
- oleDbDataAdapterPredmet : System.Data.OleDb.OleDbDataAdapter
- oleDbSelectCommandPredmet : System.Data.OleDb.OleDbCommand
- oleDbConnection1 : System.Data.OleDb.OleDbConnection
- dataSet1 : System.Data.DataSet
- CurrentP : System.Collections.IEnumerator
- CurrentN : int
+ Initialize (string ConnectionString) : void
- AddPredmet (Predmet predmet) : void
+ Fill (long ID_SkolskaGodina, long ID_Smera) : bool
+ <<Getter>> isInitialized () : bool
+ Count () : long
+ GetPredmet (long RedniBroj) : Predmet
- ResetCurrent () : void
+ <<constructor>> SpisakPredmeta ()
+ <<Default destructor>> ~SpisakPredmeta ()
1..1
0..1
0..*
Predmet
- NazivPredmeta : string
- ID_Predmeta : long
+ <<Getter>> getNazivPredmeta () : string
+ <<Setter>> setNazivPredmeta (string newNazivPredmeta) : void
+ <<Getter>> getID_Predmeta () : long
+ <<Setter>> setID_Predmeta (long newID_Predmeta) : void
SpisakProfesorNEPredajePredmet
- oleDbDataAdapterNoviProfesor : System.Data.OleDb.OleDbDataAdapter
- oleDbSelectCommandNoviProfesor : System.Data.OleDb.OleDbCommand
- oleDbConnection1 : System.Data.OleDb.OleDbConnection
- dataSet1 : System.Data.DataSet
- Initialized : bool
- CurrentP : System.Collections.IEnumerator
- CurrentN : int
+ Initialize (string ConnectionString) : void
- AddProfesor (Profesor profesor) : void
+ Fill (SmerSadrziPredmet smerSadrziPredmet) : bool
- <<Getter>> isInitialized () : bool
+ Update () : bool
- Count () : long
+ GetProfesorNEPredajePredmet*(long RedniBroj) : Profesor
- ResetCurrent () : void
+ <<constructor>> SpisakProfesorNEPredajePredmet ()
+ <<Default destructor>> ~SpisakProfesorNEPredajePredmet ()
1..1
0..1
Dijagram klasa
Strana 3

Slika 7.d.

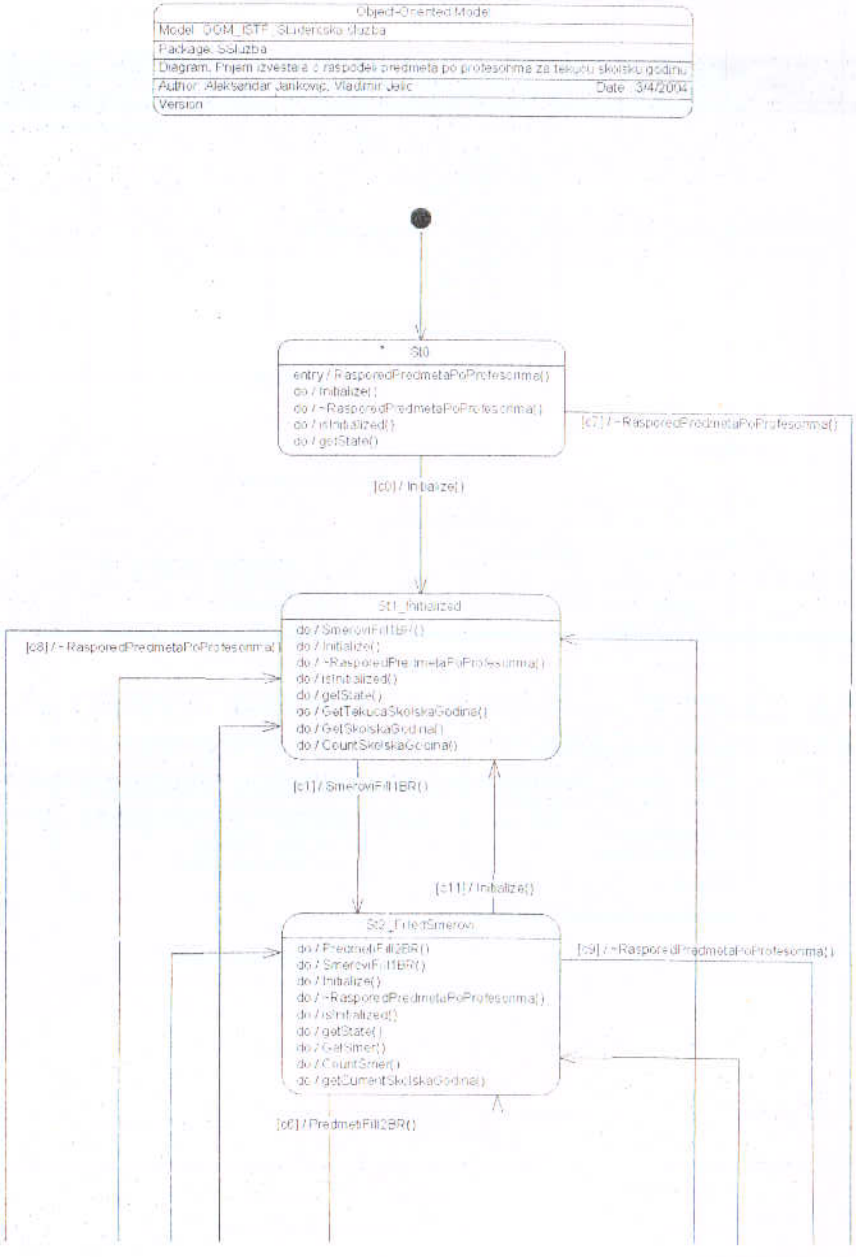

1..1
Dijagram klasa
Strana 4
SpisakProfesorPredajePredmet
- oleDbDataAdapterProfesorPredajePredmet : System.Data.OleDb.OleDbDataAdapter
- oleDbSelectCommandProfesorPredajePredmet : System.Data.OleDb.OleDbCommand
- oleDbConnection1 : System.Data.OleDb.OleDbConnection
- dataSet1 : System.Data.DataSet
- Initialized : bool
- oleDbSelectCommandSmerSadrziPredmet : System.Data.OleDb.OleDbCommand
- oleDbDataAdapterSmerSadrziPredmet : System.Data.OleDb.OleDbDataAdapter
- CurrentP : System.Collections.IEnumerator
- CurrentN : int
+ Initialize (string ConnectionString) : void
- AddProfesorPredajePredmet (ProfesorPredajePredmet profesorPredajePredmet) : void
+ Fill (long ID_SkolskaGodina, long ID_Smera, long ID_Predmeta) : bool
+ GetSmerSadrziPredmet (long IdSkolskaGodina, long IdPredmeta, long IdSmera) : SmerSadrziPredmet
- FillListe (long ID_SmerSadrziPredmet) : bool
+ <<Getter>> isInitialized () : bool
+ Update () : bool
+ Count () : long
+ GetProfesorPredajePredmet (long RedniBroj) : ProfesorPredajePredmet
- ResetCurrent () : void
+ <<constructor>> SpisakProfesorPredajePredmet ()
+ <<Default destructor>> ~SpisakProfesorPredajePredmet ()
1..1
0..1
0..*
ProfesorPredajePredmet
- BrGrupa1 : long
- BrGrupa2 : long
- ID_ProfesorPredajePredmet : long
+ <<Getter>> getBrGrupa1 () : long
+ <<Setter>> setBrGrupa1 (long newBrGrupa1) : void
+ <<Getter>> getBrGrupa2 () : long
+ <<Setter>> setBrGrupa2 (long newBrGrupa2) : void
+ <<Getter>> getID_ProfesorPredajePredmet () : long
+ <<Setter>> setID_ProfesorPredajePredmet (long newID_ProfesorPredajePredmet) : void
+ getProfesor () : Profesor
+ setProfesor (Profesor newProfesor) : void
1..1
1..1
Profesor
- JMBG : string
- Ime : string
- Prezime : string
- Adresa : string
- Mesto : string
- Telefon : string
- ID_Profesora : long
+ <<Getter>> getJMBG () : string
+ <<Setter>> setJMBG (string newJMBG) : void
+ <<Getter>> getIme () : string
+ <<Setter>> setIme (string newIme) : void
+ <<Getter>> getPrezime () : string
+ <<Setter>> setPrezime (string newPrezime) : void
+ <<Getter>> getAdresa () : string
+ <<Setter>> setAdresa (string newAdresa) : void
+ <<Getter>> getMesto () : string
+ <<Setter>> setMesto (string newMesto) : void
+ <<Getter>> getTelefon () : string
+ <<Setter>> setTelefon (string newTelefon) : void
+ <<Getter>> getID_Profesora () : long
+ <<Setter>> setID_Profesora (long newID_Profesora) : void
0..*

Slika 7.e.

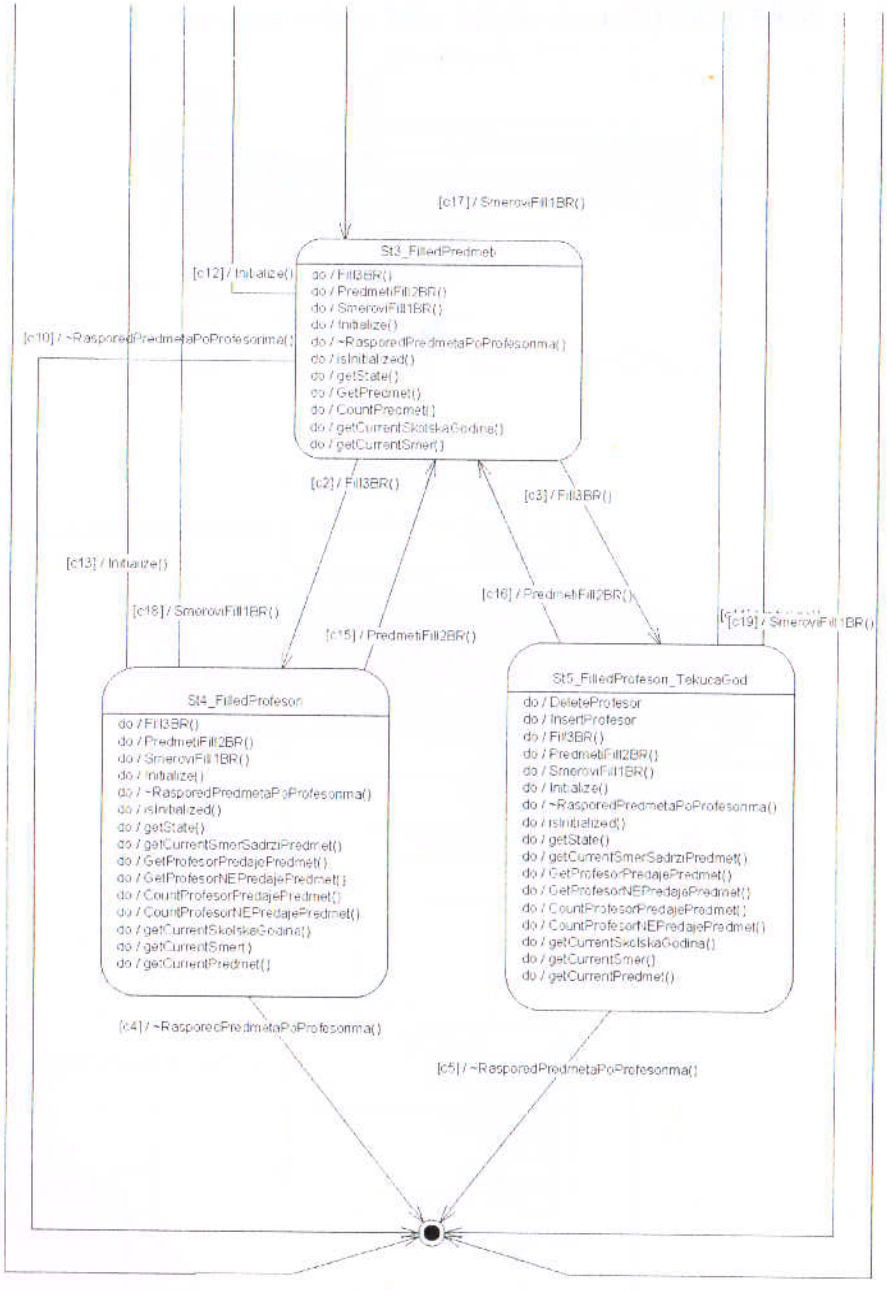

Dijagram klasa
Strana 5
1..1
SmerSadrziPredmet
- Fond1P long
- Fond2P long
- Fond1V long
- Fond2V long
- ID_SmerSadrziPredmet long
+ <<Getter>> getFond1P () long
+ <<Setter>> setFond1P (long newFond1P) void
+ <<Getter>> getFond2P () long
+ <<Setter>> setFond2P (long newFond2P) void
+ <<Getter>> getFond1V () long
+ <<Setter>> setFond1V (long newFond1V) void
+ <<Getter>> getFond2V () long
+ <<Setter>> setFond2V (long newFond2V) void
+ <<Getter>> getID_SmerSadrziPredmet () long
+ <<Setter>> setID_SmerSadrziPredmet (long newID_SmerSadrziPredmet) void
1..1

8. DIJAGRAM STANJA

Slika 8.a.Dijagram stanja /dat u dve sekcije/

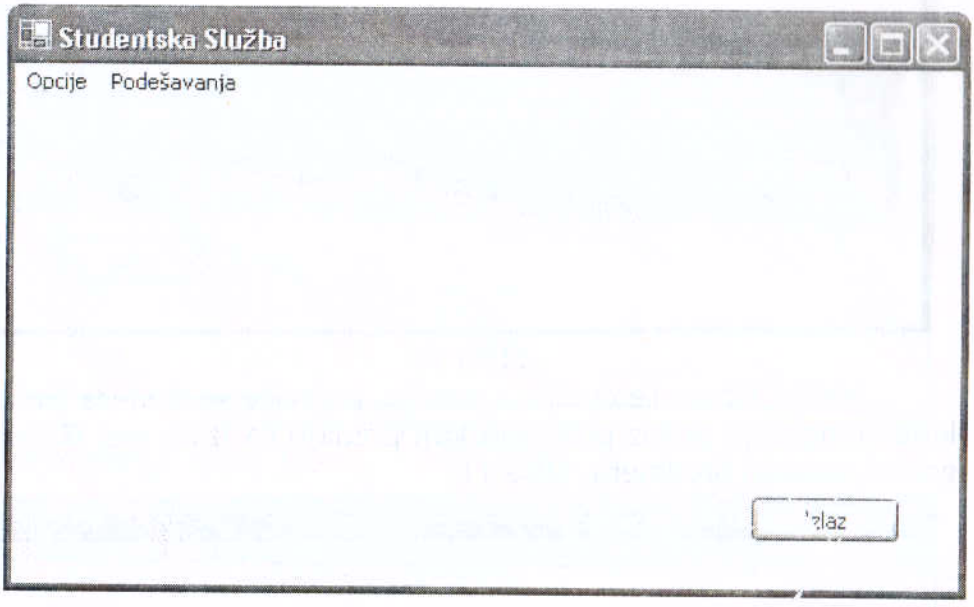


Slika 8.b.


[c17] / SmeroviFill1BR()
St3_FilledPredmet
do / Fill3BR()
do / PredmetiFill2BR()
do / SmeroviFill1BR()
do / Initialize()
do / ~RasporedPredmetaPoProfesorima()
do / isInitialized()
do / getState()
do / GetPredmet()
do / CountPredmet()
do / getCurrentSkolskaGodina()
do / getCurrentSmer()
[c12] / Initialize()
[c10] / ~RasporedPredmetaPoProfesorima()
[c2] / Fill3BR()
[c3] / Fill3BR()
[c13] / Initialize()
[c18] / SmeroviFill1BR()
[c16] / PredmetiFill2BR()
[c15] / PredmetiFill2BR()
[c19] / SmeroviFill1BR()
St4_FilledProfesor
do / Fill3BR()
do / PredmetiFill2BR()
do / SmeroviFill1BR()
do / Initialize()
do / ~RasporedPredmetaPoProfesorima()
do / isInitialized()
do / getState()
do / getCurrentSmerSadrziPredmet()
do / GetProfesorPredajePredmet()
do / GetProfesorNEPredajePredmet()
do / CountProfesorPredajePredmet()
do / CountProfesorNEPredajePredmet()
do / getCurrentSkolskaGodina()
do / getCurrentSmer()
do / getCurrentPredmet()
St5_FilledProfesor_TekucaGod
do / DeleteProfesor
do / InsertProfesor
do / Fill3BR()
do / PredmetiFill2BR()
do / SmeroviFill1BR()
do / Initialize()
do / ~RasporedPredmetaPoProfesorima()
do / isInitialized()
do / getState()
do / getCurrentSmerSadrziPredmet()
do / GetProfesorPredajePredmet()
do / GetProfesorNEPredajePredmet()
do / CountProfesorPredajePredmet()
do / CountProfesorNEPredajePredmet()
do / getCurrentSkolskaGodina()
do / getCurrentSmer()
do / getCurrentPredmet()
[c4] / ~RasporedPredmetaPoProfesorima()
[c5] / ~RasporedPredmetaPoProfesorima()

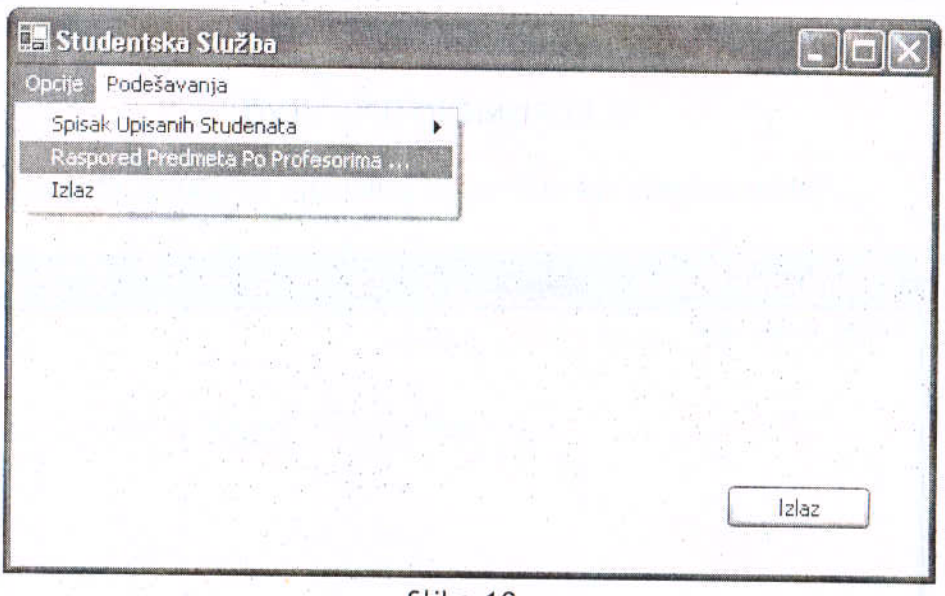
9. KORISNIČKO UPUTSTVO

Nakon pokretanja aplikacije prikazuje se glavna forma, Slika 9:



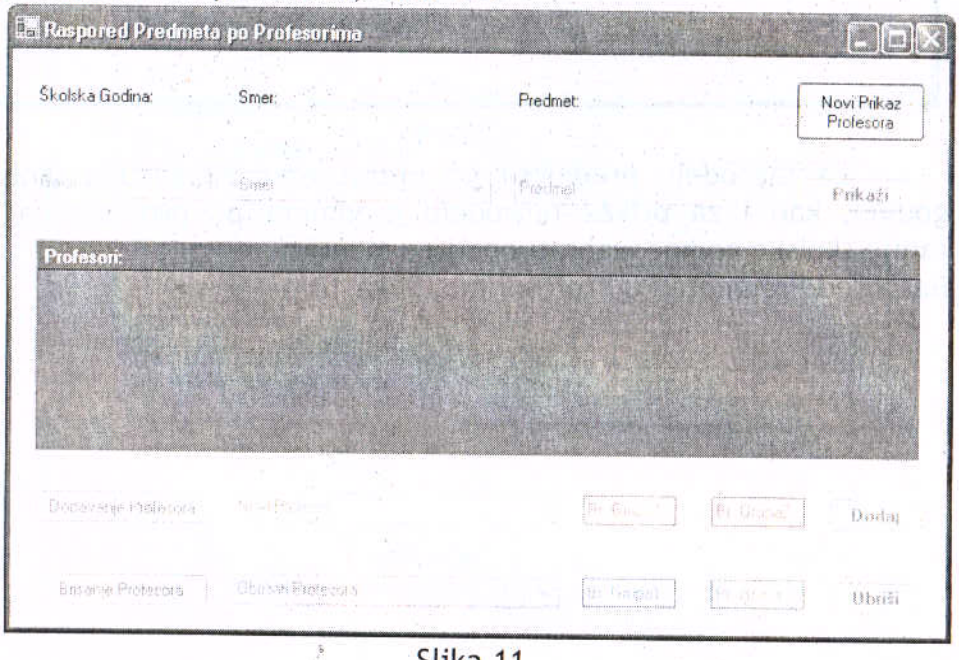
Slika 9

Za raspodelu predmeta po profesorima za tekuću školsku godinu, kao i za prikaz raspodele predmeta po profesorima za ranije školske godine izabrati opciju u meniju:
Raspored Predmeta Po Profesorima, Slika 10:



Slika 10.

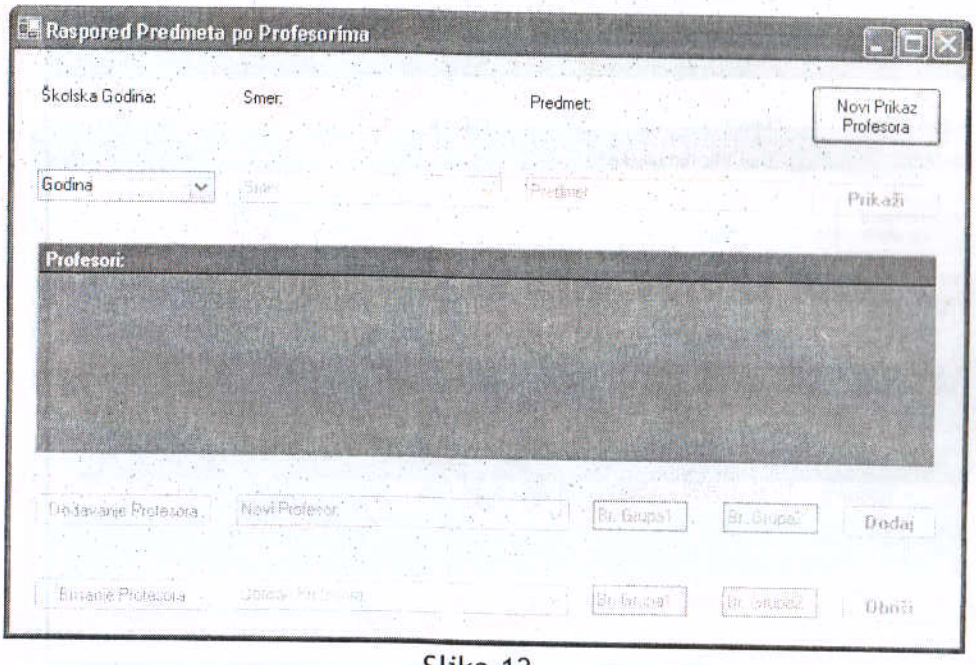
Nakon izbora te opcije u meniju, pojaviće se sledeća forma, koja omogućuje prikaz profesora koji predaju na izabranoj školskoj godini, smeru i predmetu, Slika 11:



Slika 11.

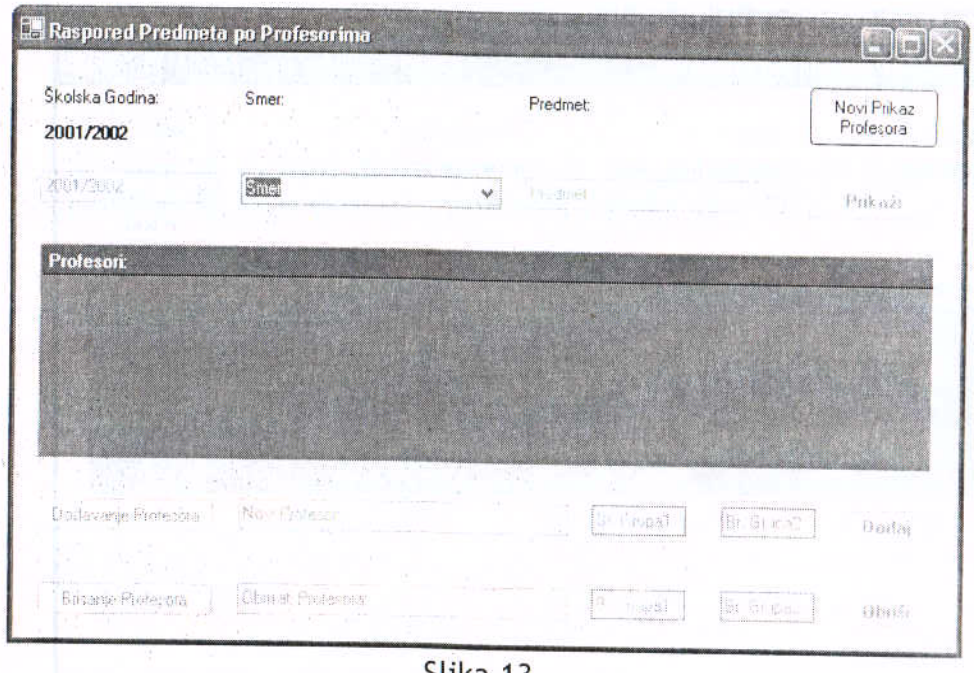
Pritiskom na taster Novi Prikaz Profesora, Omogućuje se novi izbor za prikaz, Slika 12. Zatim treba redom izabrati:

1) Željenu godinu:



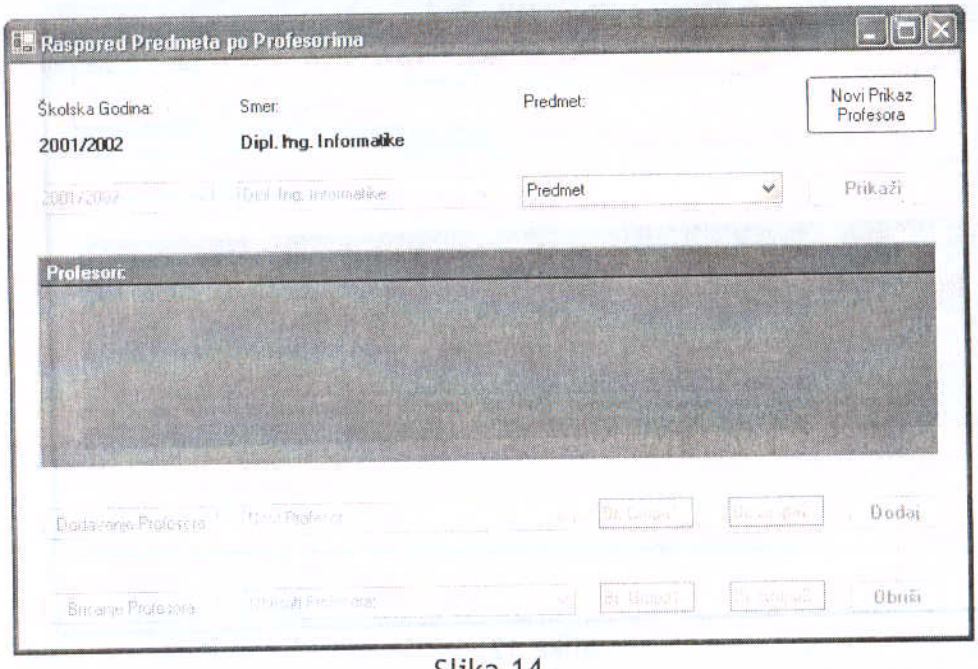
Slika 12.

2) Željeni smer, Slika 13:



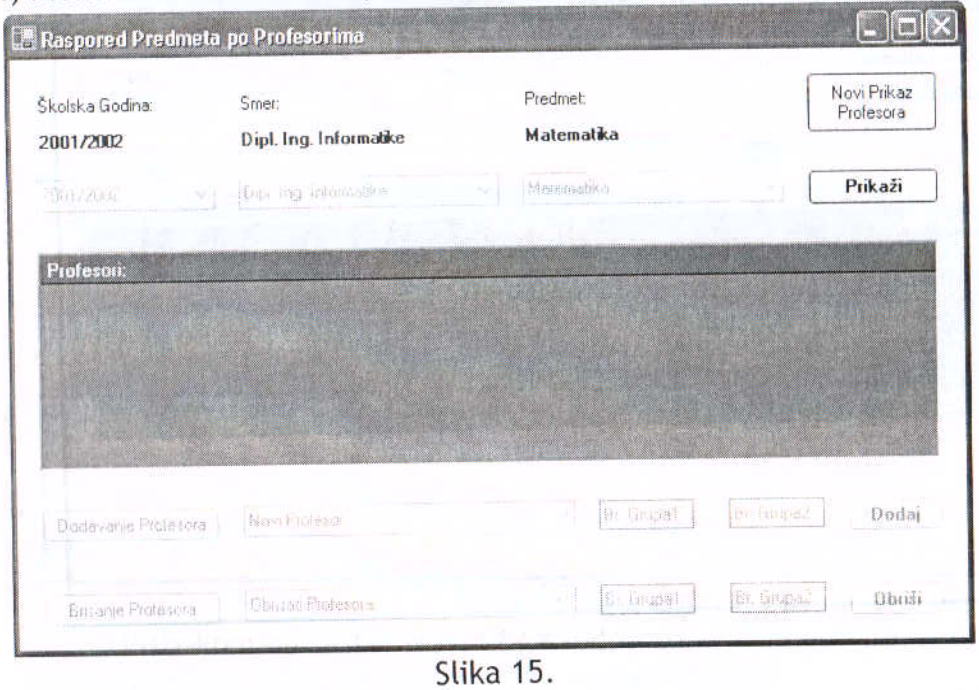
Slika 13.

3) Željeni predmet, Slika 14:



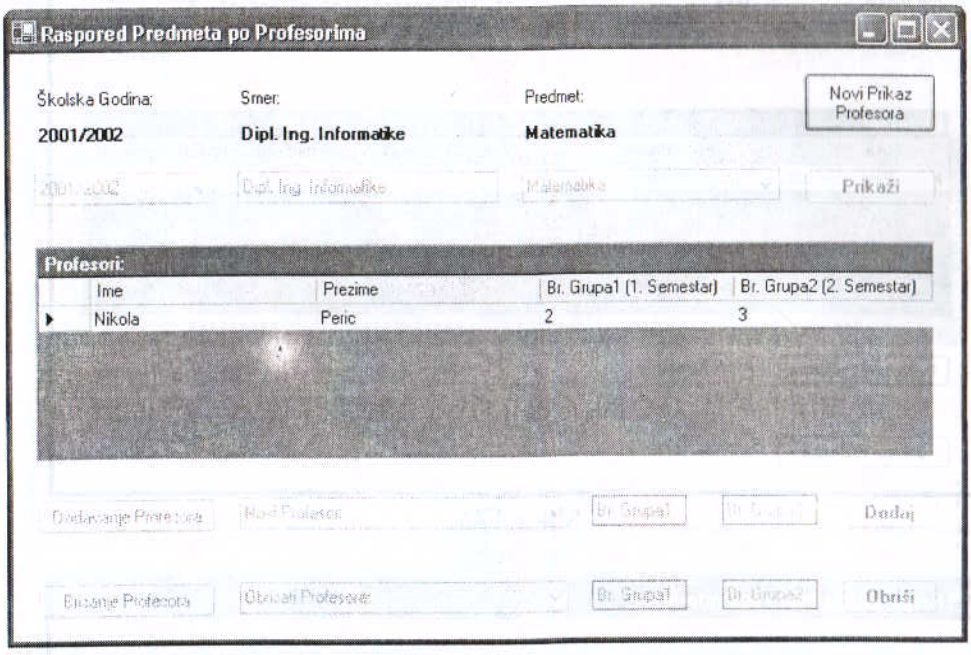
Slika 14.

4) Pritisnuti taster Prikaži, Slika 15:



Slika 15.

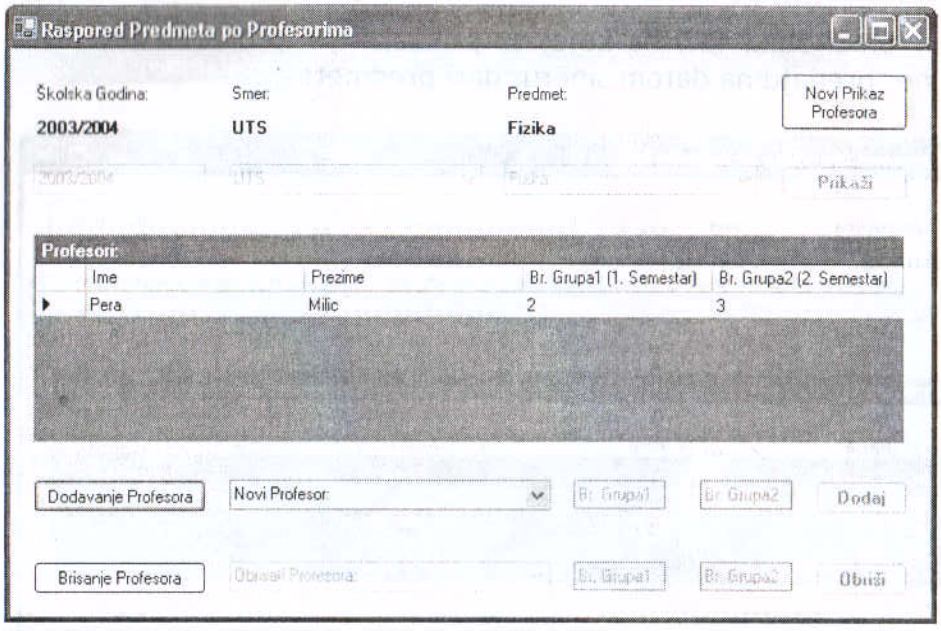
Sledi Slika 16., na kojoj su prikazani svi profesori koji date godine, predaju na datom smeru, dati predmet:



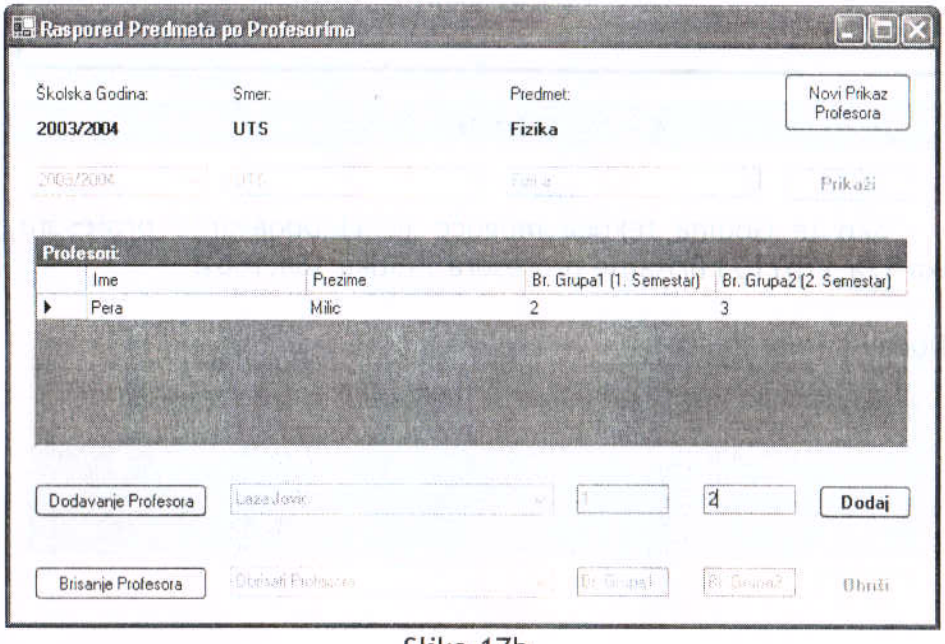
Slika 16.

Ako je Godina tekuća moguće je 1) Dodavanje profesora, (Slika 17a,17b) i 2) Brisanje profesora , (Slika 18a,18b):

1) Dodavanje profesora:

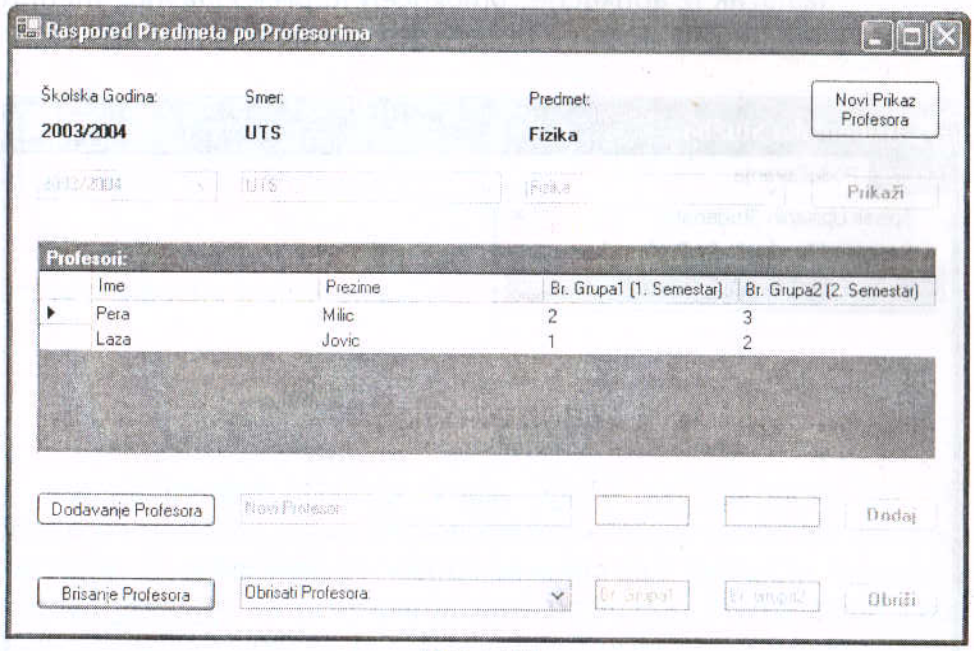


Slika 17a.

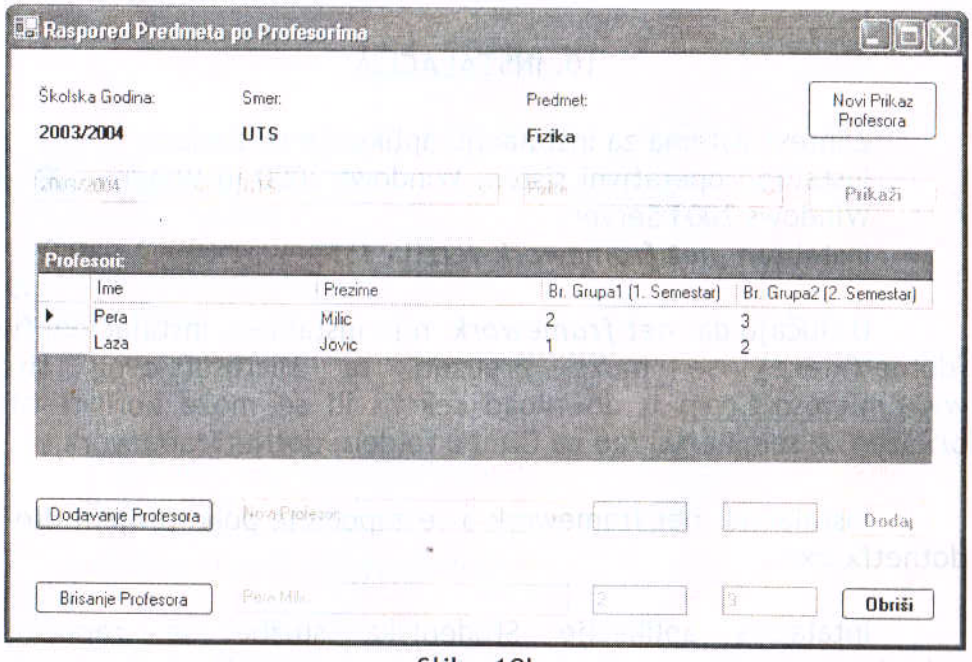


Slika 17b.

2) Brisanje profesora:

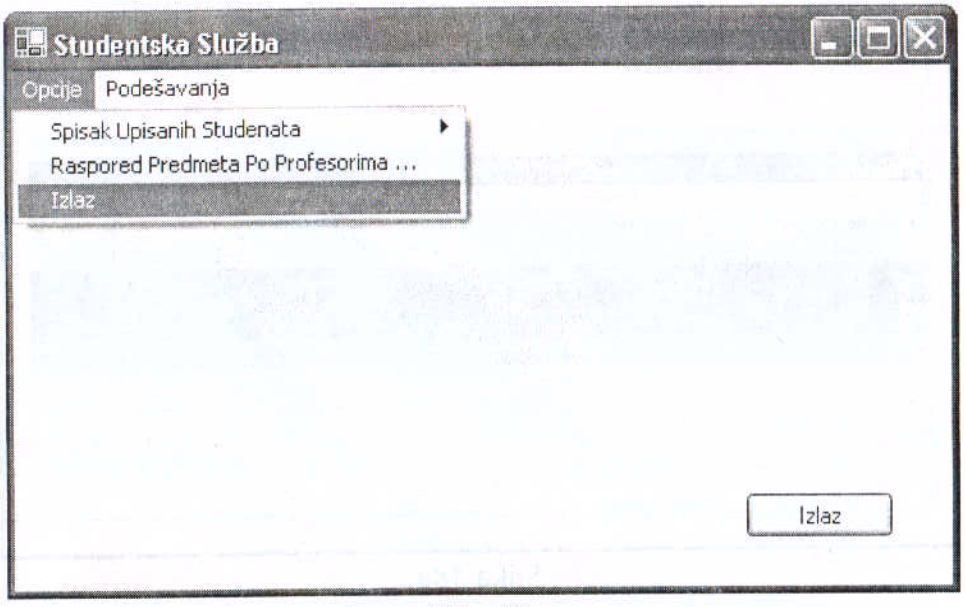


Slika 18a.



Slika 18b.

Izalazak iz aplikacije, omogućen je preko glavnog menija, opcijom izlaz, ili pritiskom na taster izlaz glavne forme, Slika 19:



Slika 19.

10. INSTALACIJA

Zahtevi sistema za instalaciju aplikacije su sledeći:

- Instaliran operativni sistem Windows 2000 ili Windows XP ili Windows 2003 Server.
- instaliran ***.net framework*** verzija ***1.1***.

U slučaju da ***.net framework*** nije instaliran, instalacioni file (dotnetfx.exe) se može preuzeti sa Microsoft-ovog site: www.microsoft.com u download sekciji ili se može korisiti isti, priložen uz seminarski rad na CD-u u folderu dotNetFramework.

Instalacija .net framework-a se započinje pokretanjem file-a dotnetfx.exe.

Intalacija aplikacije Studentska služba se započinje pokretanjem file-a StudentskaSluzba.msi, ili jednostavnim ubacivanjem priloženog CD-a u CD čitač (ako je aktivna opicja ***autorun***).

Po završetku instalacije, na desktopu će se naći shorcut pod nazivom StudentskaSluzba, kojom se aplikacija pokreće.

Obzirom da je karakter aplikacije demonstrativni, pri instalaciji aplikacije, kopiraju se i obe baze koje se koriste samo radi demonstracije aplikacije.

Svi fajlovi potrebni za rad aplikacije nalaze se u folderu u kom je aplikacija instalirana.

11. LISTING KODA

Analiziranjem listinga koda, može se zaključiti da je kompletna poslovna logika implementirana nezavisno od korisničkog interfejsa. Iz priloženog se vidi da se u glavnoj aplikaciji ne nalazi ni jedan element za pristupanje bazi podataka. Prema tome cela poslovna logika izvedena je pomoću klasa prikazanim na dijagramu klasa, čija se implementacija nalazi u dll-u, pod nazivom StudSluzaba.dll.

11.1. FORMA 1

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace SSluzba
{
        /// <summary>
        /// Summary description for Form1.
        /// </summary>

        public class Form1 : SSluzba.FormXP
        {
                /// <summary>
                /// Required designer variable.
                /// </summary>
                private System.ComponentModel.Container components = null;
```

```
public static Form1 MainForm=null;
private System.Windows.Forms.MainMenu mainMenu1;
private System.Windows.Forms.MenuItem menuOpcije;
private System.Windows.Forms.MenuItem menuSpisak;
private System.Windows.Forms.MenuItem menuRaspored;
private System.Windows.Forms.MenuItem menuIzlaz;
private System.Windows.Forms.MenuItem menuPodesavanja;
private System.Windows.Forms.MenuItem menuConn;
private System.Windows.Forms.MenuItem menuXML;
//Our def
public StudSluzba.SpisakUpisanihStudenata spisak;
private System.Windows.Forms.Button btnClose;
private static System.String connString;
private static System.String xmlPath;
private System.Windows.Forms.MenuItem menuItem1;
private System.Windows.Forms.MenuItem menuItem2;
private static System.String xmlFileName;

public System.String ConnectionString
{
        get
        {
                return connString;
        }
        set
        {
                connString = value;
        }
}
//

public System.String XmlFilePath
{
        get
        {
                return xmlPath;
        }
        set
        {
                xmlPath = value;
        }
}
public System.String XmlFileName
{
        get
        {
                return xmlFileName;
        }
```

```
                set
                {
                        xmlFileName = value;
                }
        }
        //Our def end

        public Form1()
        {
                //
                // Required for Windows Form Designer support
                //
                InitializeComponent();
                this.SetXPStyle();
                //
                // TODO: Add any constructor code after
InitializeComponent call
                //
        }


        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
                if( disposing )
                {
                        if (components != null)
                        {
                                components.Dispose();
                        }
                }
                base.Dispose( disposing );
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
                this.mainMenu1 = new
System.Windows.Forms.MainMenu();
                this.menuOpcije = new
System.Windows.Forms.MenuItem();
```

```
                    this.menuSpisak = new
System.Windows.Forms.MenuItem();
                    this.menuItem1 = new
System.Windows.Forms.MenuItem();
                    this.menuItem2 = new
System.Windows.Forms.MenuItem();
                    this.menuRaspored = new
System.Windows.Forms.MenuItem();
                    this.menuIzlaz = new
System.Windows.Forms.MenuItem();
                    this.menuPodesavanja = new
System.Windows.Forms.MenuItem();
                    this.menuConn = new
System.Windows.Forms.MenuItem();
                    this.menuXML = new System.Windows.Forms.MenuItem();
                    this.btnClose = new System.Windows.Forms.Button();
                    this.SuspendLayout();
                    //
                    // mainMenu1
                    //
                    this.mainMenu1.MenuItems.AddRange(new
System.Windows.Forms.MenuItem[] {

        this.menuOpcije,


        this.menuPodesavanja});
                    //
                    // menuOpcije
                    //
                    this.menuOpcije.Index = 0;
                    this.menuOpcije.MenuItems.AddRange(new
System.Windows.Forms.MenuItem[] {

        this.menuSpisak,


        this.menuRaspored,


        this.menuIzlaz});
                    this.menuOpcije.Text = "Opcije";
                    //
                    // menuSpisak
                    //
                    this.menuSpisak.Index = 0;
```

```
                    this.menuSpisak.MenuItems.AddRange(new
System.Windows.Forms.MenuItem[] {


        this.menuItem1,


        this.menuItem2});
                    this.menuSpisak.Text = "Spisak Upisanih Studenata";
                    this.menuSpisak.Click += new
System.EventHandler(this.menuSpisak_Click);
                    //
                    // menuItem1
                    //
                    this.menuItem1.Index = 0;
                    this.menuItem1.Text = "Svi podaci u jednoj tabeli ...";
                    this.menuItem1.Click += new
System.EventHandler(this.menuItem1_Click);
                    //
                    // menuItem2
                    //
                    this.menuItem2.Index = 1;
                    this.menuItem2.Text = "Grupisano po Godini studija i
Smeru ...";
                    this.menuItem2.Click += new
System.EventHandler(this.menuItem2_Click);
                    //
                    // menuRaspored
                    //
                    this.menuRaspored.Index = 1;
                    this.menuRaspored.Text = "Raspored Predmeta Po
Profesorima ...";
                    this.menuRaspored.Click += new
System.EventHandler(this.menuRaspored_Click);
                    //
                    // menuIzlaz
                    //
                    this.menuIzlaz.Index = 2;
                    this.menuIzlaz.Text = "Izlaz";
                    this.menuIzlaz.Click += new
System.EventHandler(this.menuIzlaz_Click);
                    //
                    // menuPodesavanja
                    //
                    this.menuPodesavanja.Index = 1;
                    this.menuPodesavanja.MenuItems.AddRange(new
System.Windows.Forms.MenuItem[] {
```

```
                    this.menuConn,


                    this.menuXML});
                    this.menuPodesavanja.Text = "Podešavanja";
                    //
                    // menuConn
                    //
                    this.menuConn.Index = 0;
                    this.menuConn.Text = "Connection String ...";
                    this.menuConn.Click += new
System.EventHandler(this.menuConn_Click);
                    //
                    // menuXML
                    //
                    this.menuXML.Index = 1;
                    this.menuXML.Text = "XML File Path ...";
                    this.menuXML.Click += new
System.EventHandler(this.menuXML_Click);
                    //
                    // btnClose
                    //
                    this.btnClose.Location = new System.Drawing.Point(368,
192);
                    this.btnClose.Name = "btnClose";
                    this.btnClose.TabIndex = 0;
                    this.btnClose.Text = "Izlaz";
                    this.btnClose.Click += new
System.EventHandler(this.btnClose_Click);
                    //
                    // Form1
                    //
                    this.AutoScaleBaseSize = new System.Drawing.Size(5,
13);
                    this.ClientSize = new System.Drawing.Size(472, 238);
                    this.Controls.Add(this.btnClose);
                    this.Menu = this.mainMenu1;
                    this.Name = "Form1";
                    this.StartPosition =
System.Windows.Forms.FormStartPosition.CenterScreen;
                    this.Text = "Studentska Služba";
                    this.Load += new System.EventHandler(this.Form1_Load);
                    this.ResumeLayout(false);

                }
                #endregion
```

```
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
                MainForm = new Form1();
                Application.Run(new Form1());
        }

        private void menuIzlaz_Click(object sender, System.EventArgs e)
        {
                Application.Exit();
        }

        private void menuSpisak_Click(object sender, System.EventArgs
e)
        {

        }

        private void menuConn_Click(object sender, System.EventArgs e)
        {
                FormConnection fConn =new FormConnection();
                fConn.Show();
        }

        private void menuXML_Click(object sender, System.EventArgs e)
        {
                FormXML fXml =new FormXML();
                fXml.Show();
        }

        private void menuRaspored_Click(object sender,
System.EventArgs e)
        {
                FormRaspored fRaspored =new FormRaspored();
                fRaspored.Show();
        }


        private void Form1_Load(object sender, System.EventArgs e)
        {
                string s, appPath, appFull, appName = "SSluzba.exe";
                appFull = Application.ExecutablePath;
                appPath = appFull.Substring(0, appFull.Length -
appName.Length);
```

```
                    s = appPath + "DosijeStudenata.mdb";

                    this.spisak = new StudSluzba.SpisakUpisanihStudenata();
                    this.ConnectionString =
@"Provider=Microsoft.Jet.OLEDB.4.0;Data Source="+s+@";Persist Security
Info=False";
                    this.XmlFilePath = @"C:\SSluzbaXML";
                    this.XmlFileName = @"SpisakUpisanihStudenata";

            }

            private void btnClose_Click(object sender, System.EventArgs e)
            {
                    Application.Exit();
            }

            private void menuItem2_Click(object sender, System.EventArgs
e)
            {
                    if (this.spisak.Initialize(this.ConnectionString))

        this.spisak.ShowXMLinIETab(this.XmlFilePath,this.XmlFileName);
            }

            private void menuItem1_Click(object sender, System.EventArgs
e)
            {
                    this.spisak.Initialize(this.ConnectionString);
                    if (this.spisak.isInitialized())

        this.spisak.ShowXMLinIE(this.XmlFilePath,this.XmlFileName);
            }

        }
}
```

11.2. FORMA XP

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

```
namespace SSluzba
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>
	public class FormXP : System.Windows.Forms.Form
	{
		public FormXP()
		{
			//
			// TODO: Add constructor logic here
			//
		}

		private void RecursivelyFormatForWinXP(Control control)
		{
			for(int x = 0; x < control.Controls.Count; x++)
			{
				// If the control derives from ButtonBase,
				//	set its FlatStyle property to
FlatStyle.System.
				if(control.Controls[x].GetType().BaseType ==
typeof(ButtonBase))
				{
	((ButtonBase)control.Controls[x]).FlatStyle = FlatStyle.System;
				}

				// If the control holds other controls, iterate
through them also.
				if(control.Controls.Count > 0)
				{
	RecursivelyFormatForWinXP(control.Controls[x]);
				}
			}
		}

		protected void SetXPStyle()
		{
			// Makes sure Windows XP is running and
			//  a .manifest file exists for the EXE.
			if(Environment.OSVersion.Version.Major > 4
				& Environment.OSVersion.Version.Minor > 0
```

```
                    &
System.IO.File.Exists(Application.ExecutablePath + ".manifest"))
                {
                    // Iterate through the controls.
                    for(int x = 0; x < this.Controls.Count; x++)
                    {
                        // If the control derives from
ButtonBase,
                        // set its FlatStyle property to
FlatStyle.System.

        if(this.Controls[x].GetType().BaseType == typeof(ButtonBase))
                        {

        ((ButtonBase)this.Controls[x]).FlatStyle = FlatStyle.System;
                        }

        RecursivelyFormatForWinXP(this.Controls[x]);
                    }
                }
            }
        }
}
```

11.3. FORMA RASPORED

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace SSluzba
{
        /// <summary>
        /// Summary description for FormRaspored.
        /// </summary>
        ///
        public class FormRaspored : SSluzba.FormXP
        {
                private System.Windows.Forms.ComboBox cmbGodina;
                private System.Windows.Forms.ComboBox cmbPredmet;
                private System.Windows.Forms.ComboBox cmbSmer;
                private System.Windows.Forms.Button btnPrikazi;
```

```
private System.Windows.Forms.ComboBox cmbNovi;
private System.Windows.Forms.ComboBox cmbObrisi;
private System.Windows.Forms.Button btnNovi;
private System.Windows.Forms.Button btnObrisi;
private System.Windows.Forms.TextBox txtNoviBr1;
private System.Windows.Forms.TextBox txtObrisiBr2;
private System.Windows.Forms.TextBox txtObrisiBr1;
private System.Windows.Forms.TextBox txtNoviBr2;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Label label3;
private System.Windows.Forms.Button btnOdabir;
private System.Windows.Forms.Label lbGodina;
private System.Windows.Forms.Label lbSmer;
private System.Windows.Forms.Label lbPredmet;
private StudSluzba.RasporedPredmetaPoProfesorima Raspored;
private static System.String connString;
private System.Windows.Forms.Button btnDodaj;
private System.Windows.Forms.Button btnBrisanje;
private System.Windows.Forms.ListView listView1;
private System.Windows.Forms.ColumnHeader columnHeader1;
private System.Windows.Forms.ColumnHeader columnHeader2;
private System.Windows.Forms.ColumnHeader columnHeader3;
private System.Windows.Forms.ColumnHeader columnHeader4;
private System.ComponentModel.Container components = null;
/// <summary>

/// </summary>

private void DisableAll()
{
        this.DisableObrisi();
        this.DisableNovi();
        this.DisableBrisanje();
        this.DisableDodavanje();
        this.DisableGodina();
        this.DisableSmer();
        this.DisablePredmet();
        this.DisablePrikazi();
}

private void DisableObrisi()
{
        this.cmbObrisi.Enabled = false;
        this.txtObrisiBr1.Enabled = false;
        this.txtObrisiBr2.Enabled = false;
        this.btnObrisi.Enabled = false;
```

```
            this.txtObrisiBr1.Text = "Br. Grupa1";
            this.txtObrisiBr2.Text = "Br. Grupa2";
            this.cmbObrisi.Text = "Obrisati Profesora:";
    }

    private void DisableCmbObrisi()
    {
            this.cmbObrisi.Enabled = false;
    }

    private void EnableCmbObrisi()
    {
            this.cmbObrisi.Enabled = true;
    }

    private void DisableCmbNovi()
    {
            this.cmbNovi.Enabled = false;
    }

    private void EnableCmbNovi()
    {
            this.cmbNovi.Enabled = true;
    }

    private void EnableObrisi()
    {
            this.cmbObrisi.Enabled = true;
            this.txtObrisiBr1.Enabled = true;
            this.txtObrisiBr2.Enabled = true;
            this.btnObrisi.Enabled = true;
    }

    private void DisableNovi()
    {
            this.cmbNovi.Enabled = false;
            this.txtNoviBr1.Enabled = false;
            this.txtNoviBr2.Enabled = false;
            this.btnDodaj.Enabled = false;

            this.txtNoviBr1.Text = "Br. Grupa1";
            this.txtNoviBr2.Text = "Br. Grupa2";
            this.cmbNovi.Text = "Novi Profesor:";
    }

    private void EnableNovi()
    {
            this.cmbNovi.Enabled = true;
```

```
            this.txtNoviBr1.Enabled = true;
            this.txtNoviBr2.Enabled = true;
            this.btnNovi.Enabled = true;
}

private void DisableGodina()
{
            this.cmbGodina.Enabled = false;
}

private void EnableGodina()
{
            this.cmbGodina.Enabled = true;
}

private void DisableSmer()
{
            this.cmbSmer.Enabled = false;
}

private void EnableSmer()
{
            this.cmbSmer.Enabled = true;
}

private void DisablePredmet()
{
            this.cmbPredmet.Enabled = false;
}

private void EnablePredmet()
{
            this.cmbPredmet.Enabled = true;
}

private void DisablePrikazi()
{
            this.btnPrikazi.Enabled = false;
}

private void EnablePrikazi()
{
            this.btnPrikazi.Enabled = true;
}

private void DisableDodavanje()
{
            this.btnNovi.Enabled = false;
```

```
        }

        private void EnableDodavanje()
        {
                this.btnNovi.Enabled = true;
        }

        private void DisableBrisanje()
        {
                this.btnBrisanje.Enabled = false;
        }

        private void EnableBrisanje()
        {
                this.btnBrisanje.Enabled = true;
        }

        private void ClearLabels()
        {
                this.lbGodina.Text = "";
                this.lbSmer.Text = "";
                this.lbPredmet.Text = "";
        }

        private bool IsNumber(string s)
        {
                if (s == "")
                        return false;
                foreach(char c in s.ToCharArray())
                        if (!(System.Char.IsNumber(c)))
                                return false;
                return true;
        }
        private void FillListView()
        {
                this.Raspored.Update();
                this.listView1.Items.Clear();
                System.Windows.Forms.ListViewItem a;
                for (long i=1, n =
this.Raspored.CountProfesorPredajePredmet(); i<=n; i++)
                {
                        string s1 =
this.Raspored.GetProfesorPredajePredmet(i).getProfesor().getIme();
                        string s2 =
this.Raspored.GetProfesorPredajePredmet(i).getProfesor().getPrezime();
                        string s3 =
this.Raspored.GetProfesorPredajePredmet(i).getBrGrupa1().ToString();
```

```
                                string s4 =
this.Raspored.GetProfesorPredajePredmet(i).getBrGrupa2().ToString();
                                a = this.listView1.Items.Add(s1);
                                a.SubItems.Add(s2);
                                a.SubItems.Add(s3);
                                a.SubItems.Add(s4);
                        }
                }

                public FormRaspored()
                {
                        //
                        // Required for Windows Form Designer support
                        //
                        InitializeComponent();

                        //
                        // TODO: Add any constructor code after
InitializeComponent call
                        //
                }

                /// <summary>
                /// Clean up any resources being used.
                /// </summary>
                protected override void Dispose( bool disposing )
                {
                        if( disposing )
                        {
                                if(components != null)
                                {
                                        components.Dispose();
                                }
                        }
                        base.Dispose( disposing );
                }

                #region Windows Form Designer generated code
                /// <summary>
                /// Required method for Designer support - do not modify
                /// the contents of this method with the code editor.
                /// </summary>
                private void InitializeComponent()
                {
                        this.cmbGodina = new
System.Windows.Forms.ComboBox();
                        this.cmbPredmet = new
System.Windows.Forms.ComboBox();
```

```
                this.cmbSmer = new System.Windows.Forms.ComboBox();
                this.btnPrikazi = new System.Windows.Forms.Button();
                this.cmbNovi = new System.Windows.Forms.ComboBox();
                this.cmbObrisi = new
ystem.Windows.Forms.ComboBox();
                this.btnNovi = new System.Windows.Forms.Button();
                this.btnObrisi = new System.Windows.Forms.Button();
                this.txtNoviBr1 = new System.Windows.Forms.TextBox();
                this.txtObrisiBr2 = new
ystem.Windows.Forms.TextBox();
                this.txtObrisiBr1 = new
ystem.Windows.Forms.TextBox();
                this.txtNoviBr2 = new System.Windows.Forms.TextBox();
                this.label1 = new System.Windows.Forms.Label();
                this.label2 = new System.Windows.Forms.Label();
                this.label3 = new System.Windows.Forms.Label();
                this.btnOdabir = new System.Windows.Forms.Button();
                this.lbGodina = new System.Windows.Forms.Label();
                this.lbSmer = new System.Windows.Forms.Label();
                this.lbPredmet = new System.Windows.Forms.Label();
                this.btnDodaj = new System.Windows.Forms.Button();
                this.btnBrisanje = new System.Windows.Forms.Button();
                this.listView1 = new System.Windows.Forms.ListView();
                this.columnHeader1 = new
System.Windows.Forms.ColumnHeader();
                this.columnHeader2 = new
System.Windows.Forms.ColumnHeader();
                this.columnHeader3 = new
System.Windows.Forms.ColumnHeader();
                this.columnHeader4 = new
System.Windows.Forms.ColumnHeader();
                this.SuspendLayout();
                //
                // cmbGodina
                //
                this.cmbGodina.Location = new
System.Drawing.Point(16, 80);
                this.cmbGodina.Name = "cmbGodina";
                this.cmbGodina.Size = new System.Drawing.Size(120,
21);
                this.cmbGodina.TabIndex = 0;
                this.cmbGodina.Text = "Godina";
                this.cmbGodina.SelectedIndexChanged += new
System.EventHandler(this.cmbGodina_SelectedIndexChanged);
                //
                // cmbPredmet
                //
```

```
                this.cmbPredmet.Location = new
System.Drawing.Point(344, 80);
                this.cmbPredmet.Name = "cmbPredmet";
                this.cmbPredmet.Size = new System.Drawing.Size(176,
21);
                this.cmbPredmet.TabIndex = 1;
                this.cmbPredmet.Text = "Predmet";
                this.cmbPredmet.SelectedIndexChanged += new
System.EventHandler(this.cmbPredmet_SelectedIndexChanged);
                //
                // cmbSmer
                //
                this.cmbSmer.Location = new System.Drawing.Point(152,
80);
                this.cmbSmer.Name = "cmbSmer";
                this.cmbSmer.Size = new System.Drawing.Size(176, 21);
                this.cmbSmer.TabIndex = 2;
                this.cmbSmer.Text = "Smer";
                this.cmbSmer.SelectedIndexChanged += new
System.EventHandler(this.cmbSmer_SelectedIndexChanged);
                //
                // btnPrikazi
                //
                this.btnPrikazi.Font = new
System.Drawing.Font("Microsoft Sans Serif", 8.25F,
System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point,
((System.Byte)(238)));
                this.btnPrikazi.Location = new
System.Drawing.Point(536, 80);
                this.btnPrikazi.Name = "btnPrikazi";
                this.btnPrikazi.Size = new System.Drawing.Size(88, 24);
                this.btnPrikazi.TabIndex = 4;
                this.btnPrikazi.Text = "Prikaži";
                this.btnPrikazi.Click += new
System.EventHandler(this.btnPrikazi_Click);
                //
                // cmbNovi
                //
                this.cmbNovi.Location = new System.Drawing.Point(152,
296);
                this.cmbNovi.Name = "cmbNovi";
                this.cmbNovi.Size = new System.Drawing.Size(224, 21);
                this.cmbNovi.TabIndex = 5;
                this.cmbNovi.Text = "Novi Profesor:";
                this.cmbNovi.SelectedIndexChanged += new
System.EventHandler(this.cmbNovi_SelectedIndexChanged);
                //
                // cmbObrisi
```

```
                        //
                        this.cmbObrisi.Location = new
System.Drawing.Point(152, 352);
                        this.cmbObrisi.Name = "cmbObrisi";
                        this.cmbObrisi.Size = new System.Drawing.Size(224, 21);
                        this.cmbObrisi.TabIndex = 6;
                        this.cmbObrisi.Text = "Obrisati Profesora:";
                        this.cmbObrisi.SelectedIndexChanged += new
System.EventHandler(this.cmbObrisi_SelectedIndexChanged);
                        //
                        // btnNovi
                        //
                        this.btnNovi.Location = new System.Drawing.Point(16,
296);
                        this.btnNovi.Name = "btnNovi";
                        this.btnNovi.Size = new System.Drawing.Size(120, 23);
                        this.btnNovi.TabIndex = 7;
                        this.btnNovi.Text = "Dodavanje Profesora";
                        this.btnNovi.Click += new
System.EventHandler(this.btnNovi_Click);
                        //
                        // btnObrisi
                        //
                        this.btnObrisi.Font = new System.Drawing.Font("Microsoft
Sans Serif", 8.25F, System.Drawing.FontStyle.Bold,
System.Drawing.GraphicsUnit.Point, ((System.Byte)(238)));
                        this.btnObrisi.Location = new System.Drawing.Point(560,
352);
                        this.btnObrisi.Name = "btnObrisi";
                        this.btnObrisi.Size = new System.Drawing.Size(64, 23);
                        this.btnObrisi.TabIndex = 8;
                        this.btnObrisi.Text = "Obriši";
                        this.btnObrisi.Click += new
System.EventHandler(this.btnObrisi_Click);
                        //
                        // txtNoviBr1
                        //
                        this.txtNoviBr1.Location = new
System.Drawing.Point(392, 296);
                        this.txtNoviBr1.MaxLength = 2;
                        this.txtNoviBr1.Name = "txtNoviBr1";
                        this.txtNoviBr1.Size = new System.Drawing.Size(64, 20);
                        this.txtNoviBr1.TabIndex = 9;
                        this.txtNoviBr1.Text = "Br. Grupa1";
                        this.txtNoviBr1.TextChanged += new
System.EventHandler(this.txtNoviBr1_TextChanged);
                        this.txtNoviBr1.Enter += new
System.EventHandler(this.txtNoviBr1_Enter);
```

```
                    //
                    // txtObrisiBr2
                    //
                    this.txtObrisiBr2.Location = new
System.Drawing.Point(480, 352);
                    this.txtObrisiBr2.Name = "txtObrisiBr2";
                    this.txtObrisiBr2.ReadOnly = true;
                    this.txtObrisiBr2.Size = new System.Drawing.Size(64, 20);
                    this.txtObrisiBr2.TabIndex = 10;
                    this.txtObrisiBr2.Text = "Br. Grupa2";
                    //
                    // txtObrisiBr1
                    //
                    this.txtObrisiBr1.Location = new
System.Drawing.Point(392, 352);
                    this.txtObrisiBr1.Name = "txtObrisiBr1";
                    this.txtObrisiBr1.ReadOnly = true;
                    this.txtObrisiBr1.Size = new System.Drawing.Size(64, 20);
                    this.txtObrisiBr1.TabIndex = 11;
                    this.txtObrisiBr1.Text = "Br. Grupa1";
                    //
                    // txtNoviBr2
                    //
                    this.txtNoviBr2.Location = new
System.Drawing.Point(480, 296);
                    this.txtNoviBr2.MaxLength = 2;
                    this.txtNoviBr2.Name = "txtNoviBr2";
                    this.txtNoviBr2.Size = new System.Drawing.Size(64, 20);
                    this.txtNoviBr2.TabIndex = 12;
                    this.txtNoviBr2.Text = "Br. Grupa2";
                    this.txtNoviBr2.TextChanged += new
System.EventHandler(this.txtNoviBr2_TextChanged);
                    this.txtNoviBr2.Enter += new
System.EventHandler(this.txtNoviBr2_Enter);
                    //
                    // label1
                    //
                    this.label1.AutoSize = true;
                    this.label1.Location = new System.Drawing.Point(16, 24);
                    this.label1.Name = "label1";
                    this.label1.Size = new System.Drawing.Size(87, 16);
                    this.label1.TabIndex = 13;
                    this.label1.Text = "Školska Godina:";
                    //
                    // label2
                    //
                    this.label2.AutoSize = true;
```

```
                this.label2.Location = new System.Drawing.Point(152,
24);
                this.label2.Name = "label2";
                this.label2.Size = new System.Drawing.Size(34, 16);
                this.label2.TabIndex = 14;
                this.label2.Text = "Smer:";
                //
                // label3
                //
                this.label3.AutoSize = true;
                this.label3.Location = new System.Drawing.Point(344,
24);
                this.label3.Name = "label3";
                this.label3.Size = new System.Drawing.Size(50, 16);
                this.label3.TabIndex = 15;
                this.label3.Text = "Predmet:";
                //
                // btnOdabir
                //
                this.btnOdabir.Location = new
System.Drawing.Point(536, 16);
                this.btnOdabir.Name = "btnOdabir";
                this.btnOdabir.Size = new System.Drawing.Size(88, 40);
                this.btnOdabir.TabIndex = 16;
                this.btnOdabir.Text = "Novi Prikaz Profesora";
                this.btnOdabir.Click += new
System.EventHandler(this.btnOdabir_Click);
                //
                // lbGodina
                //
                this.lbGodina.AutoSize = true;
                this.lbGodina.Font = new System.Drawing.Font("Microsoft
Sans Serif", 8.25F, System.Drawing.FontStyle.Bold,
System.Drawing.GraphicsUnit.Point, ((System.Byte)(238)));
                this.lbGodina.Location = new System.Drawing.Point(16,
48);
                this.lbGodina.Name = "lbGodina";
                this.lbGodina.Size = new System.Drawing.Size(36, 16);
                this.lbGodina.TabIndex = 17;
                this.lbGodina.Text = "label4";
                //
                // lbSmer
                //
                this.lbSmer.AutoSize = true;
                this.lbSmer.Font = new System.Drawing.Font("Microsoft
Sans Serif", 8.25F, System.Drawing.FontStyle.Bold,
System.Drawing.GraphicsUnit.Point, ((System.Byte)(238)));
```

```
                    this.lbSmer.Location = new System.Drawing.Point(152,
48);
                    this.lbSmer.Name = "lbSmer";
                    this.lbSmer.Size = new System.Drawing.Size(36, 16);
                    this.lbSmer.TabIndex = 18;
                    this.lbSmer.Text = "label4";
                    //
                    // lbPredmet
                    //
                    this.lbPredmet.AutoSize = true;
                    this.lbPredmet.Font = new
System.Drawing.Font("Microsoft Sans Serif", 8.25F,
System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point,
((System.Byte)(238)));
                    this.lbPredmet.Location = new
System.Drawing.Point(344, 48);
                    this.lbPredmet.Name = "lbPredmet";
                    this.lbPredmet.Size = new System.Drawing.Size(36, 16);
                    this.lbPredmet.TabIndex = 19;
                    this.lbPredmet.Text = "label4";
                    //
                    // btnDodaj
                    //
                    this.btnDodaj.Font = new System.Drawing.Font("Microsoft
Sans Serif", 8.25F, System.Drawing.FontStyle.Bold,
System.Drawing.GraphicsUnit.Point, ((System.Byte)(238)));
                    this.btnDodaj.Location = new System.Drawing.Point(560,
296);
                    this.btnDodaj.Name = "btnDodaj";
                    this.btnDodaj.Size = new System.Drawing.Size(64, 23);
                    this.btnDodaj.TabIndex = 20;
                    this.btnDodaj.Text = "Dodaj";
                    this.btnDodaj.Click += new
System.EventHandler(this.btnDodaj_Click);
                    //
                    // btnBrisanje
                    //
                    this.btnBrisanje.Location = new
System.Drawing.Point(16, 352);
                    this.btnBrisanje.Name = "btnBrisanje";
                    this.btnBrisanje.Size = new System.Drawing.Size(120,
23);
                    this.btnBrisanje.TabIndex = 21;
                    this.btnBrisanje.Text = "Brisanje Profesora";
                    this.btnBrisanje.Click += new
System.EventHandler(this.btnBrisanje_Click);
                    //
                    // listView1
```

```
                //
                this.listView1.Columns.AddRange(new
System.Windows.Forms.ColumnHeader[] {

            this.columnHeader1,

            this.columnHeader2,

            this.columnHeader3,

            this.columnHeader4});
                this.listView1.GridLines = true;
                this.listView1.Location = new System.Drawing.Point(16,
128);
                this.listView1.MultiSelect = false;
                this.listView1.Name = "listView1";
                this.listView1.Size = new System.Drawing.Size(608, 144);
                this.listView1.TabIndex = 22;
                this.listView1.View =
System.Windows.Forms.View.Details;
                //
                // columnHeader1
                //
                this.columnHeader1.Text = "Ime Profesora";
                this.columnHeader1.Width = 177;
                //
                // columnHeader2
                //
                this.columnHeader2.Text = "Prezime Profesora";
                this.columnHeader2.Width = 162;
                //
                // columnHeader3
                //
                this.columnHeader3.Text = "Br. Grupa1 (1. Semestar)";
                this.columnHeader3.Width = 132;
                //
                // columnHeader4
                //
                this.columnHeader4.Text = "Br. Grupa2 (2. Semestar)";
                this.columnHeader4.Width = 132;
                //
                // FormRaspored
                //
```

```
                this.AutoScaleBaseSize = new System.Drawing.Size(5,
13);
                this.ClientSize = new System.Drawing.Size(640, 390);
                this.Controls.Add(this.listView1);
                this.Controls.Add(this.btnBrisanje);
                this.Controls.Add(this.btnDodaj);
                this.Controls.Add(this.lbPredmet);
                this.Controls.Add(this.lbSmer);
                this.Controls.Add(this.lbGodina);
                this.Controls.Add(this.btnOdabir);
                this.Controls.Add(this.label3);
                this.Controls.Add(this.label2);
                this.Controls.Add(this.label1);
                this.Controls.Add(this.txtNoviBr2);
                this.Controls.Add(this.txtObrisiBr1);
                this.Controls.Add(this.txtObrisiBr2);
                this.Controls.Add(this.txtNoviBr1);
                this.Controls.Add(this.btnObrisi);
                this.Controls.Add(this.btnNovi);
                this.Controls.Add(this.cmbObrisi);
                this.Controls.Add(this.cmbNovi);
                this.Controls.Add(this.btnPrikazi);
                this.Controls.Add(this.cmbSmer);
                this.Controls.Add(this.cmbPredmet);
                this.Controls.Add(this.cmbGodina);
                this.Name = "FormRaspored";
                this.Text = "Raspored Predmeta po Profesorima";
                this.Load += new
System.EventHandler(this.FormRaspored_Load);
                this.ResumeLayout(false);

        }
        #endregion

        private void FormRaspored_Load(object sender,
System.EventArgs e)
        {
                ////this.oleDbSelectCommand8.Parameters.Add(new
System.Data.OleDb.OleDbParameter("ID_SmerSadrziPredmet",
System.Data.OleDb.OleDbType.Integer, 0, "ID_SmerSadrziPredmet"));
                this.SetXPStyle();
                this.ClearLabels();
                this.DisableAll();
                string s, appPath, appFull, appName = "SSluzba.exe";
                appFull = Application.ExecutablePath;
                appPath = appFull.Substring(0, appFull.Length -
appName.Length);
                s = appPath + "PredmetiIProfesori.mdb";
```

```
                    SSluzba.FormRaspored.connString = @"Jet OLEDB:Global
Partial Bulk Ops=2;Jet OLEDB:Registry Path=;Jet OLEDB:Database Locking
Mode=1;Jet OLEDB:Database Password=;Data Source="""" +s+ @"""";Password=;Jet
OLEDB:Engine Type=5;Jet OLEDB:Global Bulk
Transactions=1;Provider=""Microsoft.Jet.OLEDB.4.0"";Jet OLEDB:System
database=;Jet OLEDB:SFP=False;Extended Properties=;Mode=Share Deny
None;Jet OLEDB:New Database Password=;Jet OLEDB:Create System
Database=False;Jet OLEDB:Don't Copy Locale on Compact=False;Jet
OLEDB:Compact Without Replica Repair=False;User ID=Admin;Jet OLEDB:Encrypt
Database=False";
                    this.Raspored = new
StudSluzba.RasporedPredmetaPoProfesorima();
                    if
(this.Raspored.Initialize(SSluzba.FormRaspored.connString ))
                    {
                        // Ucitavanje Godine
                        for (long i=1, n =
this.Raspored.CountSkolskaGodina(); i<=n; i++)
                        {

        this.cmbGodina.Items.Add(this.Raspored.GetSkolskaGodina(i).getNazivSk
olskaGodina());
                        }
                        // Ucitavanje Godine end
                    }
                    else
                    {
                        System.Windows.Forms.MessageBox.Show("Nije
uspelo Initialize");
                    }
                }

                private void btnPrikazi_Click(object sender, System.EventArgs e)
                {
                    int r =
this.Raspored.Fill3BR(this.cmbPredmet.SelectedIndex +1);
                    if (r >= 0)
                    {
                        this.FillListView();
                        this.DisablePrikazi();

                        if (this.cmbGodina.SelectedItem.ToString() ==
this.Raspored.GetTekucaSkolskaGodina().getNazivSkolskaGodina())
                        {
                            this.EnableDodavanje();
                            this.EnableBrisanje();
                        }
                    }
```

```
                    //case
                    else
                    {
                            if (r == -1)

        System.Windows.Forms.MessageBox.Show("Fill3BR je pozvano u stanju u
kom to nije dozovljeno!");
                            else

        System.Windows.Forms.MessageBox.Show("Greska pri pristupu bazi!");
                    }
            }

            private void cmbPredmet_SelectedIndexChanged(object sender,
System.EventArgs e)
            {
                    string selText =
this.cmbPredmet.SelectedItem.ToString();
                    this.lbPredmet.Text = selText;
                    this.EnablePrikazi();
                    this.DisablePredmet();
            }

            private void cmbSmer_SelectedIndexChanged(object sender,
System.EventArgs e)
            {

                    //Ucitavanje Predmeta
                    this.cmbPredmet.Items.Clear();
                    this.lbSmer.Text =
this.cmbSmer.SelectedItem.ToString();

        if(this.Raspored.PredmetiFill2BR(this.cmbSmer.SelectedIndex +1) >= 0)
                    {
                            for (long i=1, n = this.Raspored.CountPredmet();
i<=n; i++)
                            {

        this.cmbPredmet.Items.Add(this.Raspored.GetPredmet(i).getNazivPredm
eta());
                            }
                            //Ucitavanje Predmeta end
                            this.EnablePredmet();
                            this.DisableSmer();
                    }
                    else
                    {
```

```
                                System.Windows.Forms.MessageBox.Show("Nije
uspelo PredmetiFill2BR!");
                        }
                }

                private void cmbGodina_SelectedIndexChanged(object sender,
System.EventArgs e)
                {
                        this.lbGodina.Text =
this.cmbGodina.SelectedItem.ToString();
                        //Ucitavanje Smera
                        this.cmbSmer.Items.Clear();
                        if
(this.Raspored.SmeroviFill1BR(this.cmbGodina.SelectedIndex + 1) >= 0)
                        {
                                for (long i=1, n = this.Raspored.CountSmer();
i<=n; i++)
                                {

        this.cmbSmer.Items.Add(this.Raspored.GetSmer(i).getNazivSmera());
                                }

                                this.EnableSmer();
                                this.DisableGodina();
                        }
                        else
                        {
                                System.Windows.Forms.MessageBox.Show("Nije
uspelo SmeroviFill1BR!");
                        }
                }

                private void cmbObrisi_SelectedIndexChanged(object sender,
System.EventArgs e)
                {
                        this.txtObrisiBr1.Text =
this.Raspored.GetProfesorPredajePredmet(this.cmbObrisi.SelectedIndex +
1).getBrGrupa1().ToString();
                        this.txtObrisiBr2.Text =
this.Raspored.GetProfesorPredajePredmet(this.cmbObrisi.SelectedIndex +
1).getBrGrupa2().ToString();
                        //Ucitavanje BrGrupa za Profesora  u cmbObrisi end
                        this.DisableCmbObrisi();
                        this.btnObrisi.Enabled = true;
                }

                private void btnOdabir_Click(object sender, System.EventArgs e)
                {
```

```
                this.listView1.Items.Clear();

                this.DisableAll();

                this.ClearLabels();

                this.cmbGodina.Text = "Godina";
                this.cmbSmer.Text = "Smer";
                this.cmbPredmet.Text = "Predmet";
                this.cmbObrisi.Text = "Obrisati Profesora:";
                this.cmbNovi.Text = "Novi Profesor:";
                this.txtNoviBr1.Text = "Br. Grupa1";
                this.txtNoviBr2.Text = "Br. Grupa2";
                this.txtObrisiBr1.Text = "Br. Grupa1";
                this.txtObrisiBr2.Text = "Br. Grupa2";

                this.EnableGodina();
        }

        private void cmbNovi_SelectedIndexChanged(object sender,
System.EventArgs e)
        {
                this.txtNoviBr1.Enabled = true;
                this.DisableCmbNovi();
        }

        private void btnNovi_Click(object sender, System.EventArgs e)
        {
                //Treba selektovati sve profesore koji ne predaju
odredjene SKOLSKE GODINE, na SMERU predaju PREDMET
                //Ucitavanje Profesora u cmbNovi
                this.cmbNovi.Items.Clear();
                for(long i=1,
n=this.Raspored.CountProfesorNEPredajePredmet(); i <= n; i++)
                {
                        string ime =
this.Raspored.GetProfesorNEPredajePredmet(i).getIme();
                        string prezime =
this.Raspored.GetProfesorNEPredajePredmet(i).getPrezime();
                        this.cmbNovi.Items.Add( ime + " " + prezime);
                }
                //Ucitavanje Profesorau cmbNoviend
                this.EnableCmbNovi();
                this.DisableObrisi();
        }

        private void btnDodaj_Click(object sender, System.EventArgs e)
        {
```

```
                    long ID_ProfesoraDodati =
this.Raspored.GetProfesorNEPredajePredmet(this.cmbNovi.SelectedIndex +
1).getID_Profesora();
                    if (this.Raspored.InsertProfesor(ID_ProfesoraDodati,
System.Convert.ToInt32(this.txtNoviBr1.Text),
System.Convert.ToInt32(this.txtNoviBr2.Text)) >=0 )
                    {

        System.Windows.Forms.MessageBox.Show("Uspelo je dodavanje
profesora!");
                        this.FillListView();
                    }
                    else
                    {
                        System.Windows.Forms.MessageBox.Show("Nije
uspelo dodavanje profesora!");
                    }
                    this.DisableNovi();
                }

                private void btnBrisanje_Click(object sender, System.EventArgs
e)
                {
                    //Ucitavanje Profesora u cmbObrisi
                    this.cmbObrisi.Items.Clear();
                    for(long i=1,
n=this.Raspored.CountProfesorPredajePredmet(); i <= n; i++)
                    {
                        string ime =
this.Raspored.GetProfesorPredajePredmet(i).getProfesor().getIme();
                        string prezime =
this.Raspored.GetProfesorPredajePredmet(i).getProfesor().getPrezime();
                        this.cmbObrisi.Items.Add( ime + " " + prezime);
                    }
                    //Ucitavanje Profesora u cmbObrisi end
                    this.EnableCmbObrisi();
                    this.DisableNovi();
                }

                private void txtNoviBr1_TextChanged(object sender,
System.EventArgs e)
                {
                    string input = this.txtNoviBr1.Text;
                    if (!this.IsNumber(input))
                    {
                        this.txtNoviBr1.Text = "";
                        this.txtNoviBr2.Enabled = false;
                    }
```

```
                else
                {
                        this.txtNoviBr2.Enabled = true;
                }
        }

        private void txtNoviBr1_Enter(object sender, System.EventArgs
e)
        {
                this.txtNoviBr1.Text = "";
                this.DisableCmbNovi();
        }

        private void txtNoviBr2_Enter(object sender, System.EventArgs
e)
        {
                this.txtNoviBr1.Enabled = false;
                this.txtNoviBr2.Text = "";
        }

        private void txtNoviBr2_TextChanged(object sender,
System.EventArgs e)
        {
                string input = this.txtNoviBr2.Text;
                if (!this.IsNumber(input))
                {
                        this.txtNoviBr2.Text = "";
                        this.btnDodaj.Enabled = false;
                }
                else
                {
                        this.btnDodaj.Enabled = true;
                }
        }

        private void btnObrisi_Click(object sender, System.EventArgs e)
        {
                long ID_ProfesoraObrisati =
this.Raspored.GetProfesorPredajePredmet(this.cmbObrisi.SelectedIndex +
1).getProfesor().getID_Profesora();
                if (this.Raspored.DeleteProfesor(ID_ProfesoraObrisati) >=
0)
                {


        System.Windows.Forms.MessageBox.Show("Brisanje profesora za dati
predmet je uspelo!");
                        this.FillListView();
```

```
                    }
                    else
                    {

        System.Windows.Forms.MessageBox.Show("Brisanje profesora za dati
predmet nije uspelo!");
                    }
                    this.DisableObrisi();
                }

                private void
oleDbDataAdapterIDProfesorPredajePredmet_RowUpdated(object sender,
System.Data.OleDb.OleDbRowUpdatedEventArgs e)
                {

                }

                private void oleDbDataAdapterProfesor_RowUpdated(object
sender, System.Data.OleDb.OleDbRowUpdatedEventArgs e)
                {

                }
        }
}
```

Student je u obavezi da preda seminarski rad na CD-u, kao što je i dato u sadržaju rada pod br.12, a čija je struktura sledeća:

12.SADRŽAJ PRILOŽENOG CD-a

1) Dokumentacija
2) Projekat poslovne logike urađen u Power Designer-u 9.5
3) Projekat aplikacije u Micorsoft Visual Studio 2003 (izvorni kod aplikacije + dll file koji implementira poslovnu logiku)
4) HTML Report za projekat poslovne logike
5) Instalacija aplikacije Studentska Služba
6) Instalacija .net Framework-a

13. LITERATURA

1. Ivana Stojanović, Dušan Surla, *Uvod u objedinjeni jezik modeliranja*, Grupa za informacione tehnologije, Novi Sad, 1999.
2. Mickey Williams, *Microsoft Visual C# .NET*
3. Robert B. Dunaway, *The Book of Visual Studio .NET—A Guide for Developers*

12.1. Korišćenje projektnih obrazaca (Use of software patterns)

Novi OO pristupi razvoju softvera i nove softverske arhitekture su danas opšte prihvaćene. Sve vodeće firme i asocijacije definišu razvojna okruženja i standarde zasnovane na ovakvim pristupima i arhitekturama.

U savremenim pristupima razvoju softverskih sistema razlikuju se sledeći koncepti:
1. Funkcionalna specifikacija sistema - definiše šta sistem treba da radi.

2.*Arhitektura sistema*[18].
3.Modeli sistema - opisuju softverski sistem sa različitih aspekata.
4.Proces razvoja sistema (metodologija) - koja definiše redosled aktivnosti koje treba preduzeti da bi se željeni sistem izgradio i koristio.

Arhitektura sistema definišući najznačajnije statičke i dinamičke karakteristike sistema, opisuje šta treba da se izgradi. Pored zahteva korisnika na arhitekturu sistema utiču: okruženje u kome se softver razvija, postojeći softver koga eventualno treba uključiti (softverska zaostavština - software legacy) i gotove softverske komponente, odnosno "aplikacioni kosturi" (frameworks) i **mustre - obrasci (*patterns*)**. Savremena razvojna okruženja predstavljaju generičke arhitekture koje u sebi sadrže mnoštvo "blokova" koji se mogu direktno koristiti, specijalizovati ili dograditi u procesu razvoja sopstvenog softverskog sistema.

Projektni obrasci

Pri projektovanju , veliku pomoć pružaju ***projektni obrasci (design patterns)***. Projektni obrazac je opis načina rešavanja nekog poznatog problema u različitim domenima.Projektni obrazac se može opisati kolaboracijom koju čine neke uloge i mehanizmi interakcije između njih. Te uloge u konkretnom problemu igraju konkretne klase iz sistema.

Projektni obrasci predstavljaju jednostavna i elegantna rešenja za neke opšte probleme koji se često susreću u različitim kontekstima softverskih sistema. Projektni obrazac sadrži opis nekog opšteg, često sretanog projektnog problema i njegovog rešenja, uključujući različite varijante tog rešenja.

Dakle, ***projektni obrazac*** zapravo i predstavlja ono što mu i ime kaže: ***obrazac za rešavanje nekog opšteg projektnog problema, koji se primenjuje u konkretnom kontekstu, tako što se uloge koje se pojavljuju u njemu konkretizuju klasama iz domena specifičnog problema***. Važno je zapamtiti da su projektni

[18] Dragica Radosav, *Softversko inženjerstvo*, Tehnički fakultet, Zrenjanin, 2001.,str.22-23.

obrasci rešenja za opšte probleme koja se generalno opisuju pomoću kolaboracija u kojima učestvuju neke apstraktne uloge.

U nastavku će se opisati sledeći projektni obrasci:

- Obrazac ***Singleton***
- Obrazac ***Command***
- Obrazac ***Template Method***
- Obrazac ***Composite***
- Obrazac ***Observer***
- Obrazac ***Adapter***
- Obrazac ***Facade***
- Obrazac ***Prototype***

Obrazac Singleton

Veoma često se javlja potreba da u programu postoji tačno jedna instanca neke klase, kao globalni, svima dostupni objekat. Potrebno je obezbediti sledeće:
-Da data klasa ima tačno jednu *instance*. To znači da sigurno postoji jedan objekat te klase i da nije moguće slučajno ili namerno napraviti druge objekte te klase.
-Da je način na koji se pristupa do tog jednog objekta unapred definisan i poznat. Pristup je jedoobrazan, te se ne mora razmišljati kako pristupiti tom jedinom objektu. Obezbeđuje se statičkom operacijom date klase koja se obično zove **Instance ( )**.

Praktično, svaki složeniji sistem ima potrebe za ovakvim pristupom. Obrazac ***Singleton***:
-Obezbeđuje kontrolisan pristup do jedinstvene instance klase.
-Smanjuje broj globalnih imena u programu.
-Dozvoljava redefinisanje operacija *Singleton* klase njenim nasleđivanjem.
-Dozvoljava i kreiranje većeg, ali kontrolisanog broja instance date klase.

Obrazac Command

Pri projektovanju mehanizama koji implementiraju funkcionalnosti aplikacije, naročito u slučaju interaktivnih

aplikacija, često postoji potreba da se zahtev za izvršavanje neke operacije enkapsulira u objekat. Dakle, umesto direktne veze između objekta-klijenta i objekta-servera i neposrednog poziva operacije servera od strane klijenta, taj zahtev za izvršavanje operacije predstavlja se objektom koji se naziva ***komanda***.

Ovaj pristup predstavlja projektni obrazac ***Command*** čija je suština da objekt-klijent, umesto direktnog poziva operacije objekta-servera, kreira objekat-komandu koju potom daje na izvršavanje. Konkretne klase koje implementiraju komande obično imaju mnogo zajedničkih osobina koje se lokalizuju u apstraktnu osnovnu klasu.

Obrazac Template Method

Svrha projektnog obrasca *Template Method* jeste da se u nekoj operaciji osnovne klase definiše skelet nekog algoritma, tj. fiksni redosled izvršavanja njegovih koraka, ali da se izvedenim klasama ostavi mogućnost da redefinišu neke od tih koraka u skladu sa njihovim potrebama.

Ovaj obrazac se primenjuje kada je potrebno fiksirati određeni deo nekog algoritma, a omogućiti variranje nekih njegovih koraka. To se dešava kada se zajedničko ponašanje izvedenih klasa može definisati kao nekakav skelet i lokalizovati u osnovnu klasu. Ovakav slučaj je posledica dobre algoritamske dekompozicije pojedinih operacija različitih klasa i generalizacije njihovog ponašanja. Ideja je da se najpre uoče sličnosti i razlike u tim ponašanjima, zatim da se sličnosti lokalizuju u operaciju osnovne klase, a razlike definišu kao promenljivi primitivni koraci.

Ovaj obrazac se veoma često koristi u mehanizmima okruženja sa bibliotekama za razne namene (*framework*). U njima osnovne klase definišu algoritme, tj. redosled koraka, a često ostavljaju mogućnost da se u određenim tačkama tih algoritama pozovu operacije izvedenih klasa koje su u mogućnosti da prošire osnovno ponašanje. Ovakve operacije koje se pozivaju u određenim tačkama algoritma, imaju neko podrazumevano ponašanje, ali se mogu redefinisati po potrebi, nazivaju se "operacije-kukice" (*hook operations*).

Prema tome, šablonska metoda može da poziva sledeće vrste operacija kao korake svog algoritma:
-konkretne metode koje su definisane u osnovnoj klasi i ne mogu se redefinisati;
-primitivne, tj.apstraktne operacije koje se moraju definisati u izvedenim klasama;
-kukice, tj. operacije koje se mogu, ali ne moraju redefinisati u izvedenim klasama.

Prilikom definisanja šablonske metode i njene osnovne klase, potrebno je naglasiti koje operacije spadaju u koju od ovih kategorija, kako bi korisniku bilo jasno šta mora, a šta može da redefiniše.

Obrazac Composite

Composite je isključivo strukturalni obrazac, tj. odnosi se samo na mogućnost formiranja složene strukture i tretiranje svih njenih elemenata na isti način. Kako će se implementirati određno ponašanje njenih elemenata, tj. kojim će se redom obilaziti čvorovi stabla da bi se obezbedila odgovarajuća funkcionalnost, zavisi od samih potreba i to ovaj obrazac ne prejudicira. Može postojati i više operacija koje istu strukturu obilaze na različite načine, čime se dobijaju željeni različiti efekti.

Ovaj obrazac omogućuje formiranje rekurzivnih hijerarhija u kojima se na očekivanom mestu primitivnog objekta, može pojaviti i kompozitni objekat. Ovakav pristup pojednostavljuje kod klijenta, jer oni ne moraju da primećuju razliku između kompozitnih i primitivnih elemenata.

Ovaj obrazac je prirodna objektno orijentisana nadogradnja tradicionalnih tehnika rekurzivnog formiranja struktura i njihovog obilaska.

Ovaj obrazac se često susreće u sistemima (npr. grafičkim editorima slika - crtanje, pomeranje, promena veličine itd.)

Obrazac Observer

U opštem slučaju obrazac Observer se koristi kada postoji više posmatrača koji su zainteresovani za promenu sadržaja nekog izvora, pri čemu je broj posmatrača i njihova raznovrsnost proizvoljna. Zato u opštem slučaju, u ovom obrascu učestvuju sledeće uloge:
- Izvor (ili subjekat, *model*),
- Posmatrač (*view*),
- Kontrolor *(controller)*.

Zbog ovakvih opštih uloga ovaj obrazac se naziva i *Model-View-Controller* (MVC).

Ovaj obrazac se često koristi u okruženjima koja razdvajaju sadržaj svog modela od samog prikaza koji može biti raznovrstan.

Obrazac Adapter

Suština ovog obrasca je da konvertuje (adaptira) interfejs neke klase u neki drugi interfejs, kako bi se omogućila saradnja klasa koje inače ne bi zajedno radile zbog neodgovarajućih interfejsa.

U opštem slučaju, obrazac Adapter primenjuje se kad god je potrebno iskoristiti postojeću klasu čiji interfejs ne odgovara potrebama ostatka sistema, ili kada je potrebno napraviti klasu upotrebljivu u različitim okruženjima, koja će u budućim proširenjima sistema ili njegovim verzijama sarađivati sa klasama čiji se interfejsi ne mogu predvideti, tj. koje nemaju ili neće obavezno imati kompatibilne interfejse.

Obe varijante ovog obrasca imaju i prednosti i mane. Prva varijanta, sa višestrukim izvođenjem, dozvoljava adapterskoj klasi da redefiniše operacije i jedne i druge strane, podrazumeva postojanje samo jednog objekta, bez postojanja veza kao kod druge varijante, i dozvoljava dvosmerne adaptacije. Druga varijanta dozvoljava da objekat adapterske klase bude vezan za objekte osnovne adaptirane klase ali i za objekte njenih izvedenih klasa, što znači da dozvoljava polimorfizam adaptirane strane.

Obrazac Facade

Složeni sistem se dekomponije na podsisteme da bi se pojednostavio. Podsistemi obično sadrže mnoštvo klasa koje su čvrsto spregnute da bi ispunile odgovarajuće odgovornosti. Cilj svake dobre dekompozicije jeste da se sprege između delova sistema što više oslabe radi bolje kontrole.

Kada mnoštvo klijentskih klasa van opsega posmatranog sistema direktno pristupa odgovarajućim klasama datog podsistema koristeći njihove usluge, veze su složene, teške za kontrolu i održavanje.

Svrha ovog obrasca je obezbeđivanje jedinstvenog interfejsa prema čitavom skupu interfejsa iz datog podsistema. Fasadana klasa predstavlja interfejs višeg nivoa koji olakšava korišćenje datog podsistema i čini veze tog podsistema ka ostatku sistema labavijim, taj podsistem čini lakšim za modifikovanje, a ostatak sistema nazavisnijim. Sem toga, ovakav pristup može značajno da skrati i vreme prevođenja velikih programa, jer smanjuje zavisnosti između klasa.

U opštem slučaju, obrazac *Facade* koristi se kada je potrebno oslabiti veze prema nekom podsistemu i pojednostaviti pristup do njega. Naime, primena ostalih projektnih obrazaca često uzrokuje pojavu velikog broja malih klasa, što usložnjava njihovo korišćenje. Slično, kada se arhitektura sistema pravi po slojevima, onda fasadna klasa može da predstavlja jedinstvenu ulaznu tačku nekog sloja.

Ovaj obrazac predstavlja još jednu podršku enkapsulaciji na višem nivou granularnosti (na nivou podsistema). Podsistem se na jeziku UML modeluje paketom sa stereotipom <<*subsystem*>>, pa fasadna klasa treba da bude javna klasa tog paketa. Ostale klase koje predstavljaju njegovu implementaciju i koje treba zaštititi od pristupa spolja, radi lakše modifikacije podsistema i njegove prenosivosti, mogu se proglasiti privatnim.

Obrazac Prototype

Ideja i suština obrasca *Prototype* je da se pravljenje objekata klase poveri posebnom objektu date klase, koji predstavlja "prototipski" objekat te klase i koji se može klonirati (kopirati). Tako se klasa koja pravi i koristi konkretne objekte oslanja samo na apstraktni interfejs osnovne klase date hijerarhije i njenu operaciju za kloniranje, kao i na postojanje posebnih prototipskih objekata.

U opštem slučaju, ovaj obrazac se koristi kad god je potrebno da klijent bude nazavisan od načina izgradnje objekata klasa iz neke hijerarhije. Ovaj obrazac omogućuje konfigurisanje klijenta koji izgrađuje složene strukture objekata iz neke hijerarhije čak i u vreme izvršavanja, prostim registrovanjem novih prototipskih objekata koji zadovoljavaju isti interfejs. Osim toga, on omogućuje, u nekim slučajevima, da se ne mora praviti posebna paralelna hijerarhija klasa koje su zadužene za izgradnju specifičnih objekata iz date hijerarhije. Operacija kloniranja posebno je pogodna kod složenih struktura objekata, čime se klijent koji pravi složeni objekat rasterećuje brige o izgradnji te složene strukture. Ovaj je obrazac takođe pogodan kad treba praviti objekte iste klase, ali sa različitim parametrima konstrukcije ili različitim strukturama. Pogodnost obrasca je što od klijenta sakriva hijerarhiju i strukturu objekata koji se prave, kao i njihove specifičnosti.

Kada se implementira ovaj obrazac, može se kao mesto za registrovanje i manipulisanje prototipskim objektima predvideti posebna klasa, tzv. *menadžer prototipova*, koji je često *Singleton*. Ovaj menadžer preslikava dati ključ u reference na željeni prototipski objekat koji se potom klonira.

Najozbiljniji korak u implementaciji ovog obrasca jeste implementacija operacije kloniranja u svakoj konkretnoj klasi date hijerarhije. Pri kloniranju objekta, treba voditi računa o tome da li je potrebno tzv. plitko ili duboko kopiranje objekta. Problem može biti i prosleđivanje argumenata operaciji kloniranja, jer broj i tipovi tih argumenata mogu da zavise od konkretne klase.

Ovaj obrazac ima mnogo primena u visoko konfigurabilnim sistemima. Jedan od najpoznatijih primera je upotreba prototipskih objekata kod grafičkih korisničkih interfejsa u kojima se mogu konfigurisati različite alatke (*tools*) čijim pritiskom korisnik pravi odgovarajuće grafičke ili druge elemente. U tom slučaju, alatke ne moraju znati sa čim konkretno rade, jer se manipulacija napravljenim elementima obavlja na isti način, preko zajedničkog interfejsa klasa tih elemenata. Zbog toga alatke ne moraju da znaju ni šta zapravo rade, pa se to, kao i mogućnost dinamičke konfiguracije alatki, postiže primenom obrasca Prototype.

Literatura:

1.Gamma E.,Helm R.,Johnson R.,Vlissides J., *Design Patterns*, Addison-Wesley,1995.
2.Milićev D., *Objektno orijentisano modelovanje na jeziku UML*, Mikroknjiga, 2001

SADRŽAJ: