



RAZVOJ APLIKATIVNOG SOFTVERA

Violeta Tomašević



UNIVERZITET SINGIDUNUM

Violeta Tomašević

RAZVOJ APLIKATIVNOG SOFTVERA

Peto izdanje

Beograd, 2022.

RAZVOJ APLIKATIVNOG SOFTVERA

Autor:

dr Violeta Tomašević

Recenzenti:

dr Boško Nikolić

dr Dejan Živković

Izdavač:

UNIVERZITET SINGIDUNUM

Beograd, Danijelova 32

www.singidunum.ac.rs

Za izdavača:

dr Milovan Stanišić

Priprema za štampu:

Violeta Tomašević

Dizajn korica:

Aleksandar Mihajlović, MA

Godina izdanja:

2022.

Tiraž:

700 primeraka

Štampa:

Caligraph, Beograd

ISBN: 978-86-7912-551-4

Copyright:

© 2022. Univerzitet Singidunum

Izdavač zadržava sva prava.

Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljeno.

Predgovor

Ova knjiga je nastala kao rezultat potrebe za odgovarajućim pisanim materijalom iz predmeta Razvoj aplikativnog softvera koji autor drži na četvrtoj godini Fakulteta za informatiku i računarstvo i Tehničkog fakulteta Univerziteta Singidunum u Beogradu. Pri pisanju je učinjen napor da knjiga bude prihvatljiva za čitaoce bez nekog većeg predznanja iz oblasti razvoja softvera. Namenjena je i prilagođena prosečnom studentu, jer je osnovni cilj autora bio da svi studenti koji slušaju predmet Razvoj aplikativnog softvera mogu na razumljiv i lak način da savladaju predviđeno gradivo.

Knjiga je podeljena u devet poglavlja kroz koja se prati ceo životni ciklus softvera.

U uvodnom poglavlju objašnjeni su pojam softvera i njegovo mesto u računarskom okruženju. Posebna pažnja je posvećena pristupu rešavanju složenih problema, tj. analizi problema i sintezi rešenja. Ukazano je na osnovne aspekte o kojima se mora voditi računa pri izradi softvera. Na kraju poglavlja, opisane su uloge pojedinih učesnika u procesu razvoju softvera.

Drugo poglavlje je posvećeno procesu razvoja softvera. U njemu su izložene faze u procesu razvoja, kao i načini planiranja razvoja. Detaljno je predstavljen veći broj tradicionalnih metoda modelovanja procesa razvoja, kao što su kaskadni model, V model, spiralni model, RUP i dr. Osim tradicionalnom pristupu, pažnja je posvećena i agilnim metodama razvoja, posebno ekstremnom programiranju.

U trećem poglavlju opisana je prva faza u procesu razvoja softvera, tj. analiza zahteva. Ukazano je na sve aktivnosti koje treba sprovesti u ovoj fazi, počevši od prikupljanja zahteva od korisnika i modelovanja ponašanja sistema različitim metodama, pa do formulisanja i dokumentovanja zahteva. Na kraju su opisani postupci verifikacije i validacije zahteva kojima se potvrđuje da se sistem razvija na pravi način i da su zahtevi ispravno definisani.

Četvrto poglavlje se bavi postupcima projektovanja sistema na osnovu specifikacije zahteva koja je rezultat prve faze u razvoju softvera. Posebno je istaknut značaj modularnosti u projektovanju. Centralni deo poglavlja opisuje najznačajnije strategije projektovanja, kao što su cevi i filtri, klijent-server arhitektura, objektno-orijentisan pristup i dr. Izlaganja strategija su praćena odgovarajućim primerima.

U petom poglavlju su iznete osnove UML jezika za modelovanje koji danas predstavlja najčešće korišćeni postupak modelovanja različitih faza procesa razvoja

softvera. Opisano je šest osnovnih dijagrama, uključujući dijagram klasa, dijagram slučajeva korišćenja, dijagram aktivnosti, itd.

Šesto poglavlje je posvećeno implementaciji softvera. U njemu je najveća pažnja posvećena pisanju programa i izradi odgovarajuće programske dokumentacije. Na kraju je navedeno o čemu sve treba voditi računa kako bi napisani program bio što kvalitetniji.

U sedmom poglavlju, detaljno je opisana faza testiranja softvera. Na početku su uvedeni pojmovi greške i otkaza i izneta njihova klasifikacija. U centralnom delu, opisane su vrste testiranja, tj. jedinično, integraciono i sistemsko testiranje. U okviru jediničnog testiranja predstavljeni su metodi „crne“ i „bele kutije“. Nakon toga, objašnjeni su različiti metodi integracije (od vrha ka dnu, od dna ka vrhu i dr.), kao i postupci sistemskog testiranja (funkcionalno testiranje, testiranje performansi i dr.). U poslednjem delu poglavlja opisan je način izvođenja testiranja.

Osmo poglavlje se bavi problemima isporuke i održavanja softvera. U fazi isporuke, posebna pažnja je posvećena obuci korisnika i pratećoj dokumentaciji. Faza održavanja je prikazana kroz vrste održavanja, probleme i troškove održavanja.

Poslednje, deveto poglavlje analizira kvalitet softvera. Najpre se analiziraju različiti aspekti sa kojih se kvalitet može posmatrati, a zatim se detaljno predstallaju najznačajniji modeli kvaliteta čiji je cilj što objektivnija procena softvera.

Na kraju udžbenika, nalazi se Dodatak koji je namenjen studentima Univerziteta Singidunum koji polažu predmet Razvoj aplikativnog softvera. U dodatku su date opšte smernice za izradu projekta koji predstavlja sastavni deo ispita. One treba da pomognu studentima da bolje razumeju proces razvoja softvera. Smernicama su obuhvaćeni samo oni delovi procesa razvoja koji su izvodljivi u datim uslovima, tj. mogu se sprovesti u fakultetskom okruženju.

Biću zahvalna svima onima koji mi ukažu na greške ili daju korisne savete za buduće ispravke i dopune ovog materijala.

Beograd, septembar 2017. god.

Autorka

SADRŽAJ

Predgovor	iii
1 Uvod	9
2 Proces razvoja softvera	15
2.1 Tradicionalne metode modelovanja	17
2.1.1 Kaskadni model	17
2.1.2 V model	19
2.1.3 Fazni razvoj	21
2.1.3.1 Inkrementalni fazni razvoj	23
2.1.3.2 Iterativni fazni razvoj	24
2.1.4 Prototipski model	25
2.1.5 Transformacioni model	27
2.1.6 Spiralni model	28
2.1.7 RUP	30
2.2 Agilne metode	33
2.2.1 Ekstremno programiranje	35
3 Analiza zahteva	43
3.1 Prikupljanje zahteva	46
3.1.1 Vrste zahteva	48
3.1.2 Razrešavanje konflikata	50
3.1.3 Prototipovi zahteva	51
3.2 Modelovanje ponašanja	53
3.2.1 ER dijagrami	53
3.2.2 Tragovi događaja	55
3.2.3 Konačni automati	56

3.3 Formulisanje zahteva	57
3.3.1 Dokumentovanje zahteva	57
3.3.2 Kvalitet zahteva	62
3.4 Validacija i verifikacija zahteva	63
3.4.1 Validacija zahteva	63
3.4.2 Verifikacija specifikacije	65
4 Projektovanje sistema	67
4.1 Modularnost u projektovanju	69
4.2 Strategije projektovanja	70
4.2.1 Cevi i filtri	72
4.2.2 Slojevita arhitektura	73
4.2.3 Klijent-server arhitektura	74
4.2.4 Ravnopravni pristup	75
4.2.5 Arhitektura zasnovana na događajima	76
4.2.6 Objektno-orijetisani pristup	76
5 UML modelovanje	79
5.1 Dijagrami slučajeve korišćenja	79
5.2 Dijagrami klase	83
5.3 Dijagrami sekvence	89
5.4 Dijagrami aktivnosti	93
5.5 Dijagrami komponenata	102
5.6 Dijagrami raspoređivanja	106
6 Implementacija softvera	111
6.1 Pisanje programa	113
6.1.1 Strukture podataka	115
6.1.2 Algoritmi	117
6.1.3 Kontrolne strukture	119
6.2 Programska dokumentacija	120
6.2.1 Unutrašnja dokumentacija	121
6.2.2 Spoljašnja dokumentacija	124

6.3 Kvalitet programiranja	125
7 Testiranje softvera	127
7.1 Greške i otkazi	128
7.1.1 Klasifikacija grešaka	129
7.2 Vrste testiranja	132
7.2.1 Jedinično testiranje	134
7.2.1.1 Metod „crne kutije“	137
Podela na klase ekvivalencije	139
Analiza graničnih vrednosti	141
Uzročno-posledični grafovi	142
7.2.1.2 Metod „bele kutije“	145
Pokrivanje iskaza	146
Pokrivanje odluka	147
Pokrivanje uslova	148
7.2.2 Integraciono testiranje	149
7.2.2.1 Integracija po principu „velikog praska“	149
7.2.2.2 Integracija od dna ka vrhu	150
7.2.2.3 Integracija od vrha ka dnu	153
7.2.2.4 „Sendvič“ integracija	154
7.2.3 Sistemsko testiranje	155
7.2.3.1 Funkcionalno testiranje	156
7.2.3.2 Testiranje performansi	157
7.2.3.3 Testiranje prihvatljivosti	159
7.2.3.4 Instalaciono testiranje	161
7.3 Proces testiranja	161
7.3.1 Plan testiranja	162
7.3.1.1 Završetak testiranja	163
7.3.2 Specifikacija testova	166
7.3.3 Realizacija testiranja	168
7.3.4 Evaluacija rezultata testiranja	170

8 Isporuka i održavanje softvera	173
8.1 Isporuka sistema	173
8.1.1 Obuka korisnika	174
8.1.2 Dokumentacija	177
8.2 Održavanje sistema	180
8.2.1 Vrste održavanja	181
8.2.2 Problemi održavanja	183
8.2.3 Troškovi održavanja	185
9 Kvalitet softvera	187
9.1 Pogledi na kvalitet softvera	187
9.1.1 Kvalitet proizvoda	188
9.1.2 Kvalitet procesa razvoja	188
9.1.3 Kvalitet sa stanovišta okruženja	189
9.2 Modeli kvaliteta	190
9.2.1 McCall-ov model	190
9.2.2 Boehm-ov model	194
9.2.3 Dromey-ov model	196
9.2.4 Model ISO 9126	198
Dodatak	203
Literatura	

1 Uvod

Intenzivan razvoj informatike i računarstva u poslednjim decenijama uveo je softver u sve oblasti života. Razvijeni su brojni softverski proizvodi sa vrlo različitim namenama. Softver je postao neohodan u svim oblastima društva: privredi, obrazovanju, zdravstvu, medijima i komunikacijama, poslovanju, politici, itd.

Generalno posmatrano, mogu se izdvojiti dve vrste softvera: sistemski i aplikativni softver. Pod *sistemskim softverom* podrazumevaju se programi niskog nivoa (*low-level*) koji omogućavaju rad sa računarskom opremom (hardverom). Može se reći da sistemski softver na neki način „oživljava“ hardver i omogućava njegovo korišćenje. U sistemski softver se ubrajaju operativni sistemi, razvojni alati za generisanje programa na različitim programskim jezicima, mrežni softveri, softveri za upravljanje bazama podataka, programske biblioteke, razne vrste prevodioca, alati za testiranje programa, itd. Sistemski softver obezbeđuje osnovnu funkcionalnost koja čini platformu za rad aplikativnog softvera. *Aplikativni softver* čine programi napravljeni za specifičnu svrhu, prema potrebama korisnika. Ova vrsta softvera nije u direktnoj vezi sa hardverom, već se u svom izvršavanju oslanja na sistemski softver, posebno na operativni sistem. Aplikativni softver obuhvata poslovne aplikacije opšte namene (tekst procesore, aplikacije za tabelarna izračunavanja, grafičke aplikacije i sl.), aplikacije za kućnu upotrebu (igrice, edukativne aplikacije i dr.), industrijski softver, uslužni softver, web aplikacije, itd. Odnos sistemskog i aplikativnog softvera je prikazan na slici 1.1.

U nekim slučajevima, nema jasne granice između sistemskog i aplikativnog softvera. Na primer, ne postoji saglasnost svetskih stručnjaka oko toga da li je pretraživač *Internet Explorer* deo operativnog sistema *Windows* ili nije.



Slika 1.1 Mesto aplikativnog softvera u računarskom okruženju

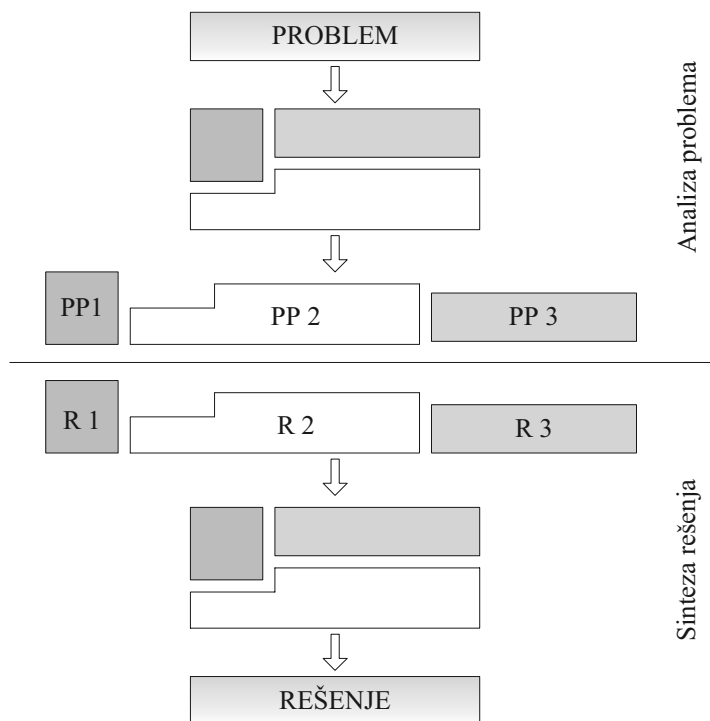
Potreba za razvojem aplikativnog softvera može da nastane kada korisnik želi da reši neki problem ili da dobije neku uslugu. Međutim, pre izrade softvera, posebno ukoliko se radi o složenom problemu, neophodno je sprovesti analizu sistema. Pod *analizom sistema* se podrazumeva sagledavanje problema i njegovo razlaganje na potprobleme koji su razumljivi i rešivi. Posebna pažnja u analizi se mora posvetiti vezama između potproblema, jer one mogu biti ključni faktor u nalaženju kompletnog rešenja. Nakon što se problem razloži na potprobleme koji mogu da se reše (sa jasno definisanim međusobnim vezama), pristupa se *sintezi rešenja*. Svaki potproblem se najpre samostalno rešava, a zatim se od dobijenih parcijalnih rešenja formira kompletno rešenje problema. Postupak analize i sinteze je ilustrovan na slici 1.2, na primeru problema koji se može razložiti na tri potproblema: PP1, PP2 i PP3.

Rešenje potproblema PP1 je R1, dok su R2 i R3 rešenja potproblema PP2 i PP3, respektivno. Spajanjem rešenja R1, R2 i R3 dobija se rešenje polaznog problema.

Rezultat sinteze rešenja ukazuje na to da li posmatrani problem treba da se rešava izradom odgovarajućeg softvera, ili se može rešiti na neki drugi način. Ako se utvrdi da je softver potreban, pristupa se njegovom razvoju.

Svaki problem koji se rešava izradom softvera, može se rešiti na više načina. Načini se međusobno razlikuju po efikasnosti, preciznosti, razumljivosti, korisnosti, mogućnosti modifikovanja i drugim osobinama. Stoga, izrada softvera zahteva posjedovanje znanja, ali i domišljatosti i veštine. Osnovni cilj u izradi

softvera jeste da softver bude sveobuhvatan, stabilan, razumljiv, da se lako održava i radi efiksano ono zbog čega je napravljen. Nerazumljivost napisanog programa narušava njegov kvalitet, iako ponekad postoji pogrešno mišljenje da je takav program izuzetan, zato što niko osim autora ne može da ga shvati.



Slika 1.2 Analiza problema i sinteza rešenja

Danas u svetu postoji veliki broj proizvođača softvera. Svaki od njih se trudi da napravi softver bez mana. Međutim, praktično ne postoji softver bez nedostataka. Neki nedostaci se pojave odmah nakon puštanja softvera u rad, dok je za druge potrebno znatno više vremena. Ni za jedan softver se ne može garantovati da za svo vreme primene neće ispoljiti nijedan nedostatak.

Da bi softver bio što bolji, pri njegovoj izradi mora se voditi o raznim aspektima, kao što su:

- neovlašćena upotreba sistema
- tržište softvera
- obezbeđivanje kvaliteta

Za softver je važno da dobro radi u svim uslovima u kojima može da se nađe. Zato, pri njegovoj izradi treba voditi računa i o mogućoj neočekivanoj upotrebi sistema. Do neočekivane upotrebe može da dođe zbog pokušaja zloupotrebe softvera, ili zbog nestručnog rukovanja od strane korisnika. Na primer, čest je slučaj zlonamernih napada na web stranice državnih organa ili drugih organizacija. Zato, pri izradi ovih web stranica, treba sprovesti određene mere zaštite sadržaja. U praksi je čest i slučaj nestručnog rukovanja, posebno kada se korisnici susreću sa novom tehnologijom.

Zbog velike konkurencije na tržištu softvera, da bi opstali, proizvođači moraju u relativno kratkim vremenskim periodima da isporučuju nove proizvode. To im ostavlja nedovoljno vremena za testiranje programa, pa su greške u programima češće. Kada se uoči neki nedostatak u softveru, proizvođač mora vrlo brzo da reaguje i da ga otkloni. On to može da učini na dva načina: da ispravi grešku menjanjem dela postojećeg softvera, ili da generiše novi softver. Donošenje prave odluke u ovoj situaciji je od izuzetne važnosti. Naime, ako je nedostatak konceptijske prirode, tj. ako je nastao zbog loše isprojektovanog sistema, njegovo otklanjanje izmenom dela kôda može da prouzrokuje nove, možda i veće probleme. Ima situacija u kojima je isplativije ponovo razviti ceo sistem, nego popravljati stari koji se ne može popraviti. U svakom slučaju, donošenje odluke je najbolje prepustiti nekom iskusnom projektantu.

O kvalitetu softverskog proizvoda mora se voditi računa tokom celog procesa razvoja softvera. Dokazano je da što nedostatak ostane duže vremena neotkriven, to njegovo otklanjanje više košta. Na primer, troškovi ispravljanja greške u fazi analize su deset puta manji od troškova ispravljanja iste greške nakon isporuke. Zato je jedan od primarnih ciljeva u svim fazama razvoja otkrivanje i otklanjanje grešaka.

U proces definisanja i stvaranja softverskog proizvoda uključeno je više učesnika koji mogu da imaju sledeće uloge:

- kupac
- razvojni tim
- korisnik

Kupac je kompanija, organizacija ili pojedinac koji naručuje softver i finansira njegov razvoj. Softver se pravi prema potrebama kupca i namenjen je rešavanju nekog njegovog problema. Kupac kontaktira kompaniju, organizaciju ili pojedinca koji će napraviti željeni softver i u tu svrhu, ova kompanija formira *razvojni tim*. *Korisnik* predstavlja jednog ili više pojedinaca koji će stvarno koristiti sistem. Korisnik i kupac obično pripadaju istoj kompaniji ili organizaciji. Na primer, ako je

potrebno razviti aplikaciju za podršku šalterskom radu u banci, kupca predstavlja rukovodstvo banke, a korisnika šalterski službenici.

Za uspešnost projekta, veoma je važna dobra komunikacija između svih učesnika. Nakon postizanja dogovora o realizaciji softvera, kupac i organizacija koja razvija softver potpisuju ugovor. Zatim, pošto je razumeo šta želi kupac, razvoji tim kontaktira krajnje korisnike da bi prikupio informacije o načinu funkcionisanja sistema u radnom okruženju. Na osnovu njih, razvojni tim realizuje softverski proizvod. Kada sistem bude gotov, isporučuje se, i kupac potpisuje papire o tehničkom prijemu softvera.

Opisane uloge ne moraju uvek da pripadaju različitim učesnicima. U nekim, uglavnom manjim projektima, ista osoba ili grupa može da ima dve, ili čak sve tri uloge. Na primer, ako neka kompanija ima svoj sektor za razvoj, može se doneti odluka da taj sektor razvija softver koji će biti namenjen praćenju troškova sopstvenih projekata. U ovom slučaju, sektor postaje i kupac i razvojni tim i krajnji korisnik.

U poslednje vreme, odnosi između uloga postaju znatno složeniji. To se dešava zato što se kupci i korisnici sve više uključuju u proces razvoja softvera.

Kupac i korisnik imaju veoma značajne uloge u definisanju sistema. Međutim, u realizaciji sistema, glavnu ulogu ima razvojni tim. Njega čine softverski inženjeri specijalizovani za različite aspekte razvoja. Broj članova u timu zavisi od veličine softverskog sistema koji se razvija. U većim projektima, mogu se izdvojiti sledeće uloge članova razvojnog tima:

- *analitičar* – član razvojnog tima koji obavlja prve korake u razvoju. U komunikaciji sa korisnikom, analitičar utvrđuje šta korisnik želi i dokumentuje njegove zahteve. Nakon definisanja zahteva, analitičar radi zajedno sa projektantima na generisanju opisa funkcija sistema.
- *projektant* – projektuje, tj. dizajnira sistem prema zadatim funkcionalnim zahtevima, tako da kasnije programeri mogu lako da ga implementiraju.
- *programer* – piše programski kôd na odgovarajućem programskom jeziku koristeći predviđeno razvojno okruženje. Kôd mora da odgovara projektu sistema urađenom na osnovu korisničkih zahteva.
- *inženjer za testiranje* – testira programski kôd koji je napisao programer. Prvo testiranje obično obavljaju sami programeri, a zatim se kôd prosleđuje na detaljnije testiranje inženjerima za testiranje. Pri integraciji sistema, tj. spajanju programskih modula u jednu celinu, timovi za testiranje rade na verifikaciji sistema. Zatim sledi validacija, tj. provera da li sistem ispunjava zahteve korisnika.

- *inženjer za isporuku i održavanje* – isporučuje i instalira softver u radnom okruženju, obučava korisnike za operativno korišćenje sistema i bavi se poslovima održavanja sistema nakon njegove isporuke.

Navedene razvojne uloge kod velikih projekata obavljaju različite osobe, dok kod manjih projekata, ista osoba može da ima više uloga.

Osim navedenih uloga, koje direktno imaju udela u realizaciji softverskog proizvoda, neophodno je obezbediti i podršku skladnom radu razvojnog tima. Kod velikih projekata, dimenzije i složenost sistema, kao i potreba za intenzivnom interakcijom između članova razvojnog tima, značajno utiču na kvalitet rada tima, a samim tim i na kvalitet finalnog proizvoda. U tim situacijama je prilično teško kontrolisati različite aspekte projekta. Zato se u razvojni tim uključuju i članovi koji pružaju samo usluge podrške. To su, na primer, bibliotekari (bave se dokumentacijom), tim za upravljanje konfiguracijom (obezbeđuje koordinaciju različitih verzija sistema, usklađenost zahteva, projekta, implementacije i testova) i dr.

2 Proces razvoja softvera

U opštem slučaju, pod procesom se podrazumeva uređeni skup zadataka koje treba obaviti da bi se napravio neki proizvod ili pružila neka usluga. U svakom procesu se definišu aktivnosti koje treba izvršiti, kao i resursi koji će tom prilikom biti korišćeni. I aktivnosti i resursi podležu brojnim ograničenjima. Na primer, ograničenja predstavljaju redosled aktivnosti u nekom poslu, raspoloživost alata, budžet, prostor, vremenski rokovi, itd. Svaka aktivnost ima uslove pod kojima otpočinje i pod kojima se završava. Aktivnosti su često međusobno povezane i uslovljene. Proces rezultira proizvodom ili uslugom.

Pošto softver predstavlja jednu vrstu proizvoda, navedene osobine važe i za proces razvoja softvera. Ovaj proces se naziva i životnim ciklusom softvera, zato što opisuje „život“ softverskog proizvoda od početka njegove izrade do njegovog operativnog korišćenja i održavanja.

Proces razvoja softvera je vremenom evoluirao. U početku, sa pojavom prvih računara, softver je bio vrlo jednostavan i programeri su mogli da ga generišu direktnim pisanjem kôda. Nije bilo nikakvog planiranja, već se na osnovu nekoliko odluka formirao sistem.

Kasnije, kada su zahtevi postali brojniji, softver je mogao da razvija samo veći broj programera koji su činili razvojni tim. Da bi proizveli željeni proizvod, članovi razvojnog tima su bili upućeni na međusobnu saradnju. Za uspešnost projekta najvažnije je bilo da ceo razvojni tim ima isti pogled na sistem, tj. da na isti način shvata problem, kao i njegovo rešenje. Zbog povećanja složenosti softvera, dodavanje novih funkcionalnosti je postajalo sve teže, kao i pronalaženje i otklanjanje grešaka. Vremenom se došlo do ideje da bi ovaj problem najlakše mogao da se prevaziđe uvođenjem metodologije razvoja. Pod metodologijom se podrazumeva disciplina u procesu razvoja koja ima za cilj bolju predvidljivost i veću efikasnost razvoja softvera.

Razvoj softvera je podeljen u nekoliko faza:

- *analiza i definisanje zahteva*. U ovoj fazi razvojni tim, u saradnji sa kupcem i korisnicima, utvrđuje zahteve koje sistem treba da zadovolji. Pri analizi zahteva moraju se uzeti u obzir svi entiteti, aktivnosti i ograničenja koji postoje u sistemu. Posebna pažnja se mora posvetiti interakciji sistema sa okruženjem. Rezultat ove faze je lista korisničkih zahteva.
- *projektovanje sistema*. U cilju ispunjenja zahteva definisanih u prvoj fazi, u ovoj fazi se generiše projekat sistema koji daje plan rešenja. Plan uključuje komponente i algoritme koji će biti korišćeni, kao i arhitekturu sistema.
- *projektovanje programa*. U ovoj fazi se definišu podprojekti pogodni za programsku realizaciju. Svaki podprojekat predstavlja jedan modul sa datom funkcionalnošću. Definišu se veze između modula i načini razmene podataka između njih.
- *izrada programa*. Ovo je faza direktne izrade softvera u kojoj programeri pišu programski kôd prema urađenom projektu.
- *testiranje programa*. Ova faza se bavi nalaženjem i ispravljanjem grešaka u sistemu. Nakon pisanja programa, najpre se testiraju individualni delovi kôda, tj. pojedinačni moduli, što se naziva jediničnim testiranjem. Zatim se vrši integraciono testiranje tokom koga se moduli povezuju u jednu celinu. Na kraju sledi završno testiranje u kome se proverava da li sistem ispunjava postavljene zahteve korisnika.
- *isporuka sistema*. U ovoj fazi, sistem se isporučuje naručiocu, softver se instalira u radnom okruženju i obavlja se obuka neposrednih korisnika.
- *održavanje*. Ovo je dugotrajna faza u kojoj se ispravljaju greške u sistemu koje se javljaju nakon njegove isporuke. Takođe se radi i na daljem unapređenju pojedinih delova sistema u skladu sa zahtevima korisnika ili promenama u okruženju.

Teorijski posmatrano, pod procesom razvoja softvera podrazumeva se svaki opis razvoja softvera koji sadrži neke od nabrojanih faza, organizovanih tako da proizvode odgovarajući ispravan i proveren kôd.

Uvođenje planiranja u proces razvoja softvera prema zadatim fazama dovelo je do nastanka tradicionalnih metoda modelovanja. Postoji veliki broj tradicionalnih metoda, a najznačajnije od njih su opisane u poglavlju 2.1.

Brojne teškoće u izvođenju tradicionalnih metoda modelovanja dovele su do pojave novog pristupa procesu razvoja softvera. To je agilni pristup koji negira

potrebu za velikim planiranjem i obimnom dokumentacijom, već se zalaže za fleksibilniji razvoj u kome mnogo toga zavisi od znanja i veština ljudskog faktora. Ovaj pristup je opisan u poglavlju 2.2.

2.1 Tradicionalne metode modelovanja

Modelovanje procesa razvoja softvera obezbeđuje potpunu kontrolu i koordinaciju svih aktivnosti koje treba sprovesti da bi se proizveo željeni softver i ispunili ciljevi projekta. Razloga za modelovanje ima više:

- kada projektni tim opiše sistem, taj opis postaje zajedničko shvatanje svih učesnika u razvoju; to znatno olakšava komunikaciju unutar tima i otklanja mnoge nesporazume i pogrešna tumačenja
- modelovanje odražava ciljeve razvoja; na osnovu modela, projektni tim, pre pisanja softvera, može da oceni predviđene aktivnosti sa aspekta njihove usklađenosti sa postavljenim zahtevima
- modelovanje pomaže u nalaženju nedoslednosti, suvišnih ili izostavljenih elemenata, što poboljšava efikasnost procesa razvoja
- modeli se prave prema konkretnoj situaciji; međutim, mogu se koristiti i u drugim situacijama uz određena prilagođenja; s druge strane, dobro je da neki trag ostane o projektu i nakon njegovog završetka

Da bi se napravio model procesa razvoja softvera, potrebno je definisati skup aktivnosti koje treba da budu izvršene, njihov redosled, ulazne i izlazne podatke za svaku aktivnost pojedinačno, preduslove koji moraju da budu ispunjeni da bi neka aktivnost mogla da se izvrši, kao i posledice izvršavanja pojedinačnih aktivnosti.

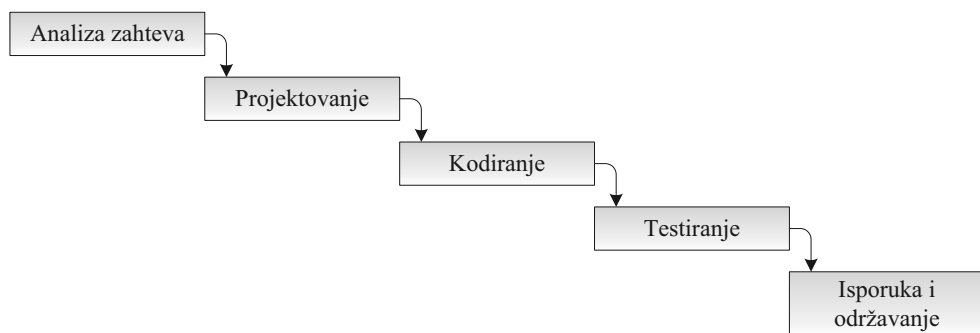
2.1.1 Kaskadni model

Kaskadni model ili model vodopada je verovatno najstariji publikovani model razvoja softvera. Iako je i ranije korišćen u nekim organizacijama, o njemu je među prvima pisao Royce 1970. godine.

Kaskadni model predstavlja veoma visok nivo apstrakcije razvojnog procesa. Naziv modela potiče od načina na koji su u njemu razvojne faze međusobno povezane. Faze su povezane kaskadnom vezom koja se ostvaruje tako što se na narednu fazu prelazi tek nakon završetka prethodne faze. Izlaz iz prethodne faze se prosleđuje narednoj fazi kao ulaz.

Model vodopada sadrži pet faza razvoja označenih i povezanih kao na slici 2.1. U prvoj fazi se radi analiza zahteva kupca. Tek nakon završetka ove faze, može se preći na projektovanje sistema. Slično, po završetku projektovanja, otpočinje kodiranje, tj. pisanje programa. Zatim, po istom principu, slede testiranje, isporuka i održavanje. Svaka faza je praćena obimnom dokumentacijom, pa se za ovaj model često kaže da je vođen dokumentima.

U svakoj fazi, mogu se definisati kritične tačke, koje predstavljaju repere na osnovu kojih se lako može pratiti izvođenje projekta. Kritične tačke mogu da budu, na primer, sastanci u zakazanom terminu na kojima se prezentiraju rezultati, ili informacije da su neki moduli završeni. Osim kritičnih tačaka, u svakoj fazi mogu se definisati i međuproizvodi, na osnovu kojih se dobija uvid u trenutni stepen gotovosti projekta. Za razliku od kritičnih tačaka, koje su više informativnog karaktera, međuproizvodi predstavljaju konkretne celine, na primer, istestiran programski kôd.



Slika 2.1 Kaskadni model

Kaskadni model ima prednosti i nedostatke. Glavne prednosti modela su:

- *jednostavnost*. Visok nivo apstrakcije modela olakšava komunikaciju sa kupcima koji ne moraju da budu upoznati sa procesom razvoja softvera.
- *lako praćenje projekta*. Postojanje kritičnih tačaka i međuproizvoda omogućava rukovodiocu projekta da u svakom trenutku ima informaciju o tome kako projekat napreduje i da li je došlo do nekih bitnih odstupanja u realizaciji na koje treba da reaguje.
- *laka primena modela*. Pošto se ceo sistem razvija u samo jednoj iteraciji, kaskadni model je pogodan za korišćenje u slučajevima kada je potrebno stari sistem u kratkom roku zameniti novim.

Iako je kaskadni model jasan i jednostavan, zbog svojih nedostataka, mali je broj situacija u kojima se može primeniti. Nedostaci kaskadnog modela su sledeći:

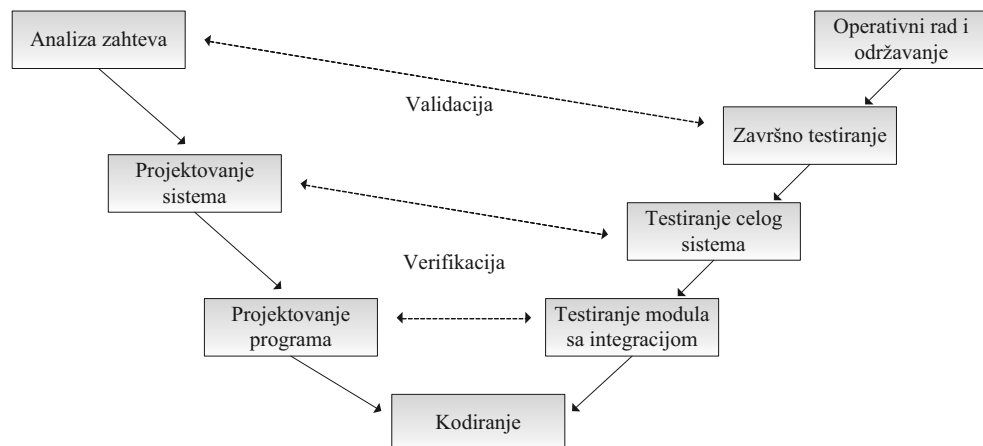
- *ne podržava povratne sprege.* U praksi, svaki složeniji softver se razvija u više iteracija. To je tako zato što je praktično nemoguće u početnoj fazi definisati kompletan skup projektnih zahteva (mnogi faktori razvoja nisu u tom trenutku ni poznati), već se neki od njih, prema potrebi, dodaju kasnije. Osim toga, može se javiti i potreba za izmenama u fazama koje su već završene (na primer, pri kodiranju se ustanovi greška u projektovanju). Ove izmene predstavljaju povratne sprege od kasnijih ka ranijim fazama razvoja. Kaskadni model, po svojoj prirodi, ne ostavlja mogućnost vraćanja na prethodne faze, pa se ne može primeniti u sistemima sa povratnim spregama (a takva je većina sistema).
- *ne ukazuje na način povezivanja faza.* U kaskadnom modelu, svaka faza ima svoj izlaz koji je istovremeno i ulaz naredne faze. Na primer, izlaz prve faze predstavljaju projektni zahtevi na osnovu kojih se zatim, u narednoj fazi, projektuje sistem. Međutim, u modelu nije naznačeno kako zahteve treba transformisati u dizajn. Slično važi i za ostale prelaze između faza (transformacija dizajna u programski kôd).
- *razvoj softvera se ne posmatra kao rešavanje problema.* Model vodopada odražava proizvođački (industrijski, hardverski) pogled na proces razvoja, koji se zasniva na proizvodnji pojedinačnih proizvoda i njihovom repliciranju. Međutim, softver se proizvodi na drugačiji način. On evoluira sa boljim razumevanjem problema kroz ispitivanje različitih varijanti mogućih rešenja. Razvoj softvera je stvaralački, a ne proizvođački proces.
- *ima ograničenu interakciju sa korisnikom.* Kaskadni model dopušta interakciju sa korisnikom samo u početnoj fazi definisanja zahteva i poslednjoj fazi kada se vrši isporuka. To se smatra nedovoljnim. Bilo bi bolje da korisnici češće i pravovremeno mogu da daju svoje sugestije, jer bi to proces razvoja učinilo efikasnijim.

Zbog značajnih nedostataka, kaskadni model je vremenom doživeo mnoga unapređenja. Na osnovu njega, predloženo je nekoliko novih modela procesa razvoja softvera u kojima su otklonjeni neki od navedenih nedostataka.

2.1.2 V model

V model projektovanja softvera je nastao 1992.godine u Nemačkoj, za potrebe nemačkog Ministarstva odbrane. On predstavlja proširen i poboljšan kaskadni model. Naziv modela potiče od činjenice da se proces razvoja softvera predstavlja u vidu dijagrama u obliku slova V, kao što je prikazano na slici 2.2. Leva strana dijagrama odgovara fazama u kaskadnom modelu, na dnu dijagrama se nalazi faza

kodiranja, a u desnom delu dijagrama faze testiranja i održavanja. Kao i u slučaju kaskadnog modela, i kod V modela faze su sekvencijalne, što znači da naredna faza može da počne tek kada se prethodna faza završi.



Slika 2.2 V model

Kao novinu, V model uvodi veze između razvojnih faza, u kojima se precizira šta i kako sistem treba da radi (levi deo dijagrama), i njima odgovarajućih faza testiranja (desni deo dijagrama). Za razliku od kaskadnog modela, koji se uglavnom bavi dokumentima i međuproizvodima, V model svu pažnju usredsređuje na aktivnosti koje se bave ispravnim radom sistema.

Da bi ispravno radio, sistem se testira kroz tri faze: testiranje pojedinačnih modula sa integracijom, testiranje integrisanog sistema i završno testiranje. Prve dve od navedenih faza testiranja služe za verifikovanje dizajna sistema, dok treća faza služi za validaciju sistema. Cilj testiranja pojedinačnih modula uz postepenu integraciju jeste da se razvojni tim uveri da je svaki aspekt sistema ispravno implementiran. Testiranje integrisanog sistema treba da dokaže da sistem kao celina radi ono što se prema projektu od njega očekuje. Kao što se vidi, ove dve faze služe za proveru da li je sistem ispravno i u potpunosti implementiran prema urađenom projektu. Završno testiranje proverava da li sistem ispunjava sve zahteve korisnika navedene u specifikaciji zahteva. Ovo testiranje često sprovode naručioci softvera, jer oni mogu najbolje da procene da li sistem odgovara njihovim potrebama.

Po V modelu, proces razvoja softvera se odvija tako što se najpre sprovodi analiza zahteva, zatim projektuju sistem i programi, a onda se prelazi na kodiranje. Nakon toga slede faze testiranja. Ukoliko se u nekoj fazi testiranja pojavi problem (bilo pri verifikaciji ili pri validaciji), prema zadatim povratnim spregama,

ponavljaju se odgovarajuće aktivnosti iz faza u levom delu dijagrama. Popravke i dopune se mogu odnositi na korisničke zahteve, dizajn sistema ili organizaciju programa. Pošto se unesu potrebne izmene, interveniše se u programskom kôdu, a onda se ponovo sprovodi testiranje. Sistem se može razvijati u više iteracija, ukoliko postoji potreba da se povratne sprege koriste više puta, dok se ne dođe do konačne verzije sistema koja će biti isporučena.

Zbog svoje jednostavnosti i lake primene, V model je jedan od najčešće korišćenih modela za razvoj softvera. Prednosti V modela su sledeće:

- *podržava povratne sprege.* V model je praktično primenljiv ne samo za razvoj jednostavnijih, već i složenijih softverskih sistema, jer dopušta povratak na prethodne faze razvoja, što je čest slučaj.
- *omogućava verifikaciju i validaciju sistema.* V modelom se može proveriti da li je projekat dobro urađen i implementiran, kao i da li sistem ispunjava zahteve korisnika.
- *generiše kvalitetan proizvod.* V model predstavlja strogo kontrolisan proces razvoja, tako da se može garantovati dobar kvalitet softverskog proizvoda.
- *vodi računa o testiranju u ranim fazama projekta.* Tim za testiranje ima uticaja i na ranije faze razvoja, što doprinosi boljem razumevanju na projektu od samog njegovog početka. To kasnije dovodi do značajne uštede u vremenu potrebnom za testiranje.

Nedostaci V modela su:

- *nedovoljna fleksibilnost.* U slučaju pojave nekog problema i povratka na ranije faze, nije dovoljno samo uneti izmene, već se moraju ažurirati i sve naredne faze, uključujući i prateću dokumentaciju.
- *zahteva obimne resurse.* Izvođenje V modela zahteva obimne resurse i veća novčana sredstva (brojniji razvojni tim), tako da ga mogu primenjivati samo veće kompanije.

2.1.3 Fazni razvoj

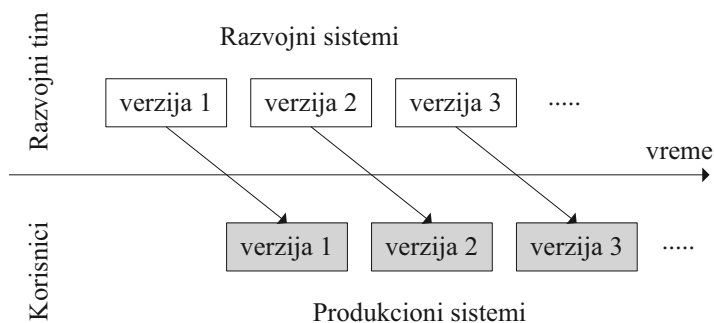
Velika konkurencija na tržištu softvera od proizvođača zahteva skraćanje vremena potrebnog za razvoj softvera. Više se ne može dozvoliti da protekne dug vremenski period od definisanja zahteva do isporuke sistema. Brojne studije pokazuju da proizvođači softvera veliku većinu prihoda ostvaruju od prodaje svojih

novijih softvera (napravljenih u poslednjih par godina). Jedan od načina da se ubrza isporučivanje sistema korisnicima jeste primena faznog razvoja.

Fazni razvoj je način projektovanja softvera koji omogućava isporučivanje sistema u delovima, tj. fazama. Svaka faza obuhvata određen skup funkcija definisan projektom. Kompletan funkcionalnost sistema se dobija objedinjavanjem funkcija iz svih faza. Ovakav način razvoja podrazumeva da je u datom trenutku korisnicima na raspolaganju jedan skup funkcija, dok su ostale funkcije još u razvoju.

Prema tome, postoje paralelno dva sistema: produkcion i razvojni. *Produkcion sistem* je sistem koji trenutno koriste naručioci ili korisnici. *Razvojni sistem* je sistem na kome radi razvojni tim. Razvojni sistem je, u stvari, naredna verzija sistema koja se priprema da zameni postojeći produkcion sistem.

Produkcion i razvojni sistemi se obično označavaju svojim verzijama (izdanjima). Broj mogućih verzija odgovara broju faza koje se isporučuju pri faznom razvoju. Ako korisnici trenutno rade na n -toj verziji produkcionog sistema, onda razvojni tim radi na $(n+1)$ -oj verziji razvojnog sistema. Nakon isporuke ove faze, korisnici napuštaju n -tu verziju produkcionog sistema i prelaze na novu $(n+1)$ -u verziju. Razvojni tim počinje da radi na novom skupu funkcija koji predstavlja novu, $(n+2)$ -u verziju razvojnog sistema. Sličan postupak se ponavlja sve dok se ne isporuči i poslednja faza, tj. dok korisnik ne dobije kompletan softverski proizvod. Na slici 2.3 dat je opšti model faznog razvoja.



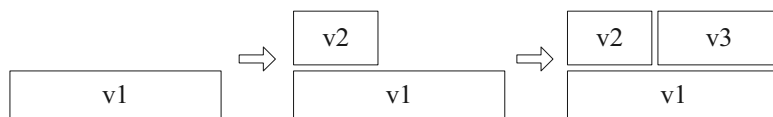
Slika 2.3 Model faznog razvoja

Postoji više pristupa organizaciji faznog razvoja. Oni se razlikuju po načinu formiranja verzija koje se isporučuju korisniku. Dva najpopularnija pristupa su: inkrementalni i iterativni razvoj.

2.1.3.1 Inkrementalni fazni razvoj

Inkrementalni fazni razvoj podrazumeva podelu sistema na podsisteme prema funkcijama definisanim u specifikaciji zahteva. Verzije se definišu kao mali funkcionalni podsistemi koji, svaki za sebe, sadrže različite skupove funkcija. Svaka isporuka nove verzije znači dalju nadogradnju sistema do njegove pune funkcionalnosti.

Inkrementalni razvoj će biti ilustrovan na jednom primeru. Neka je potrebno realizovati softver koji podržava tri vrste funkcionalnosti: unos podataka, proračune i prikaz rezultata. Sistem se može razvijati kroz tri verzije (faze), v1, v2 i v3, kao što je prikazano na slici 2.4.



Slika 2.4 Primer inkrementalnog razvoja

Nakon isporuke prve faze, korisnicima su na raspolaganju samo funkcije za unos podataka iz v1, a nakon isporuke druge faze, funkcije za unos podataka i funkcije za proračune (iz v1 i v2). Tek po isporuci treće faze, korisnici mogu da koriste sve funkcije sistema (iz v1, v2 i v3).

Prednosti koje korisnicima pruža inkrementalni fazni razvoj su sledeće:

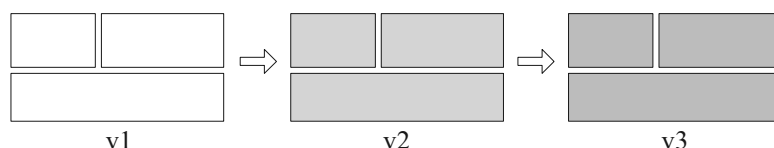
- *brza isporuka operativnog skupa funkcija.* Već nakon isporuke prve faze, korisnici imaju na raspolaganju potpuno realizovan podskup funkcija koje će biti sastavni deo konačnog proizvoda. Svaka naredna faza uvećava broj funkcija koje imaju svoj finalni oblik.
- *vidljiv napredak na projektu.* Zbog isporuke dela funkcionalnosti nakon svake faze, progres na projektu se može vrlo lako i jednostavno pratiti. On je vidljiv ne samo preko dokumenata, već i u praktičnom radu.
- *permanentni odziv korisnika.* Interakcija sa korisnikom koja je prisutna tokom celog ciklusa razvoja vodi ka stabilnijim međurezultatima i sistemu uopšte. Pošto se korisnik vrlo rano sreće sa prvim operativnim funkcijama, može na vreme da interveniše ukoliko u njima uoči neke nedostatke. Ako se ti nedostaci otklone u ranijim fazama projekta, troškovi su manji, a razvoj efikasniji.

- *mali projektni tim.* Zbog malih verzija, razvojni tim može da ima relativno mali broj članova. To smanjuje ukupne troškove na projektu, ali može da utiče na kvalitet proizvoda. Naime, pošto isti razvojni tim radi na svim verzijama, a poslovi se mogu bitno razlikovati od verzije do verzije, teško je obezbediti da svako u timu u dovoljnoj meri poznaje sve potrebne tehnologije. Na primer, za programera je teško ako mesec dana treba da razvija korisnički interfejs u datom okruženju, zatim da programira u C-u, da bi u sledećoj fazi radio sa bazama podataka. Ovakva dinamika rada svakako mora da utiče na dobijeni rezultat. Da bi se ipak postigao potreban kvalitet, razvojnom timu je potrebno više vremena za rad, što opet anulira početne uštede u troškovima zbog malog broja članova u razvojnom timu.

2.1.3.2 Iterativni fazni razvoj

Iterativni fazni razvoj, slično inkrementalnom, podrazumeva podelu sistema na podsisteme prema funkcijama. Međutim, sada se u svim verzijama isporučuje potpuni sistem, s tim što se u svakoj novoj verziji menjaju funkcije svakog od podsistema. To znači da svaka nova verzija unapređuje prethodnu, dok se ne dobije kompletan sistem.

Na slici 2.5 prikazan je iterativni postupak razvoja softvera iz primera opisanog u okviru inkrementalnog razvoja.



Slika 2.5 Primer iterativnog razvoja

Kao što se vidi, u svim fazama se isporučuje ceo sistem. Međutim, u verziji v1 obezbeđeni su primitivni oblici sve tri vrste funkcionalnosti sistema. Na primer, podaci se mogu unositi ručno, proračuni su nekompletni, i može se generisati uprošćena varijanta izveštaja. Nakon isporuke druge faze (v2), sistem ima istu funkcionalnost, ali je kvalitet poboljšan. Na primer, jedan skup ulaznih podataka se automatski preuzima iz drugog programa, proračuni su efikasniji, a izveštaj detaljniji. Iz ovoga se vidi da je sistem unapređen. Potpuna funkcionalnost se dobija tek nakon isporuke treće faze v3.

Prednosti iterativnog faznog razvoja su sledeće:

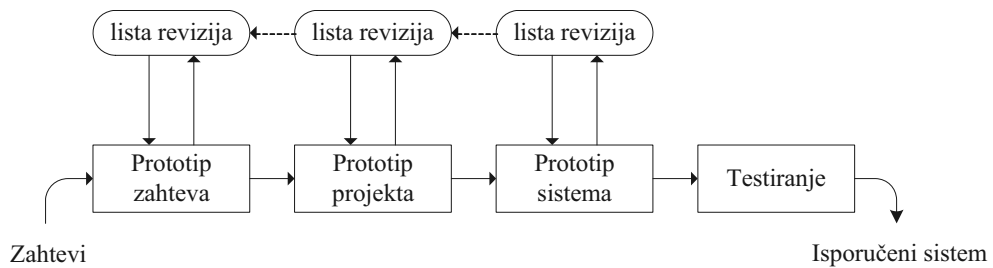
- *moгуćnost rane obuke*. Pošto se već u prvoj verziji isporučuje ceo sistem, korisnici mogu odmah da počnu sa delimičnom obukom. Ona se ogleda u upoznavanju sa organizacijom korisničkog interfejsa, kao i sa načinima izvršavanja pojedinih funkcija. Iako funkcije nisu kompletno realizovane, korisnik saznaje šta može da očekuje u konačnoj verziji. Ovako rana obuka je dobra, jer korisnici mogu razvojnom timu na vreme da sugerišu moguća poboljšanja u kasnijim verzijama.
- *česte isporuke*. Ukoliko je moguća, isporuka novih verzija u kratkim vremenskim intervalima uvek daje dobre rezultate. Problemi se brzo i lako otklanjaju zahvaljujući informacijama dobijenim od korisnika. Međutim, česte isporuke zahtevaju veliku odgovornost po pitanju kvaliteta isporučene verzije.
- *moгуćnost specijalizovanih verzija*. U različitim verzijama, razvojni tim može da se posveti usavršavanju različitih aspekata sistema. Na primer, u jednoj verziji može da unapredi korisnički interfejs, u drugoj da unapredi performanse sistema, itd.

2.1.4 Prototipski model

Prototipski model razvoja softvera se zasniva na izradi prototipova softverske aplikacije. Pod prototipom se podrazumeva nekompletna verzija programa koji se razvija. Zahvaljujući prototipovima, u kratkom vremenskom periodu se mogu generisati kompletan sistem ili njegovi delovi u cilju pojašnjavanja ili boljeg razumevanja otvorenih pitanja. Postoje prototipovi različitih vrsta, što zavisi od toga šta se njima želi postići, tj. sa kojim ciljem se izrađuju. Bez obzira na vrstu, svrha svih prototipova je da se smanje rizik i neodređenost prilikom razvoja sistema.

Protipovi mogu da budu uključeni u finalni proizvod, ali i ne moraju. Ukoliko se ne uključuju, njihova uloga je da se brzo i efikasno ispitaju različite mogućnosti u fazama razvoja.

Na slici 2.6 prikazan je opšti oblik prototipskog modela. Razvoj po prototipskom modelu otpočinje definisanjem skupa zahteva koji odgovaraju potrebama korisnika ili naručioca. Zatim se analiziraju predlozi različitih varijanti izgleda ekrana korisničkog interfejsa, načina unosa podataka, tabela sa podacima, izlaznih izveštaja i svega onoga što je direktno na raspolaganju korisnicima. Korisnici izlažu svoje želje i revidiraju postojeće zahteve. Posle više iteracija, kada se postigne konačni dogovor o zahtevima, kao rezultat se dobija prototip zahteva.



Slika 2.6 Prototipski model

Zatim projektni tim prelazi na izradu prototipa projekta. Istražuju se različite varijante dizajna, često uz konsultovanje sa korisnicima. Početni dizajn se revidira sve dok svi učesnici u projektovanju ne budu zadovoljni prototipom projekta. Ako se pri projektovanju nađe na probleme koji potiču od zahteva, povratnom spregom se ostvaruje vraćanje na specificirane zahteve. Oni se opet analiziraju, menjaju i dobija se novi, izmenjen prototip zahteva.

Nakon generisanja prototipova zahteva i projekta, prelazi se na kodiranje, tj. izadu prototipa sistema. Opet se razmatraju različite varijante kodiranja, uz mogućnost povratka na analizu projekta, ili čak analizu zahteva.

Na kraju se sistem testira i isporučuje korisnicima.

Prednosti prototipskog razvoja su sledeće:

- *redukovanje vremena i troškova.* Prototipski model poboljšava kvalitet zahteva zbog detaljnih analiza koje se sprovede pri izradi prototipa zahteva. Rano utvrđivanje šta korisnik zaista želi bitno ubrzava razvoj i smanjuje troškove na projektu koji sa vremenom eksponencijalno rastu.
- *intenzivna interakcija sa korisnicima.* Uključivanje korisnika u izradu prototipova značajno smanjuje greške koje nastaju zbog različitih tumačenja zainteresovanih strana u projektu. Pošto korisnici ipak najbolje poznaju domen problema, intenzivna interakcija povećava kvalitet finalnog proizvoda.

Prototipski razvoj ima sledeće nedostatke:

- *nedovoljna analiza.* Fokusiranje na prototipove koji predstavljaju pojednostavljene komponente sa ograničenom funkcionalnošću može da odvraća razvojni tim od detaljne analize na nivou celog projekta. Zbog toga se dešava da dođe do previđanja boljih rešenja, izrade nekompletne specifikacije ili konverzije prototipova u konačna rešenja koja su teška za održavanje.

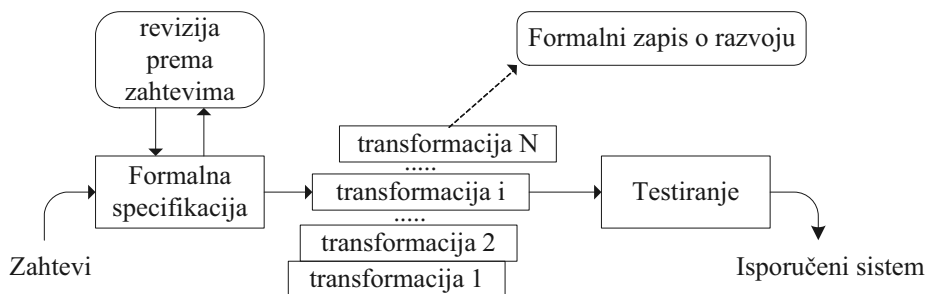
- *konfuzija između prototipova i finalnih sistema.* Korisnici mogu da dobiju pogrešan utisak da su prototipovi (koji su inače namenjeni analizi, a zatim se uglavnom odbacuju) u stvari finalni proizvodi koje samo treba malo doraditi. Oni, na primer, nisu svesni koliki napor treba uložiti da bi se dodala provera grešaka ili mehanizmi zaštite koje prototip ne podržava. To dovodi do toga da korisnici očekuju od prototipova da precizno modeluju performanse finalnog sistema, što nije namera razvojnog tima. Takođe, korisnici ponekad prihvataju svojstva uključena u prototip kao finalna, iako će ona kasnije biti isključena iz sistema, pa može da dođe do konflikta sa razvojnim timom.

2.1.5 Transformacioni model

Transformacioni model (Balzer, 1985.godine) predstavlja pokušaj automatskog modelovanja procesa razvoja softvera. Ideja je da se smanji mogućnost greške u modelovanju uvođenjem niza transformacija kojima se polazna specifikacija prevodi u sistem koji može da se isporuči korisniku. Pošto su transformacije unapred definisane i raspoložive, modelovanje se svodi na izbor sekvence transformacija iz zadatog skupa kojom se može ostvariti postavljeni cilj projekta. Tipične transformacije su: prelazak sa jednog na drugi način predstavljanja podataka, različiti algoritmi, metodi optimizacije, prevođenje i sl.

Do istog cilja se može doći na mnoge načine, tj. mogu se koristiti različite sekvence transformacija. Zato je važno da se pri korišćenju transformacionog modela sekvence transformacija i prateće odluke čuvaju u vidu formalnih zapisa u projektu sistema.

Na slici 2.7 prikazan je transformacioni model.



Slika 2.7 Transformacioni model

Kao što se vidi, najpre se zahtevi sistema prevode u formalnu specifikaciju. To je neophodno zato što transformacije mogu da se izvršavaju samo nad formalnom specifikacijom (jer su i same formalne). Nakon toga se određuje sekvenca transformacija koja polaznu specifikaciju prevodi u sistem, koji se zatim testira i isporučuje. Ceo proces modelovanja je praćen formalnim zapisima.

Transformacioni model je veoma mnogo obećavao, mada se već može postaviti pitanje da li će ispuniti ta očekivanja, s obzirom da je proteklo mnogo vremena od kada je predložen. Najveći problem da ovaj model zaživi i bude šire prihvaćen je u neophodnosti formalne specifikacije koju nije lako napraviti. U poslednje vreme, dosta se radi u oblasti formalnih specifikacijskih modela, što bi moglo da dovede do intenzivnije upotrebe transformacionog modela.

2.1.6 Spiralni model

Spiralni model je formulisao Boehm 1986.godine. Ovaj model zahteva da se u procesu razvoja softvera vodi računa o postojećim rizicima. To se postiže tako što se uobičajene aktivnosti razvoja softvera kombinuju sa analizom rizika. Cilj je da se omogućí upravljanje rizicima, kako bi se smanjio njihov broj i olakšala njihova kontrola.

Spiralni model je nastao kao rezultat napora da se objedine dobre osobine razvoja odozgo-nadole (projektovanje sistema njegovom dekompozicijom na podsisteme sa postepenim prelaskom na detalje) i odozdo-nagore (prototipsko projektovanje manjih celina koje se posle nadograđuju i povezuju u kompletan sistem). Može se reći da spiralni model kombinuje kaskadni sa prototipskim modelom. Spiralni model je namenjen razvoju velikih, složenih i skupih sistema.

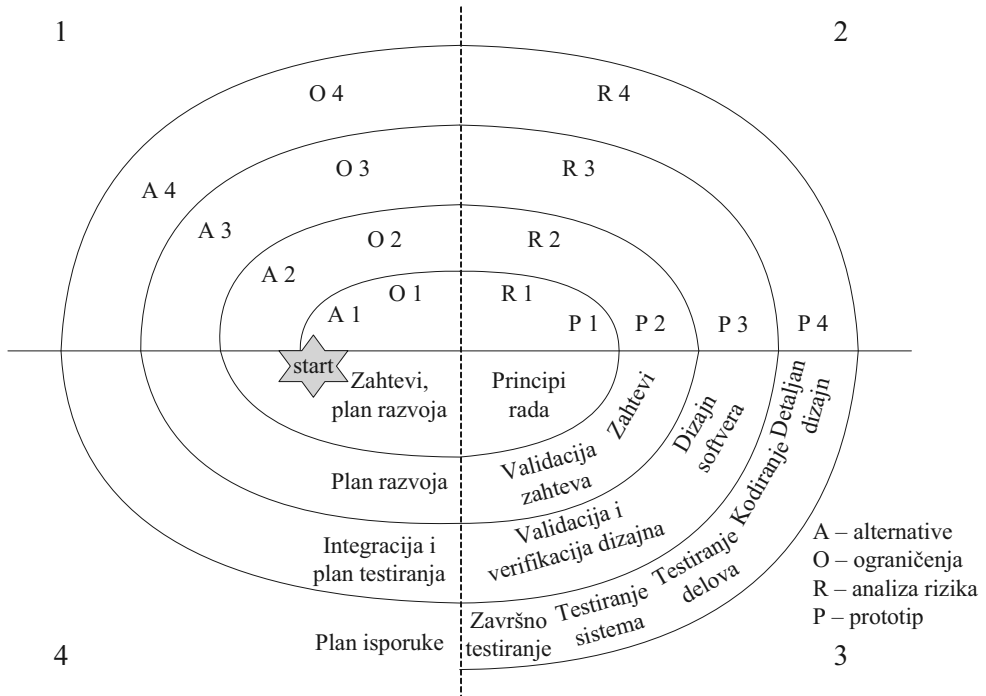
Na slici 2.8 prikazana je spirala razvoja koja se izvodi u više iteracija, što znači da model predstavlja iterativni razvoj.

Kao što se vidi, razvoj softvera se odvija u četiri iteracije. Prva iteracija se odnosi na zahteve i planiranje životnog ciklusa, druga na planiranje razvoja, treća na plan integracije i testiranja, a četvrta na implementaciju.

Svaka iteracija obuhvata jedan pun krug na dijagramu i prolazi kroz četiri kvadranta koji odgovaraju sledećim aktivnostima:

- | | |
|-------------|---|
| kvadrant 1: | određivanje ciljeva, alternativa i ograničenja |
| kvadrant 2: | evaluacija alternativa, identifikacija i procena rizika |
| kvadrant 3: | razvoj i verifikacija putem testiranja |
| kvadrant 4: | planiranje sledeće iteracije |

Aktivnosti koje treba obaviti na početku svake iteracije (kvadrant 1) su: identifikacija ciljeva iteracije, utvrđivanje različitih alternativa za postizanje ovih ciljeva i utvrđivanje ograničenja koja se u pojedinim alternativama nameću.



Slika 2.8 Spiralni model

Nakon ovih, slede aktivnosti (kvadrant 2) koje se odnose na evaluaciju alternativa i ograničenja uz uključivanje neizvesnosti i rizika. Za svaku alternativu se prave prototipovi ili simulacije da bi se smanjio rizik u donošenju odluka.

Nakon toga, pristupa se razvoju softvera (kvadrant 3) prema kaskadnom modelu imajući u vidu rezultate analize rizika.

Na kraju (kvadrant 4), pristupa se planiranju naredne iteracije.

Prednosti spiralnog modela su:

- *redukovanje rizika.* Po spiralnom modelu, sistem se razvija tako što se najpre izdvoje karakteristike sa najvišim prioritetom, a zatim se na osnovu njih razvija prototip. Nakon testiranja prototipa, željene izmene se unose u novi sistem. Ovakav način razvoja minimizira rizike i greške pri izradi sistema.

- *dobra kontrola troškova.* Pošto su prototipovi mali fragmenti u okviru projekta, troškovi se mogu lako proceniti. Zato kupac može da ima dobru kontrolu administriranja novog sistema.
- *aktivno učešće korisnika.* Polazeći od prve do finalne faze u modelu, znanje korisnika o sistemu stalno raste. Interakcija sa korisnikom omogućava ravnomeran razvoj softverskog proizvoda koji zadovoljava potrebe korisnika.

Spiralni model ima sledeće nedostatke:

- *ograničena primena.* Spiralni model najbolje funkcioniše u slučaju velikih i složenih projekata, dok za manje projekte nije pogodan.
- *neophodno znanje o rizicima.* Model zahteva veliku veštinu u evaluaciji neizvesnosti i rizika. Osim toga, uvođenje rizika dovodi do povećanja troškova, pa se dešava da troškovi evaluacije rizika budu veći od troškova izrade sistema.
- *složenost modela.* Spiralni model ima striktno definisan protokol razvoja, koga je ponekad teško ispoštovati.

2.1.7 RUP

RUP (*Rational Unified Process*) je metodologija razvoja softvera kreirana u kompaniji *Rational Software* 1996.godine (kompanija se od 2003.godine nalazi u sastavu IBM-a). Po prirodi, to je adaptivni proces opštije namene, što znači da svaka razvojna organizacija može da selektuje elemente RUP-a i tako formira proces razvoja koji joj najviše odgovara. RUP je iterativni postupak razvoja softvera orijentisan ka arhitekturi i vođen slučajevima korišćenja. Proces je dobro strukturiran i jasno definiše ko, šta i kako treba da uradi na projektu. Opšteg je karaktera i može se prilagoditi kako malim, tako i velikim projektima i razvojnim timovima. Prilagođavanja se izводе izborom elemenata koje nudi RUP i njihovim organizovanjem u razvojni proces koji zadovoljava konkretne potrebe.

Osnovni gradivni elementi RUP-a su:

- *uloge (ko),* koje definišu skup povezanih veština, sposobnosti i odgovornosti
- *proizvodi rada (šta),* koji predstavljaju rezultat nekog zadatka, uključujući modele, proizvode, dokumentaciju i sl.

- *zadaci* (kako), koji opisuju posao dodeljen ulozi koji proizvodi neki koristan rezultat

U svakoj iteraciji, zadaci su organizovani u devet disciplina, šest inženjerskih (poslovno modelovanje, zahtevi, analiza i projektovanje, implementacija, testiranje i isporuka) i tri discipline za podršku (konfiguracija i upravljanje izmenama, upravljanje projektom i okruženje).

RUP posmatra životni ciklus projekta kroz sledeće četiri faze:

- faza započinjanja (*Inception Phase*)
- faza razrade (*Elaboration Phase*)
- faza konstrukcije (*Construction Phase*)
- faza tranzicije (*Transition Phase*)

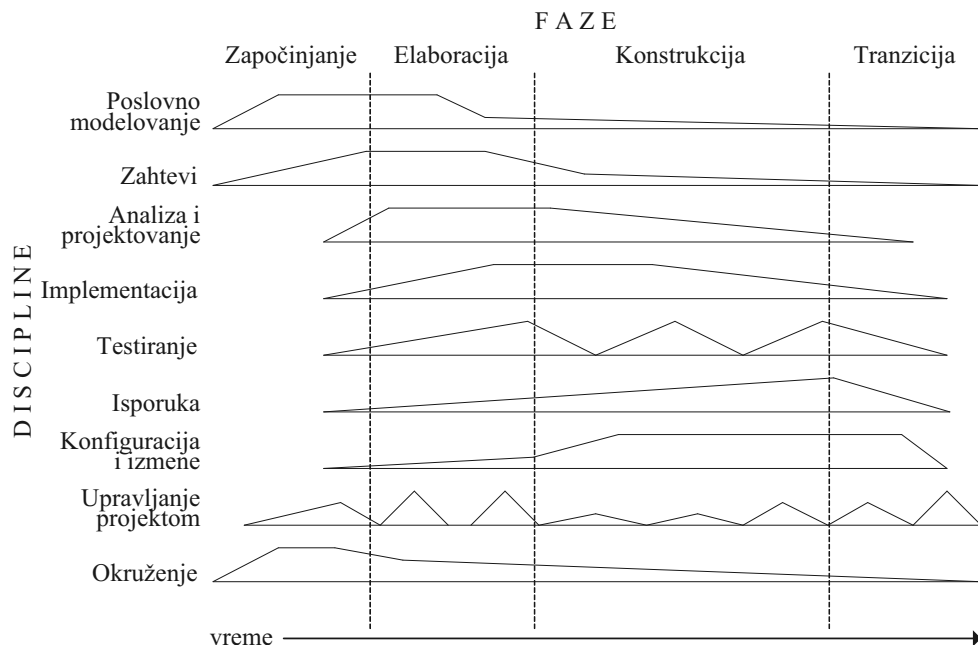
Navedene faze omogućavaju predstavljanje procesa razvoja na visokom nivou apstrakcije (slično kaskadnom modelu), iako je suština razvoja, u stvari, u iteracijama koje se odvijaju u svim fazama. Organizacija RUP-a po fazama prikazana je na slici 2.9.

Svaka faza ima jasno definisane ciljeve i kritičnu tačku (*milestone*) na kraju faze. Kritična tačka podrazumeva davanje odgovora na pitanja o ispunjenosti ciljeva faze.

Svrha *faze započinjanja* je razumevanje obima i ciljeva projekta. To znači da treba precizno utvrditi šta treba da se napravi, koliki je obim sistema, kakva je vizija, ko želi sistem i šta je za njega važno. Nakon toga se identifikuju osnovne funkcionalnosti sistema i definišu slučajevi korišćenja. U ovoj fazi se predlaže bar jedno rešenje. Takođe, razmatraju se i troškovi, kao i potencijalni rizici na projektu. Ukoliko je projekat složen, faza započinjanja se odvija u više iteracija, jer je razvojnom timu potrebno više vremena da ga dobro razume. Na kraju ove faze je prva kritična tačka u kojoj treba odlučiti da li je projekat izvodljiv i finansijski prihvatljiv. Ukoliko jeste, prelazi se na narednu fazu. U suprotnom, odustaje se od projekta.

Faza razrade se bavi definisanjem osnovne arhitekture sistema na osnovu slučajeva korišćenja, što je preduslov za dobro projektovanje i implementaciju. Cilj ove faze je smanjenje rizika u pogledu definisanih zahteva, predložene arhitekture i izbora alata. I ova faza se može odvijati u više iteracija u zavisnosti od veličine projekta. Na kraju faze je druga kritična tačka koja se odnosi na pitanja u vezi sa postojanjem detaljnog plana vođenja projekta i preduzetim postupcima za

minimizaciju rizika. Ako je sve u redu, prelazi se na sledeću fazu. U suprotnom, ili se odustaje od projekta, ili se izvode suštinske izmene.



Slika 2.9 RUP faze i discipline

U *fazi konstrukcije* razvijaju se potrebne komponente, obavljaju se testiranja i kompletira dokumentacija projekta. Ciljevi faze su: minimizacija troškova razvoja, obezbeđivanje odgovarajućeg kvaliteta softverskog proizvoda i priprema za isporučivanje proizvoda korisniku. Faza se može odvijati u više iteracija. Na kraju se dolazi do treće kritične tačke koja pruža odgovor na pitanje da li je finalna verzija spremna za isporuku. Ako jeste, prelazi se na poslednju fazu, a ukoliko nije, treba odlučiti da li dalji razvoj ima svrhe, ili se treba vratiti na neku od prethodnih faza.

Faza tranzicije podrazumeva isporučivanje gotovog proizvoda. Tokom ove faze preduzimaju se sve aktivnosti kako bi korisnik bio zadovoljan proizvodom, jer to i jeste osnovni cilj faze. I ova faza se odvija u više iteracija, pri čemu se u svakoj od njih prikupljaju sugestije korisnika koje će biti uključene u narednu verziju projekta. Na kraju faze, dolazi se do poslednje kritične tačke u kojoj treba odgovoriti na pitanje da li je sistem kompletan, stabilan i u potpunosti spreman za isporuku. Ako jeste, sledi donošenje odluke o tome da li treba otpočeti novi ciklus razvoja. U suprotnom, treba se vratiti na neku od prethodnih faza.

Prednosti RUP-a su sledeće:

- *visok nivo prilagodljivosti* konkretnom projektu, jer RUP ništa ne zahteva, već samo preporučuje
- *iterativnost procesa* koja omogućava postepen razvoj sistema, što vodi smanjenju troškova i vremenskim uštedama
- *upravljanje rizicima* što usmerava razvoj ka nižim troškovima

Glavni nedostaci RUP-a su:

- *neprilagođenost malim projektima* kod kojih faze započinjanja i razrade gube na značaju
- *iterativnost procesa*, što može da predstavlja i manu, ukoliko su rukovodioci i razvojni timovi neiskusni, jer tada često dolazi do velikih propusta čije su posledice prekoračenja rokova, ili čak odustajanje od projekta

Zbog svoje fleksibilnosti, RUP je veoma korišćena metodologija za razvoj softvera od strane iskusnijih razvojnih timova.

2.2 Agilne metode

Počeci agilnih procesa razvoja datiraju iz osamdesetih godina prošlog veka. Međutim, zvanični nastanak agilnih metoda se vezuje za 2001. godinu i formiranje Agilne alijanse, neprofitne organizacije sa globalnim članstvom, koja se zalaže za primenu agilnih principa u procesu razvoja. Cilj alijanse je da softverska industrija postane produktivnija, humanija i da bude održiva.

Agilne metode su nastale kao otpor mnogim ranijim modelima procesa razvoja koji su pokušavali da nametnu neki oblik discipline u osmišljavanju softvera, projektovanju, implementaciji, testiranju i dokumentovanju. Osnovna agilna ideja je da se naglasi uloga fleksibilnosti u spretnom i brzom razvoju softvera.

Nove, agilne principe razvoja za koje se zalaže, Agilna alijansa je objavila u Manifestu agilnog razvoja softvera (*Manifesto for Agile Software Development*, februar 2001.). Svaki princip formulisani je tako da jasno pokazuje šta se u agilnom razvoju više vrednuje (označeno kurzivom):

- *pojedinci i interakcije* od procesa i alata. U agilnom razvoju, samoorganizovanje i motivacija članova razvojnog tima su izuzetno važni, kao i njihova interakcija. Oni imaju primat u odnosu na procese i alate koji se koriste u razvoju. Filozofija agilnog razvoja u prvi plan stavlja kvalitet pojedinaca i kvalitet njihove saradnje. Smatra se da je dovoljno razvojnom timu obezbediti potrebne resurse, a onda treba imati poverenje u njega da će dobro odraditi svoj posao. Komunikacija u okviru razvojnog tima se ostvaruje direktno, licem u lice, a ne posredstvom dokumentacije. U agilnom razvoju je poželjno da razvojni tim bude iskusan i usklađen tokom rada na ranijim projektima, jer je tada razvoj softvera najefikasniji.
- *primenljiv softver* od detaljne dokumentacije. Po ovom agilnom principu, bolje je utrošiti vreme na izradu softvera koji radi, nego na pisanje detaljne dokumentacije. Na primer, na sastancima sa kupcem, prioritet se uvek daje prezentaciji softvera u odnosu na opise i izveštaje u papirnoj formi. Merilo uspešnosti projekta je do koje mere softver realizuje potrebnu funkcionalnost.
- *saradnja sa kupcem* od ugovornih aranžmana. Pošto zahtevi obično ne mogu u potpunosti da budu prikupljeni na početku razvoja softvera, stalna saradnja sa kupcem je od velikog značaja. Zajedničkim radom, kupac se uključuje u glavne aspekte razvoja.
- *reakcija na promene* od pridržavanja plana. Agilni razvoj se fokusira na brzo odgovaranje na sve promene uz dalje nastavljjanje procesa razvoja. Smatra se da, u slučaju potrebe za nekom izmenom, ne treba trošiti vreme na replaniranje i praćenje plana, već se treba odmah usredsrediti na izvođenje izmene.

Postoji veliki broj različitih agilnih metoda koje poštuju principe navedene u Manifestu. Neke od njih su:

- *XP – Extreme Programming*. Obuhvata skup tehnika u kojima se naglašava kreativnost timskog rada uz minimizaciju prekomernog administriranja.
- *Scrum*. Propisuje načine upravljanja zahtevima, iteracijama razvoja, implementacijom i isporukom.
- *Crystal*. Predstavlja familiju metodologija koje se fokusiraju na razvojni tim, a ne na procese i međuproizvode. Smatra se da svaki projekat zahteva različite dogovore i metodologije i da najveći uticaj na kvalitet softvera ima razvojni tim. Produktivnost na projektu se povećava dobrom komunikacijom i čestim isporukama, čime se smanjuje potreba za međuproizvodima.

- *ASD – Adaptive Software Development*. Zasniva se na šest principa, od kojih prvi predstavlja misiju koja postavlja cilj razvoja, ali ne propisuje način na koji će cilj biti ostvaren. Pošto su za naručioca najvažnija svojstva softvera, projekat se organizuje tako da realizuje komponente koje implementiraju svojstva. Razvoj se odvija u više iteracija, pri čemu sve imaju podjednaku važnost. Izmene koje treba uraditi ne posmatraju se kao greške, već kao prilagođavanje sistema realnim uslovima razvoja. Fiksno vreme isporuke nalaže razvojnom timu da redukuje zahteve u svakoj verziji sistema. Rizik se prihvata, tako da razvojni tim najpre rešava najteže probleme.

2.2.1 Ekstremno programiranje

Ekstremno programiranje (XP) je agilni metod koji je predložio Kent Beck 1996.godine. Uglavnom ga koriste manji i srednji razvojni timovi koji imaju od 6 do 20 članova. Tim koji razvija softver na ovaj način ima dobru saradnju sa klijentima i sve napore usmerava ka kratkim iteracijama koje korisnicima daju direktan i koristan rezultat. Ovakav pristup ide na uštrb planiranja i izradi obimne dokumentacije na projektu.

XP ne propisuje strogu metodologiju koje se razvojni tim mora pridržavati, već samo daje uzore koji timu ostavljaju mogućnost izmene procedura ili redosleda izvođenja pojedinih aktivnosti. Ovaj način razvoja softvera uvodi četiri elementa koji se međusobno dopunjuju: osnovne vrednosti, principe, aktivnosti i prakse. Centralni deo predstavljaju osnovne vrednosti koje se postižu prihvatanjem principa. Principi se sprovode pomoću skupa mehanizama koji se koriste u praksi. Sve navedeno se ogleda u aktivnostima rada razvojnog tima.

U ekstremnom programiranju je od velikog značaja da razvojni tim bude upoznat sa *osnovnim vrednostima* razvoja i da ih prihvati. Ove vrednosti poštuju agilne vrednosti i dalje ih nadograđuju. U osnovne vrednosti XP-a spadaju:

- *komunikacija*. Izrada softvera zahteva intenzivnu komunikaciju između članova razvojnog tima. Za razliku od formalnih metoda, kod kojih se komunikacija uglavnom zasniva na razmeni dokumenata, kod XP-a je cilj da se, što je moguće brže, znanje raširi među članovima tima, kako bi svi imali isti pogled na sistem (pogled sličan korisnikovom). Uspeh projekta je moguć samo ako se znanje nesmetano prenosi unutar tima.
- *jednostavnost*. U XP-u se uvek polazi od jednostavnih rešenja koja se kasnije mogu nadograditi, ukoliko je to potrebno. Dakle, ne troši se unapred vreme na izradu opštijih i složenijih rešenja, koja možda nikad

neće u potpunosti biti iskorišćena. To odgovara YAGNI principu po kome ne treba praviti nešto što trenutno nije potrebno.

- *povratna sprega*. Veoma je važno da se u svakom trenutku zna u kakvom je stanju sistem koji se razvija. Ova informacija se može dobiti jedino zahvaljujući raznim povratnim spregama: programer-sistem, programer-programer, korisnik-sistem, korisnik-programer. Povratne sprege su više praktične nego verbalne prirode, jer neposredno ukazuju na konkretne probleme u razvoju.
- *hrabrost*. Hrabrost podrazumeva stalno prihvatanje rizika i promena. U direktnoj je vezi sa samopouzdanjem članova tima i poverenjem u razvojni tim. Hrabrost pomaže timu da se u nekim situacijama opredeli za odbacivanje dela posla koji je već uradio i otpočne nešto iznova. To je i posvećenost blagovremenim i čestim isporukama funkcija, što nije jednostavno. Pri svakoj isporuci mora se dobro razmisliti i proveriti šta se isporučuje.
- *poštovanje*. Svaki član razvojnog tima mora da poštuje sebe i druge. U svom radu mora da se trudi da postigne visok kvalitet ne ugrožavajući rad drugih. Na primer, programer ne može da unosi izmene koje postojeće testove čine pogrešnim. Ili, ne može svojim poslom da prouzrokuje zakašnjenje drugih. Niko u timu ne sme da se oseća zapostavljenim ili manje vrednim.

Navedene vrednosti su međusobno povezane i imaju veliki uticaj jedne na druge. One predstavljaju osnovne kriterijume po kojima se može sagledati uspešnost posla. Ipak, suviše su opšte da bi se na osnovu njih mogli definisati praktični mehanizmi koji bi obezbedili uspešnost. Zato su uvedeni principi koji otkrivaju različite alternative koje služe za odlučivanje, a uključuju osnovne vrednosti.

Principi XP-a proističu iz Manifesta agilne metodologije, uz dodatna proširenja. Oni predstavljaju osnovu XP-a. Konkretniji su od osnovnih vrednosti i mogu se lakše prevesti u smernice razvoja u praktičnim situacijama. Principi XP-a su:

- *povratna sprega*. U XP-u je važno da vreme potrebno za dobijanje povratnih informacija bude kratko, jer je ono kritično za sticanje novih znanja i obavljanje izmena. Zato su kontakti sa korisnicima znatno češći nego kod tradicionalnih metoda.
- *pretpostavljena jednostavnost*. XP odbacuje planiranje i kodiranje u svrhu ponovne upotrebe kôda, koji su karakteristični za tradicionalne metode. U

XP-u, rešenje svakog problema se posmatra kao „ekstremno“ jednostavno. Vreme koje se na ovaj način uštedi, znatno prevazilazi vreme koje se izgubi u retkim slučajevima kada to nije tačno. Teži se malim izmenama i čestim isporukama, zato što korisnik tako ima bolji uvid u razvoj projekta, što je s druge strane korisno i za razvojni tim.

- *prihvatanje promena*. Svaka izmena se prihvata ma koliko bila velika. Strategija menjanja nalaže da se najpre izvode izmene koje rešavaju najvažnije probleme, a zatim se prelazi na manje važne izmene.
- *kvalitet rada*. Uspeh projekta je jedino moguć ako se svi članovi razvojnog tima maksimalno zalažu, tj. rade svoj posao najbolje što mogu. U takvoj radnoj atmosferi, svi dobijaju nove motive, stimulišu jedni druge i postaju zadovoljniji svojim poslom.

Iz navedenih principa proističu odluke koje se obaveznim praksama prevode u aktivnosti na projektu. Razvojni tim mora vrlo disciplinovano, tj. skoro do ekstrema, da primenjuje obavezne prakse, odakle i potiče naziv ovog agilnog metoda.

Pod *obaveznim praksama* se podrazumeva skup praktičnih mehanizama pomoću kojih se izvode aktivnosti u XP procesu razvoja softvera. Ovaj skup sadrži sledeće prakse:

- *programiranje u paru*. Podrazumeva rad dvoje programera istovremeno na istom računaru. Pošto rešavaju isti zadatak, jedan programer piše programski kôd i razmišlja o detaljima konkretne implementacije, dok drugi kontroliše napisani kôd imajući u vidu širi kontekst rešenja. Programeri povremeno razmenjaju svoje uloge. Parovi nisu fiksni, čak je bolje ukoliko se češće menjaju članovi para. Na taj način, svaki član tima upoznaje ceo sistem i delove posla koji rade drugi članovi tima, čime se poboljšava komunikacija na projektu. Dobra strana ovakvog programiranja je i ta što, ukoliko neki član napusti tim, drugi mogu lako da preuzmu njegov posao, bez većih poremećaja na projektu.
- *igra planiranja*. Ovo je glavni proces planiranja u XP-u. Igra se odvija na sastanku koji se održava jednom po iteraciji (obično na nedeljnom nivou). Tokom igre se generišu mape svih budućih verzija (sadržaj verzije i rokovi isporuke). Proces planiranja obuhvata dve aktivnosti. Prva je utvrđivanje zahteva koji su uključeni u dolazeću verziju i vreme njene isporuke. Ovo planiranje izvode zajedno razvojni tim i klijenti. Drugu vrstu planiranja vrši samo razvojni tim. Cilj ovog planiranja je da se utvrde i rasporede dalji zadaci na projektu i procene termini do kada oni treba da budu završeni.

- *razvoj vođen testovima*. U XP-u, testovi se pišu pre pisanja kôda. Ovakav pristup stimuliše programere da prilikom kodiranja vode računa o uslovima u kojima će njihov kôd biti testiran. Istestiranim kôdom se smatra onaj kôd koji se više ne može naći u uslovima u kojima ne bi radio ispravno.
- *celovitost tima*. Klijent nije samo onaj ko plaća razvoj softvera, već i neko ko će zaista koristiti sistem. Zato u XP-u korisnik treba permanentno da bude dostupan za razna pitanja razvojnog tima. Dobro je da se u tim uključi i neko od strane korisnika. Na primer, ako se razvija softver za finansijske poslove, u tim se može uključiti računovođa.
- *stalna integracija*. Razvojni tim uvek treba da radi na aktuelnoj (poslednjoj) verziji softvera. Pošto članovi tima istovremeno rade na različitim delovima kôda, unete izmene i poboljšanja obično čuvaju na lokalnim računarima. Zato je potrebno da se što češće (svakog sata ili na par sati) radi integracija sistema uključivanjem najnovijeg kôda generisanog od strane svih članova tima u aktuelnu verziju. Postoje specijalizovani softveri koji omogućavaju ovakav rad. Stalnom integracijom, izbegava se kašnjenje u kasnijim fazama projekta zbog problema u integraciji.
- *poboljšanje dizajna (refaktorizacija)*. Uzimanje u obzir samo trenutnih potreba i jednostavna rešenja (karakteristična za XP) mogu da dovedu do problema u razvoju. Oni se manifestuju, na primer, u neophodnom multipliciranju kôda pri dodavanju neke nove funkcionalnosti, ili u širokom uticaju učinjenih izmena u jednom delu kôda na mnoge druge delove kôda. Ako se pojave ovakvi problemi, to je u XP-u znak da treba uraditi refaktorizaciju kôda uz menjanje arhitekture i njeno uopštavanje.
- *male verzije*. Razvoj se vrši u malim iteracijama, čime se smanjuju troškovi izmena i neophodnosti odbacivanja neodgovarajućih rešenja. Svaka nova verzija daje se korisniku na upotrebu, čime se on aktivno uključuje u proces razvoja.
- *standardi kodiranja*. Standardi kodiranja podrazumevaju specifikaciju konzistentnog stila i formata izvornog kôda, uz poštovanje prirode izabranog programskog jezika. Moraju da ih se pridržavaju svi članovi razvojnog tima. Uvođenjem standarda izbegnuto je prilagođavanje na tuđi stil programiranja, što često oduzima mnogo vremena. Standardi, takođe, doprinose i boljem razumevanju u timu.
- *kolektivno vlasništvo kôda*. Kolektivno vlasništvo kôda podrazumeva da je svaki član razvojnog tima odgovoran za kompletan kôd. Istovremeno, svako može da menja bilo koji deo kôda u sistemu. Na ovaj način se izbegava bavljenje pogrešnim idejama i rešenjima i smanjuje otpor prema

promenama. Kôd se brže razvija, jer svaku grešku može da otkloni bilo koji programer. Međutim, davanjem mogućnosti svakom programeru da menja kôd, uvodi se rizik da programer unese grešku u sistem zbog nesagledavanja nekih zavisnosti koje postoje. To se rešava definisanjem dobrih testova.

- *jednostavan dizajn*. U XP-u, prilikom pisanja kôda, programer treba uvek da se pita da li postoji jednostavniji način da se realizuje zahtevana funkcionalnost. Ako postoji, treba izabrati taj način. Takođe, treba koristiti i refaktorizaciju da bi neki složeni kôd postao jednostavniji.
- *metafora*. Tim razvija svoj rečnik i terminologiju za obeležavanje osobina sistema i korišćenih elemenata. Na taj način se olakšava prepoznavanje čemu koja komponenta u sistemu (klasa, metoda i sl.) služi i postiže usaglašenost oko vizije rada sistema na nivou razvojnog tima. Pojačava se i osećanje pripadnosti timu.
- *održiv korak*. Na projektu je neophodno uspostaviti harmonični ritam rada održiv u dužem vremenskom periodu (dok traje projekat). To se može postići poštovanjem 40-časovne radne nedelje. Prekovremeni rad se u XP-u izbegava, jer se smatra da umorni ljudi više greše. Ukoliko ovakvo radno vreme nije dovoljno za uspešnu realizaciju projekta, onda projekat nije dobro ugovoren, tj. rokovi ili resursi nisu dobro procenjeni.

Da bi softver bio realizovan, potrebno je definisati konkretne aktivnosti koje razvojni tim treba da izvrši. Aktivnosti su usmerene ka postizanju visokog kvaliteta u što kraćem vremenu i uz što manje troškove. One se sprovode na način koji je striktno propisan obaveznim praksama.

Aktivnosti ekstremnog programiranja su:

- *kodiranje*. U XP-u, jedini zaista važan proizvod procesa razvoja jeste programski kôd. Ukoliko se ne proizvede kôd, smatra se da nema nikakvog rezultata. Programski kôd je vrlo jasan, precizan, nedvosmislen i zato se ne može izvršavati na više načina. U mnogim slučajevima, kôd pomaže u komunikaciji, kako između programera pri razjašnjavanju problematičnih situacija, tako i između programera i računara. Programski kôd izražava ne samo ideje za rešavanje problema, već i testove za proveru ispravnosti, primenjene algoritme, itd.
- *testiranje*. Testiranju u XP-u se posvećuje velika pažnja zato što se smatra da ako testiranje malog obima otklanja izvestan broj grešaka, testiranje velikog obima eliminiše mnogo više grešaka. Generišu se jedinični testovi i testovi prihvatanja. Jedinični testovi proveravaju da li je neka osobina

sistema realizovana na odgovarajući način. Broj ovih testova za datu osobinu nije ograničen, već se testovi pišu dok god se može zamisliti situacija u kojoj kôd ne bi ispravno radio. Testovi prihvatanja služe za proveru da li zahtevi, onakvi kako ih je razumeo programer, odgovaraju stvarnim zahtevima korisnika.

- *slušanje*. Pošto programeri u početku ne moraju ništa da znaju o poslovnoj logici koju sistem realizuje, neophodno je da aktivno slušaju klijenta kako bi se upoznali sa njom. Pri tome, programeri treba da ukazuju klijentu na to šta je lakše, a šta teže, realizovati i iz kojih razloga. Iz čestih razgovora sa klijentima, treba dobiti što više korisnih informacija koje bi doprinele boljem razumevanju sistema.
- *projektovanje*. Uprošćeno posmatrano, u XP-u je dovoljno sprovesti prethodne tri aktivnosti da bi se dobio finalni sistem, tj. programski kôd. Međutim, u praksi, to nije tako. Može se desiti da se prođe dug put u razvoju sistema, a onda da se pojavi nerešiv problem. Sistem može da postane previše složen, a zavisnosti u njemu vrlo nejasne. Problem se može prevazići projektovanjem strukture koja na organizovan način opisuje logiku sistema. Tako se otklanjaju mnoge zavisnosti u sistemu, jer je sistem tako organizovan da izmene u jednom delu ne utiču u velikoj meri na ostale njegove delove.

Ekstremno programiranje nalazi sve širu primenu u razvoju softvera. Prednosti ovakvog razvoja su evidentne kako za razvojni tim, tako i za kupce i menadžment projekta.

XP dopušta razvojnom timu da se fokusira na izradu softvera, ne vodeći mnogo računa o dokumentaciji i sastancima. Radna atmosfera je prijatnija, ima mnogo mogućnosti za učenje i dalje napredovanje, a radno vreme je ograničeno na osam sati dnevno.

Kraće vreme razvoja softvera, uz relativno malo nedostataka pogoduju kupcima. Oni mogu da promene svoje mišljenje uz minimalne troškove i malo prigovora od strane razvojnog tima.

Sa stanovišta menadžmenta projekta, XP generiše dobar softver po prihvatljivoj ceni. Rizik je smanjen time što se izmene prihvataju u svakom trenutku razvoja, uz permanentno postojanje validnog radnog kôda. Osim toga, ne postoji zavisnost od pojedinaca koji bi bili nezamenljivi na projektu, što stvara bolje odnose u timu, pa članovi tima ređe odlaze.

Osnovni nedostaci ekstremnog programiranja su:

- metod se teško sprovodi

- nije lako naći dovoljan broj programera koji bi prihvatili i izvodili obavezne prakse, jer to zahteva strogu disciplinu
- klijentima ne mora da odgovara ideja da budu uključeni u projekat, jer imaju druge obaveze
- teško je uklopiti različite karaktere članova razvojnog tima u skladnu celinu koja bi mogla savršeno da funkcioniše

3 Analiza zahteva

Prvi korak u procesu razvoja nekog softvera je evidentiranje zahteva koje taj softver treba da ispuni. Skup zahteva nastaje kao rezultat analize koju treba sprovesti u cilju razumevanja osnovnih problema i potreba naručioca. Analiza podrazumeva intenzivnu saradnju sa naručiocima, a posebno sa korisnicima softvera. Od uspešnosti ove analize u mnogome zavisi ishod celog projekta razvoja softvera. Grupa Standish je 1994.godine obavila istraživanje u vezi sa ishodima više od 8000 softverskih projekata u preko 350 softverskih kompanija. Došli su do saznanja da je oko 35% projekata otkazano pre nego što je završeno, a da je samo 9% projekata isporučeno na vreme i u okviru planiranog budžeta. Pokazalo se da su razlozi za ovako loše rezultate uglavnom u direktnoj vezi sa definisanjem i upravljanjem zahtevima. Naime, proces definisanja zahteva je prilično složen i ne predstavlja samo jednostavno navođenje zahteva koje softver treba da ispuni. On zahteva pažljivu analizu kako bi se na pravi način shvatili pojedinačni zahtevi, kao i veze koje postoje između njih. Problemi u vezi sa zahtevima mogu da nastanu iz dva razloga. Prvo, ukoliko skup zahteva nije potpun ili nije adekvatno definisan, to direktno ugrožava uspešnost projekta. Drugo, u praksi je često nemoguće definisati kompletan i dovoljno precizan skup zahteva na samom početku rada na projektu, jer postoji realna mogućnost pojave novih zahteva u kasnijim fazama projekta. U ovom slučaju, razvojni tim treba da bude sposoban i spreman da na pravi način odreaguje na pojavu novih zahteva kako projekat ne bi bio ugrožen. Boehm i Papaccio su 1988.godine sprovedi istraživanje koje je pokazalo da otkrivanje i otklanjanje grešaka u zahtevima u kasnijim fazama projekta ima vrlo visoku cenu. Naime, otklanjanje iste greške u fazi projektovanja sistema košta pet puta više nego u fazi definisanja zahteva, u fazi pisanja programa deset puta više, a ako je greška otkrivena nakon isporuke softvera čak dve stotine puta više.

Zahtevi koje softver treba da ispuni mogu da budu vrlo raznovrsni, da potiču iz različitih izvora, pa čak da budu i kontradiktorni. Stoga je važno da se pravilno

odabere metod specifikacije zahteva koji odgovara razmatranom projektu. Na izbor metoda utiču razni faktori, kao što su obim projekta, njegova složenost i važnost. Od suštinskog značaja je da na kraju ove faze u procesu razvoja softvera budemo sigurni da su zahtevi ispravno definisani i da je njihov skup potpun u skladu sa trenutnim uslovima.

Prilikom naručivanja softvera, naručilac ima opštu ideju šta bi taj softver trebalo da radi. Potreba za softverom obično nastaje u sledećim slučajevima:

- kada treba automatizovati poslove koji su se do tada obavljali ručno (na pr. elektronsko plaćanje računa, elektronsko naručivanje robe i sl.)
- kada treba poboljšati ili proširiti postojeći sistem (na pr. dodavanje novih usluga u postojeći telefonski sistem, proširivanje skupa mogućih načina kreditiranja i sl.)
- kada treba napraviti sistem koji obavlja neki posao koji do tada nije rađen (na pr. automatsko kontrolisanje doziranja nekog leka, elektronsko izdavanje novina i sl.)

Bez obzira kako je nastala potreba za softverom, tj. da li se radi o potpuno novom softveru ili o izmeni postojećeg, softver poseduje namenu i ciljeve koje treba da ispuni.

Zahtev predstavlja izraz željenog ponašanja softvera. Prilikom definisanja zahteva, razmatraju se osobine objekata i entiteta koji postoje u datom sistemu, stanja u kojima se objekti i entiteti mogu naći, kao i funkcije koje omogućavaju promene stanja ili osobina objekata. Na primer, pretpostavimo da je potrebno razviti softver za obračun plata u kompaniji naručioca. Jedan od zahteva bi mogao da bude da se svakog meseca štampaju liste sa obračunom zarade za svakog radnika. Drugi zahtev može da bude mogućnost uplate stimulacije za radnike koji se posebno ističu, a treći zahtev da se sistemu može pristupati sa više različitih lokacija u kompaniji. Navedeni zahtevi opisuju pojedinačne funkcije sistema koje su u direktnoj vezi sa opštom namenom sistema, tj. obračunom plata.

Zahtevi mogu da identifikuju objekte ili entitete u sistemu (na pr. „Radnik je osoba koja radi u kompaniji.“), da definišu ograničenja za pojedine entitete (na pr. „Radnik ne može biti plaćen za više od 40 sati rada nedeljno.“), ili da opisuju odnose između entiteta (na pr. „Radnika X nadgleda radnik Y, ukoliko je Y ovlašćen da izmeni zaradu koju prima X.“). Posebno je važno ustanoviti način interakcije sistema sa okruženjem.

Cilj analize zahteva je da se precizno ustanovi kakvo ponašanje naručilac očekuje od softvera. Pri tome se uopšte ne vodi računa (osim ako naručilac to

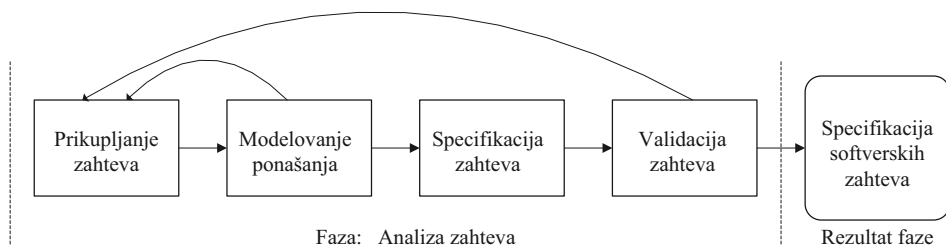
eksplicitno ne zahteva) o tome kako će to ponašanje biti implementirano u sistemu. To znači da se u ovoj fazi ne razmatra izbor tehnologije u kojoj će sistem biti realizovan, izbor baze podataka, arhitekture sistema koja će se koristiti i sl.

Tokom analize, zahtevi se obično izražavaju opisima iz realnog okruženja naručioca, bez upotrebe stručnih termina koji će biti korišćeni u sistemu. Na primer, zahtevi u vezi sa obračunom plata referišu se na radnike, stimulacije i dr., a ne na sistemске procedure i podatke. Razlog za ovakvo izražavanje zahteva je u želji da se ostvari što bolje razumevanje sa naručiocem, kako bi se izbegla različita tumačenja istih osobina i pojava. Naručiocu je lakše da svoje potrebe iskaže u kategorijama sopstvenog poslovanja, dok projektanti moraju da se prilagode naručiocu. Međutim, i projektanti ovakvim pristupom dobijaju određene pogodnosti. Njima je ostavljena maksimalna fleksibilnost u odlučivanju kako će sistem ispuniti postavljene zahteve (bez ikakvog uticaja naručioca).

Analizu zahteva izvodi analitičar zahteva. Tokom analize, analitičar zahteva treba da obavi sledeće aktivnosti:

- *prikupljanje zahteva.* Analitičar prikuplja zahteve kroz razgovor sa naručiocem, čitanjem dostupnih materijala koji su od interesa, posmatranjem aktuelnih ponašanja u okruženju i dr.
- *modelovanje ponašanja.* Evidentirane zahteve analitičar analizira i formira model ili prototip ponašanja sistema. Model mu pomaže da bolje razume zahteve i njihove međusobne veze. U ovoj aktivnosti često se otvaraju nova pitanja koja je potrebno razmotriti sa naručiocem, a koja nisu bila predmet prethodne aktivnosti.
- *specifikacija zahteva.* Pošto je postignuto dobro razumevanje zahteva, prelazi se na izradu specifikacije u okviru koje se definiše koji će delovi zahtevanog ponašanja biti implementirani u softveru.
- *validacija i verifikacija zahteva.* U ovoj aktivnosti se proverava da li specifikacija odgovara onome što naručilac očekuje od softverskog proizvoda. Ovde se mogu otkriti propusti u modelu ili specifikaciji koji se rešavaju ponovnom posetom naručiocu ili vraćanjem na neku od prethodnih aktivnosti.

Rezultat analize zahteva sprovedene kroz navedene aktivnosti je Specifikacija softverskih zahteva (*Software Requirements Specification – SRS*) koja se dalje koristi za komunikaciju sa razvojnim timom. Postupak definisanja skupa zahteva je prikazan na slici 3.1.



Slika 3.1 Definisanje skupa zahteva

U nastavku je detaljno razmotrena svaka od navedenih aktivnosti.

3.1 Prikupljanje zahteva

Prikupljanje zahteva predstavlja izuzetno važnu aktivnost u procesu definisanja zahteva, jer se tokom nje utvrđuje šta naručioci i korisnici stvarno žele. Tehnike koje se tom prilikom koriste mogu biti različite.

Ukoliko naručeni softver treba da automatizuje neki posao koji se do tada obavljao ručno, ili se radi o poboljšanju postojećeg sistema, do skupa zahteva može se doći na relativno lak način. Analitičar zahteva treba najpre da prouči proces obavljanja posla, a zatim da, postavljanjem pravih pitanja, od korisnika dobije odgovore koje može da pretoči u zahteve.

Međutim, ako je problem potpuno nov, neophodna je intenzivnija saradnja sa naručiocima i korisnicima, pošto oni nemaju praktična iskustva u vezi sa problemom. Naručilac poznaje svoj posao, ali ne može uvek na pravi način nekome sa strane da opiše svoje poslovne probleme. Nekad mu je teško da dovoljno precizno iskaže ono što mu treba, pri tome može da koristi žargonske reči ili pretpostavke sa kojima drugi nisu bliski. S druge strane, ni analitičar zahteva ne može uvek dovoljno dobro da shvati tuđe poslovne probleme, jer ne mora dobro da poznaje domen poslovanja. I on može da koristi sopstveni žargon ili stručne termine koji nisu bliski naručiocu. Imajući ovo u vidu, u ranoj fazi projekta, dešava se da su zahtevi nedovoljno dobro formulisani ili pogrešno shvaćeni.

Da bi se došlo do dobro definisanog skupa zahteva, neophodno je najpre, u zajedničkom dogovoru, uskladiti korišćenu terminologiju. Ukoliko se ne postigne da i naručilac i analitičar tumače isti termin na isti način, projekat je osuđen na neuspeh.

Iako većina analitičara smatra da treba da se teži formiranju što potpunijeg i doslednijeg skupa zahteva, postoje i drugačija mišljenja. Easterbrook i Nuseibeh su 1996.godine dokazivali da je često bolje tolerisati nedoslednosti koje se pojave prilikom definisanja zahteva, a u nekim slučajevima ih čak i podsticati. Oni smatraju da je u ranoj fazi procesa definisanja zahteva besmisleno ulagati veliki trud u rezešenje nadoslednosti, jer to može biti težak, skup, a ponekad i nemoguć posao. Razlog za ovakvo mišljenje je u tome što se tokom projekta kod svih učesnika akumulira znanje iz date oblasti, čime se poboljšava razumevanje problema. Nedoslednosti treba uočiti, ali se njima ne treba baviti dok ne bude dovoljno informacija za donošenje prave odluke. Na ovaj način se izbegavaju i greške vezane za fazu projektovanja sistema.

U procesu prikupljanja zahteva, osim već pomenutih, naručioca, korisnika i analitičara zahteva, važnu ulogu mogu da imaju i drugi zainteresovani subjekti, kao na primer, stručnjaci iz konkretne oblasti primene razmatranog softvera (pri obračunu zarada mogu se konsultovati poreski stručnjaci), istraživači tržišta (određuju buduće trendove i potrebe potencijalnih korisnika), softverski inženjeri (mogu naručiocu da preporučé nove funkcionalnosti koje su dostupne zahvaljujući novim hardverskim i softverskim tehnologijama) i dr. Svaki od ovih subjekata ima svoj pogled na sistem, pri čemu ovi pogledi ponekad mogu da budu i protivrečni. Na primer, korisnik može da zahteva lako korišćenje sistema, a da to povlači za sobom znatno sporiji rad, što je neprihvatljivo sa inženjerskog aspekta posmatranja. Analitičar zahteva mora da razume svaki pogled i da formuliše zahteve tako da odražavaju interese svih učesnika. Pri tome, treba da ima u vidu i brojne predrasude, kako korisnika i naručioca, tako i projektanata sistema. Na primer, korisnici obično misle da projektanti ne razumeju operativne probleme, stavljaju preveliki naglasak na tehnička pitanja, pokušavaju da korisnicima nametnu kako treba da rade svoj posao i sl. Nasuprot tome, projektanti često vide korisnike kao nedovoljno sposobne da formulišu svoje potrebe, da im dodele prioritete, da preuzmu odgovornost za sistem i sl. Iz ovoga sledi da dobar sistem analitičar najpre mora da ima izuzetno dobre međuljudske odnose sa svim učesnicima u projektu, a takođe, da ima i odgovarajuća tehnička znanja i sposobnosti.

U fazi prikupljanja zahteva, analitičar zahteva koristi različite tehnike:

- *razgovor* sa zainteresovanim subjektima, uz postavljanje pitanja i dobijanje odgovora; dobra praksa je razgovor u grupama, jer učesnici postaju inspirisani idejama drugih
- *pregledanje raspoložive dokumentacije*, kao što su dokumentovane procedure, korisnička uputstva za rad, razne vrste šema, i dr.

- *upoznavanje postojećeg sistema* u cilju prikupljanja informacija o tome kako korisnici zaista obavljaju predviđene aktivnosti; vrlo je bitno sagledati sistem u celini kako se ne bi desilo da se nakon isporuke novog sistema pojedine aktivnosti i dalje obavljaju ručno, jer ih projektanti sistema nisu predvideli
- *učenje posla od korisnika*; dobro je da se analitičar detaljnije upozna sa poslovima koje korisnik izvršava dok ih on radi
- *upotreba strategija specifičnih za datu oblast* podrazumeva korišćenje poznatih teorijskih znanja (na pr. strategija PIECES – Wetherbe 1984. u oblasti informacionih sistema) sa ciljem da zainteresovani subjekti više vode računa o specifičnostima zahteva u njihovoj konkretnoj situaciji
- *poređenje sa sličnim, ranije razvijenim sistemima* čija iskustva mogu znatno da poboljšaju kvalitet softvera

3.1.1 Vrste zahteva

Prilikom formulisanja skupa zahteva, analitičar zahteva mora detaljno da analizira ne samo potrebnu funkcionalnost sistema, već i razne vrste ograničenja koje sistem mora da ispuni. Mogu se izdvojiti sledeće vrste zahteva:

- *funkcionalni zahtevi*. Ovi zahtevi opisuju ponašanje sistema, tj. daju odgovore na pitanja šta sistem treba da radi, koje usluge treba da pruži, kakav je format ulaznih i izlaznih podataka, kako sistem treba da reaguje na određeni ulazni podatak, kako se menja ponašanje sistema u vremenu, itd. Na primer, u slučaju obračuna plata, funkcionalni zahtevi definišu koji su ulazni podaci neophodni za bi plate mogle biti obračunate, koliko često će biti štampani listići za plate, pod kojim uslovima se može promeniti iznos za isplatu, koji su razlozi uklanjanja radnika sa platnog spiska i sl. Funkcionalni zahtevi određuju granice prostora rešenja razmatranog problema.
- *zahtevi u pogledu kvaliteta*. Ovi zahtevi opisuju koje osobine treba da ima softver da bi se moglo reći da je prihvatljiv sa stanovišta kvaliteta. Na primer, može se reći da je softver kvalitetan ukoliko ima kratko vreme odziva, ako je pogodan za učenje i korišćenje, ako je pouzdan, ima niske troškove održavanja i sl. Zahtevi u pogledu kvaliteta mogu da budu vrlo detaljni i da se odnose na:
 - *performanse sistema* (vreme izvršenja, vreme odziva, protokol podataka i dr.)

-
- *upotrebljivost sistema* (lakoća korišćenja, moguće vrste obuke, mogućnost neadekvatne upotrebe sistema i sl.)
 - *bezbednost sistema* (kontrola pristupa, izolovanost podataka po korisnicima, šifrovanje podataka, fizička bezbednost i sl.)
 - *pouzdanost i raspoloživost sistema* (detekcija grešaka, srednje vreme između otkaza, vreme za pokretanje sistema nakon otkaza, povremeno pamćenje kopija podataka i dr.)
 - *održavanje sistema* (ispravljanje grešaka, lakoća izvođenja manjih izmena, mogućnost značajnijih izmena u cilju poboljšanja sistema, mogućnost prelaska na drugu platformu i dr.)
 - *preciznost i tačnost podataka*.
 - *projektna ograničenja*. Ova ograničenja nastaju kao posledica donetih odluka na projektu. Na primer, jedno ograničenje je izabrana platforma na kojoj će softver raditi. Projektna ograničenja se mogu odnositi na:
 - *fizičko okruženje* (lociranje opreme, potrebni atmosferski uslovi u smislu temperature, vlažnosti i sl., napajanje, grejanje, klimatizacija i dr.)
 - *povezivanje sistema sa okolinom* (format ulaznih i izlaznih podataka, interakcija sa drugim sistemima, način preuzimanja i prosleđivanja podataka i sl.)
 - *implementaciju* (izbor platforme, izbor programskog jezika, korišćene tehnologije i dr.)
 - *procesna ograničenja*. Ova ograničenja se odnose na tehnike i resurse koji će biti korišćeni u izgradnji sistema. Na primer, naručilac može da insistira na primeni agilnih metoda kako bi što ranije mogao da koristi određene delove sistema. Procesna ograničenja po pitanju resursa utvrđuju koji materijali će biti korišćeni u realizaciji, koje osoblje će biti angažovano na projektu, koje sposobnosti su neophodne za obavljanje određenih poslova, itd. Procesna ograničenja proističu i iz odluke o potrebnoj dokumentaciji (tipovi dokumenata, obim dokumenata, dokumentovanje na računaru, u papirnoj formi, ili oba).

Poštovanje navedenih vrsta zahteva doprinosi sužavanju prostora mogućih rešenja problema, tako što se neka od njih proglašavaju prihvatljivim za realizaciju, dok se druga mogu proglasiti neupotrebljivim.

3.1.2 Razrešavanje konflikata

Prilikom formulisanja skupa zahteva, analitičar zahteva pokušava da ispuni želje svih zainteresovanih subjekata. Međutim, dešava se da pojedini subjekti imaju različita mišljenja o tome kakvi zahtevi treba da budu. Stoga je neophodno ustanoviti mehanizam za razrešavanja ovakvih konflikata.

Da bi nastali konflikti bili što jednostavnije rešeni, potrebno je od naručioca zahtevati da odredi prioritete postavljenih zahteva. U zavisnosti od prioriteta, zahtevi se mogu klasifikovati u tri kategorije:

- *suštinski zahtevi*. Ovi zahtevi moraju da budu ispunjeni u konačnoj verziji softvera.
- *poželjni zahtevi*. Ispunjenje ovih zahteva nije neophodno, ali bi bilo dobro kada bi sistem mogao i njih da ispuni.
- *opciono zahtevi*. Ovi zahtevi se mogu ispuniti, ali se mogu i izostaviti iz konačne verzije sistema.

Na primer, kod obračuna plata, suštinski zahtev bi bio proračun iznosa koji treba da bude isplaćen svakom radniku. Poželjan zahtev bi bio da iznos za isplatu bude razložen na stavke koje utiču na njega, dok bi opciono zahtev bio da se tekst platne liste štampa u crnoj boji sa naglašenim odbircima u crvenoj boji.

Dodeljivanje prioriteta zahtevima je vrlo korisno zato što pomaže svim učesnicima na projektu da bolje razumeju šta je stvarno potrebno realizovati. Osim toga, ako na projektu postoje stroga vremenska ograničenja po pitanju njegovog zavšetka, ili ograničenja po pitanju predviđenog budžeta, prioriteta zahteva se mogu iskoristiti da bi se ova ograničenja ispunila. Na primer, ako je vreme za razvoj kratko, ili nema dovoljno novca, obično se opciono zahtevi izostavljaju, dok se poželjni zahtevi analiziraju sa ciljem eliminisanja nekih od njih ili odlaganja njihove realizacije za kasnije verzije.

Konflikti se najčešće javljaju kod zahteva u pogledu kvaliteta. Često se dešava da nikakvom optimizacijom nije moguće zadovoljiti dva kontradiktorna zahteva ovog tipa. Na primer, teško je istovremeno zadovoljiti zahteve za lakim održavanjem sistema i kratkim vremenom odziva, jer se lako održavanje obično ostvaruje enkapsulacijom i razdvajanjem nadležnosti, što usporava sistem. Ili, prilagođavanje sistema za efikasan rad na jednoj platformi ugrožava njegovu prenosivost na druge platforme. Dodeljivanje prioriteta zahtevima od strane naručioca pomaže projektantima da na razuman, ako ne i optimalan način reše nastale konflikte, tako što izlaze u susret zahtevima koji su za naručiocima najvažniji.

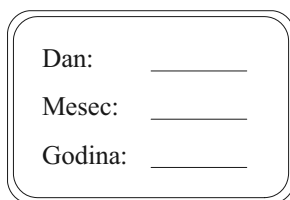
Ukoliko projektanti nikako ne mogu da razreše nastale konflikte po pitanju zahteva, moraju preći na pregovaranje i ponovno ocenjivanje pogleda svih zainteresovanih subjekata. To nije nimalo jednostavan proces, jer zahteva toleranciju, strpljenje i iskustvo kako bi se došlo do svima prihvatljivog rešenja. Subjekti se retko razilaze po pitanjima vezanim za funkcionalnost sistema. Sukobi nastaju uglavnom oko ograničenja vezanih za rešavanje problema (na pr. koji će sistem za upravljanje bazama podataka biti korišćen, koji algoritmi za šifrovanje, koja vrsta korisničkog interfejsa i dr.). Najveći problemi nastaju kada ne postoji saglasnost oko dodeljenih prioriteta. Na primer, univerzitetski odseci mogu da zahtevaju da različita pravila ocenjivanja važe za različite odseke, dok bi administrativnoj službi više odgovaralo da su pravila ujednačena. U ovom slučaju, treba najpre tačno utvrditi razloge nepopustljivosti obe strane i sa njima upoznati sve subjekte. Na taj način će svi zainteresovani subjekti bolje razumeti probleme i potrebe drugih i proceniti da li su oni veći ili manji od njihovih sopstvenih problema i potreba. Na kraju dobro vođenog pregovaračkog procesa, obično se dobijaju rešenja koja su prihvatljiva za sve učesnike.

3.1.3 Prototipovi zahteva

U procesu definisanja zahteva, naručioci često nisu sigurni šta tačno žele i šta im je potrebno. Tada se kao rezultat dobija lista zahteva sa veoma malo detalja i bez naznake da li je skup zahteva potpun. Međutim, kada softverski proizvod bude gotov, naručioci uglavnom znaju da li sistem zadovoljava njihove potrebe ili ne i tada može da dođe do problema. Do ovoga dolazi i zato što je lakše kritikovati postojeći proizvod, nego detaljno zamisliti novi. Ukoliko naručilac nije u stanju da jasno ukaže na zahteve, jedini način da se sazna više detalja o proizvodu jeste da se napravi njegov prototip, a zatim da se od naručioca zahteva povratna informacija o tome da li taj prototip zadovoljava njegove zahteve. Pod prototipom se podrazumeva delimično razvijen proizvod koji omogućava sagledavanje različitih aspekata sistema. Na osnovu prototipa, korisnik može da uoči koja su poboljšanja neophodna, koje osobine nisu dovoljno korisne, koja funkcionalnost nedostaje, itd. Prototip može da pomogne i projektantu da ispita izvodljivost predloženog rešenja, kao i mogućnost optimizacije zahteva sa aspekta njihovog kvaliteta.

Izrada prototipa je u nastavku objašnjena na jednom primeru. Pretpostavimo da su naručioci softvera psiholozi i predavači koji izvode vežbe, a korisnici sistema njihovi klijenti, tj. oni koji rade vežbe. S obzirom da korisnicima ovog sistema računari ne moraju biti bliski, od izuzetne važnosti je kako je izrađen korisnički interfejs. Na primer, neka korisnici treba da unesu datume izvođenja vežbi. Ukoliko predavači nisu sigurni kako bi taj unos trebalo da izgleda, pogodno je izraditi prototip koji demonstrira mogućnosti unosa. Na slici 3.2 a) prikazan je prvi

prototip u kojem korisnik mora da unese dan, mesec i godinu kada vežba. Lakši način za unos je dat u prototipu 3.2 b) u kome korisnik samo mišem treba da selektuje određeni datum.

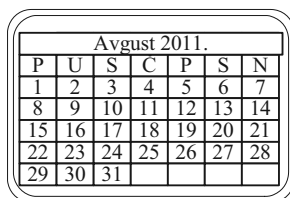


Dan: _____

Mesec: _____

Godina: _____

a)



Avgust 2011.						
P	U	S	C	P	S	N
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

b)

Slika 3.2. Prototipovi korisničkog interfejsa za unos datuma

Izrada prototipova u ovom primeru pomaže da se odabere pravi izgled ekrana za interakciju sa korisnikom. Oni vizualizuju zahtev i time olakšavaju donošenje odluke (lakše je doneti odluku na osnovu prototipa nego na osnovu opisa datog rečima).

Pri izradi prototipova mogu se primeniti dva pristupa: izrada ilustrativnog prototipa i izrada evolutivnog prototipa. *Ilustrativni prototip* predstavlja softver koji se razvija radi boljeg upoznavanja sa problemom i on ne ulazi u softver koji se isporučuje korisniku. Pri izradi ovog prototipa, ne mora se voditi računa o kvalitetu izrađenog softvera, njegovoj strukturiranosti, efikasnosti, proveriti grešaka i sl. On jednostavno predstavlja samo fasadu koja implementira željenu funkcionalnost i brzo dovodi do suštine problema ili predloženog rešenja. Kada se dobiju željeni odgovori, ilustrativni prototip se odbacuje i pristupa se inženjerskom poslu na izradi softvera koji će biti isporučen. Za razliku od ilustrativnog, *evolutivni prototip* predstavlja softver koji se razvija ne samo da pomogne u nalaženju odgovora na određena pitanja, već da postane sastavni deo softverskog proizvoda koji će biti isporučen. Zato evolutivni prototip mora vrlo pažljivo da se razvija tako da zadovolji zahteve u pogledu kvaliteta (na pr. brzinu odziva, modularnost, itd.).

3.2 Modelovanje ponašanja

Modelovanje ponašanja sistema na osnovu prikupljenih zahteva doprinosi:

- boljem razumevanju zahteva
- lakšem uočavanju pitanja koja treba postaviti naručiocu ili korisniku
- lakšem nalaženju nedostataka u smislu nepoznatog ponašanja u pojedinim situacijama
- lakšem otkrivanju nedoslednosti u zahtevima (na primer, ako se ponovnim dovođenjem istog ulaznog signala dobijaju različite vrednosti na izlazu)

Tokom procesa modelovanja, osim podizanja nivoa razumevanja problema, svi učesnici stiču i nova znanja, jer neki aspekti problema postaju jasniji i očigledniji tek tokom modelovanja.

U literaturi postoji veliki broj metoda i notacija za modelovanje zahteva. Međutim, mnoge od ovih metoda imaju velike sličnosti, pa se mogu izdvojiti samo desetak različitih paradigmi. U nastavku su opisane tri od njih.

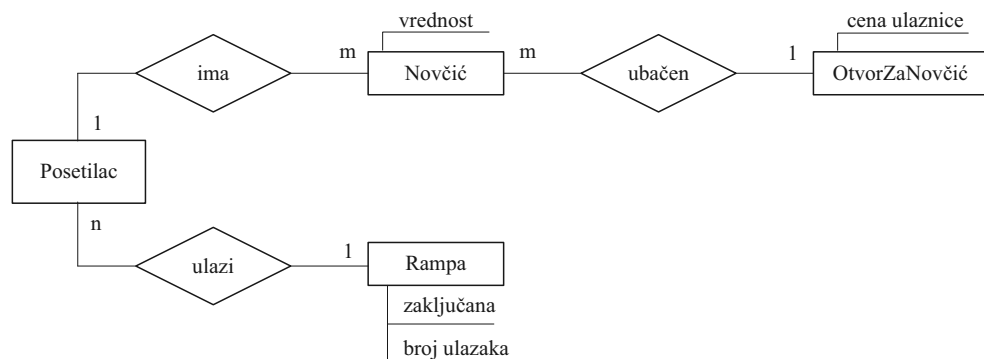
3.2.1 ER dijagrami

U ranoj fazi definisanja zahteva, korisno je napraviti konceptualni model problema koji podrazumeva identifikovanje relevantnih objekata i entiteta u sistemu, uočavanje njihovih osobina i međusobnih odnosa. ER dijagram (*Entity-Relationship diagram*) je grafička notacija za predstavljanje konceptualnih modela koju je 1976.godine uveo Chen. Ovaj metod se može koristiti ne samo za modelovanje zahteva, već i za modelovanje dizajna sistema, strukture softverskog proizvoda, baze podataka, itd.

Izrada ER dijagrama će biti ilustrovana na primeru softverski kontrolisane obrtno kapije na ulazu u zoološki vrt, koji su 1995.godine dali Jackson i Zave. Obrtna kapija radi tako što, kada se u predviđeno mesto na kapiji ubaci novčić, kapija se otključava i posetilac može da je gurne i uđe u vrt. Nakon što kapija zarotira i omogući jedan ulazak, ponovo se sama zaključava, tako da sledeći posetilac ne može da uđe u vrt ukoliko na plati kartu, tj. ponovo ubaci novčić.

Osnovni elementi na ER dijagramu su: entitet, atribut i relacija. Pod entitetom se podrazumeva skup objekata (klasa objekata) iz realnog sveta koji imaju zajedničke osobine i ponašanje. U datom primeru sa obrtnom kapijom, entiteti

mogu biti *Posetilac*, *Novčić*, *OtvorZaNovčić* i *Rampa*. Oni se na ER dijagramu predstavljaju pravougaonicima. Atributima se opisuju svojstva entiteta. Tako, atribut entiteta *Novčić* može biti *vrednost*, atributi entiteta *Rampa* mogu biti *zaključana* i *broj ulazaka*, a atribut entiteta *OtvorZaNovčić*, *cena ulaznice*. Relacije definišu tip odnosa između dva entiteta i na dijagramu se predstavljaju linijom koja spaja dva entiteta na čijoj sredini se nalazi romb u kome je naveden tip veze. Na krajevima relacija često se prikazuje kardinalnost veze koja označava broj entiteta koji mogu učestvovati u vezi. Na primer, relacija između entiteta *Novčić* i *OtvorZaNovčić* može biti *ubačen*. Pošto se novčići mogu ubacivati samo u jedan otvor, kardinalnost na kraju relacije prema entitetu *OtvorZaNovčić* je 1. Međutim, u zavisnosti od vrednosti raspoloživih apoeni i cene ulaznice, posetilac može da ubaci jedan ili više (*m*) novčića u otvor. To se opisuje kardinalnošću *m* na strani entiteta *Novčić*. ER dijagram za slučaj obrtno rampe prikazan je na slici 3.3.



Slika 3.3 Model obrtno rampe predstavljen ER dijagramom

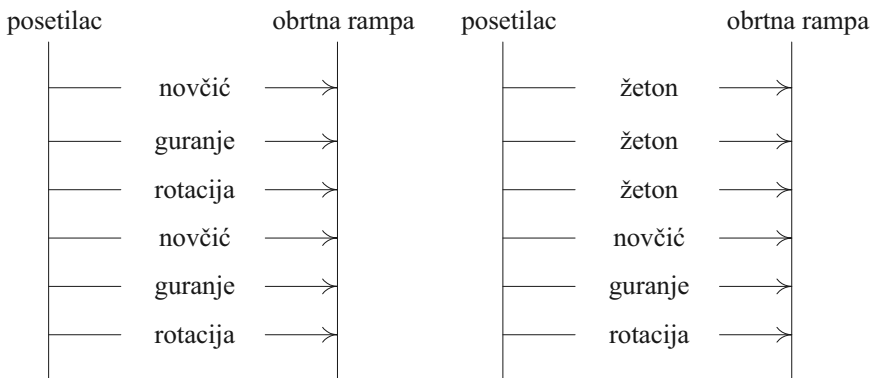
ER dijagrami se smatraju stabilnim jer preko entiteta uključuju sve učesnike. Moguće izmene u zahtevima obično se odnose na promenu ponašanja jednog ili više entiteta u skupu, dok se sam skup entiteta retko menja. Ovakav način modelovanja deluje jednostavno, ali često nije baš tako. Naime, nivo detalja modelovanja nije uvek lako odrediti. Na primer, može se postaviti pitanje da li otvor za novac i rampa treba da budu predstavljeni pomoću dva entiteta (kao na slici), ili jednim apstraktnijim entitetom kao što je *Obrtna_rampa*. Takođe, nije lako odlučiti koji podaci predstavljaju entitete, a koji attribute. Uvek postoje argumenti za i protiv svakog mogućeg izbora. Osnovni kriterijumi za donošenje odluka treba da budu: da li neka mogućnost pojašnjava opis i da li izabrana mogućnost bez potrebe uvodi neka ograničenja.

ER notacija se često koristi u složenijim pristupima. Na primer, u UML jeziku za modelovanje, ER notacija se koristi u dijagramu klasa, pri čemu klase predstavljaju realne entitete bitne za rešavanje problema.

3.2.2 Tragovi događaja

ER dijagrami pružaju strukturiran pogled na problem, ukazujući na entitete važne za njegovo rešavanje, osobine entiteta i odnose koji postoje između njih. Međutim, ova notacija ne govori ništa o ponašanju entiteta. Stoga je bilo potrebno razviti nove notacije za modelovanje ponašanja sistema. Jedna od njih su tragovi događaja.

Tragovi događaja predstavljaju grafički opis niza događaja koji se u stvarnosti razmenjuju između entiteta. Opis se sastoji od vertikalnih i horizontalnih linija. Vertikalne linije odgovaraju vremenskim osama na čijim vrhovima se nalaze nazivi entiteta na koje se linije odnose. Horizontalne linije predstavljaju same događaje, tj. interakciju između entiteta, i mogu se shvatiti kao poruke koju pojedini entiteti šalju drugim entitetima. Vreme duž ose teče odozgo na dole. Svaki dijagram prikazuje po jedan trag koji predstavlja jedno od više mogućih ponašanja. Na slici 3.4 data su dva traga za slučaj obrtne rampe. Na levoj strani slike, dat je trag koji opisuje uobičajeno ponašanje rada rampe. Na desnoj strani je trag koji prikazuje neuobičajeno ponašanje, tj. situaciju kada posetilac pokuša da u otvor za novčić ubaci nešto drugo, tj. žeton.



Slika 3.4 Model obrtne rampe predstavljen tragovima događaja

Projektanti i naručioci često primenjuju notaciju tragova događaja jer ona ima preciznu semantiku i laka je za razumevanje. Jednostavnost notacije potiče od mogućnosti dekompozicije zahteva na scenarije, pri čemu se onda svaki scenario može modelovati posebnim tragom.

Tragovi događaja se ne koriste za modelovanje celokupnog zahtevanog ponašanja, jer bi u tom slučaju broj scenarija bio vrlo veliki. Uglavnom se koriste u početnoj fazi projekta kada je potrebno postići usaglašenost po pitanju ključnih zahteva na projektu i identifikovati najvažnije entitete.

Kao i ER dijagrami, i tragovi događaja se često koriste u složenijim pristupima. Na primer, u UML jeziku za modelovanje, tragovi događaja se koristi u dijagramu sekvence.

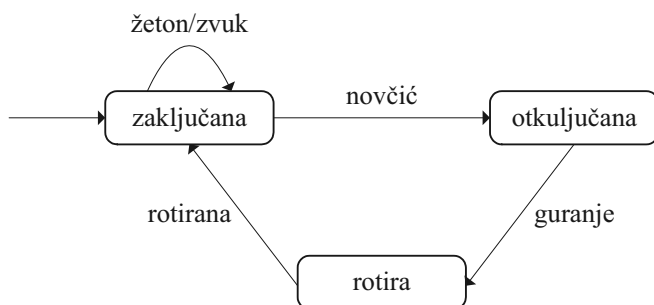
3.2.3 Konačni automati

Konačni automat predstavlja grafički opis komunikacije između sistema i njegovog okruženja. On objedinjuje sve tragove događaja u okviru jedinstvenog modela.

Notacija se sastoji od čvorova koji predstavljaju stanja u kojima konačni automat može da se nađe. Stanju odgovara stabilan skup uslova koji važe u intervalu između dva događaja. Čvorovi su povezani granama koje označavaju prelaze iz stanja u stanje. Do promene stanja dolazi usled pojave nekog događaja koji menja uslove u sistemu. Zato je svaki prelaz iz jednog stanja u drugo označen pobudnim događajem, a ponekad i izlaznim događajem (u dijagramu mu prethodi simbol „/“).

Konačni automati se koriste za definisanje dinamičkog ponašanja, tj. promena u ponašanju sistema nastalih usled odvijanja nekih događaja. Ovaj metod je posebno pogodan za modelovanje različitog ponašanja izlaza sistema kada se na njegov ulaz dovede fiksna vrednost. Naime, postoje sistemi kod kojih izlaz zavisi ne samo od ulaza, već i od veličina koje definišu tekuće stanje sistema.

Na slici 3.5 dat je model konačnog automata za slučaj obrtne rampe.



Slika 3.5 Model obrtne rampe predstavljen konačnim automatom

Kao što se vidi, po datom modelu, obrtna rampa može biti u tri stanja: *zaključana*, *otkjučana* i *rotira*. Ponašanje rampe zavisi od pobudnog događaja. Ukoliko se desi neki neregularan događaj, na primer *guranje* kada je rampa u stanju *zaključana*, taj događaj će biti odbačen jer ga nema u modelu. Ovaj pobudni događaj bi se mogao uvesti kao prelaz iz stanja *zaključana* u stanje *zaključana*. Međutim, uključivanje ovakvih prelaza ne bi imalo nikakvog efekta, već bi samo

opteretilo sistem, i zato se oni i ne uvode. Ima smisla uključiti samo one prelaze koji imaju vidljiv efekat, kao na primer, generisanje nekog izlaznog događaja.

Putanja prelazaka konačnog automata iz stanja u stanje, počevši od nekog inicijalnog stanja, predstavlja tragove događaja u sistemu. U modelu na prethodnoj slici, mogući tragovi su:

novčić, guranje, rotirana, novčić, guranje, rotirana, ...
žeton, žeton, žeton, novčić, guranje, rotirana, ...

Razmatrane notacije modelovanja opisuju probleme sa različitih aspekata. Jedne modeluju entitete i odnose između njih, druge tragove događaja, treće moguća stanja tokom izvršavanja, četvrte funkcije, itd. Zbog različitih pogleda na isti problem, nijedna notacija nije uspjela da se izdvoji kao dominantna. U stvari, u praksi se pokazalo da je najbolje koristiti kombinaciju više različitih notacija. Tako se došlo na ideju o formulisanju jezika za specifikaciju zahteva koji bi uključivali više notacija. Među današnjim jezicima za specifikaciju, kao najpopularniji se izdvojio UML (*Unified Modeling Language*). Osim za specifikaciju zahteva, UML se efikasno koristi i u ostalim fazama razvoja softvera (pri projektovanju, implementaciji), pa su zato njegove osnove izdvojene u poglavlju 5.

3.3 Formulisanje zahteva

Bez obzira na to koji je metod izabran za definisanje zahteva, neophodno je da zahtevi budu dobro dokumentovani. To podrazumeva dobru organizaciju zahteva, jasne tekstualne opise i preciznu prateću dokumentaciju u vidu raznih dijagrama i ilustracija. Posebna pažnja se mora posvetiti kvalitetu zahteva.

3.3.1 Dokumentovanje zahteva

Definisane zahteve koriste različiti subjekti na projektu za različite namene. Analitičari zahteva, naručioci i krajnji korisnici koriste zahteve da opišu ponašanje sistema. Za projektante, projektni zahtevi su ograničenja vezana za usvojeno rešenje. Tim za testiranje koristi zahteve da formuliše završne testove koji treba da uvere naručioca da isporučeni sistem odgovara sistemu koji je naručio. U fazi održavanja sistema, tim za održavanje održava (ispravlja greške) i unapređuje sistem (dodaje nove osobine) poštujući definisane zahteve kako se ne bi odstupilo od originalne namene sistema.

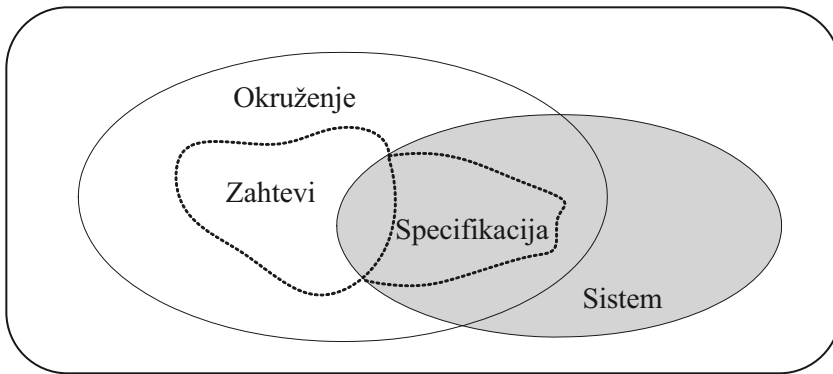
Iako dokument sa zahtevima može biti jedinstven za sve učesnike na projektu, često se izrađuju dva dokumenta:

- *definicija zahteva.* Ovaj dokument je namenjen poslovnom auditorijumu, kao što su kupci, naručioci i korisnici.
- *specifikacija zahteva.* Ovaj dokument je namenjen tehničkom auditorijumu, kao što su projektanti, timovi za testiranje, timovi za održavanje i rukovodioci projekata.

Sadržaj navedenih dokumenata će biti ilustrovan na ranije opisanom primeru softverski kontrolisane obrtno kapije na ulasku u zoološki vrt.

Definicija zahteva predstavlja kompletan spisak zahteva koje naručilac želi da budu ispunjeni. Ona nastaje kao rezultat zajedničkih napora analitičara zahteva i naručioca/korisnika. U dokumentu su, osim zahteva, opisani i entiteti iz okruženja u kome će sistem biti instaliran, kao i ograničenja vezana za te entitete. Pošto zahtevi treba da budu realizovani u praksi, vrlo je bitna njihova povezanost, odnosno interakcija sa okruženjem. U razmatranom primeru, mogu se postaviti dva zahteva: 1) niko ne može da uđe u zoološki vrt bez plaćanja ulaza i 2) sistem ne može da spreči onoga ko je platio ulaz da uđe. Logičnija formulacija drugog zahteva bi bila da svakome ko plati kartu treba obezbediti da uđe u zoološki vrt, ali ovaj zahtev ne bi bilo moguće realizovati. Naime, ne postoji način da sistem spreči neki spoljašnji faktor da onemogući posetioca koji je platio kartu da uđe u vrt. Na primer, može da se desi da jedan posetilac ubaci novčić, a drugi ga gurne i on uđe u vrt, ili posetilac može da ubaci novčić i onda se predomisli i ode na drugu stranu.

Specifikacija zahteva sadrži zahteve o ponašanju softverskog sistema. Ona, takođe, uzima u obzir i okruženje u kome će sistem raditi, s tim što se referiše samo na entitete iz okruženja kojima može da se pristupi preko odgovarajućih interfejsa iz sistema. Ovaj dokument piše analitičar, a koriste ga svi učesnici u razvoju softvera. Analitičar mora da uspostavi jednoznačnu vezu između svakog zahteva navedenog u definiciji zahteva i zahteva u specifikacionom dokumentu. Pri tome ne sme da dođe do gubitka informacije ili njene izmene prilikom pretvaranja definicionog zahteva u specifikaciju. Na slici 3.6 prikazan je odnos opisanih dokumenata. Eliptički oblici predstavljaju okruženje i sistem. Oni imaju zajednički interfejs koji se nalazi u preseku elipsi. Zahtevi mogu biti definisani bilo gde u domenu okruženja, uključujući i interfejs sistema, dok se specifikacija ograničenja nalazi uvek u preseku domena okruženja i domena sistema.



Slika 3.6 Formulisanje zahteva

U primeru obrtno kapije, prvi zahtev je bio da niko ne može da uđe u zoološki vrt bez plaćene ulaznice. Pošto se na kapiji nalazi otvor za ubacivanje novčića, sigurno se može detektovati da li je odgovarajući apoen ubačen. Međutim, sam događaj ulaska u vrt (ko će ući) ne može da bude kontrolisan od strane sistema. Stoga, prvi zahtev iz definicije zahteva mora se pretočiti u nešto drugačiji zahtev u specifikaciji zahteva, koji, opet, mora da odslikava suštinu polaznog zahteva. Sistem može da realizuje zahteve isključivo na osnovu događaja koje kapija može da detektuje, a to su: da li je kapija otključana ili zaključana, i da li je posetilac gurnuo kapiju ili nije. Stoga, prvi zahtev u definiciji zahteva u specifikaciji zahteva postaje:

Kada posetilac gurne otključanu obrtnu kapiju, ona se automatski rotira za jedan polukrug nakon čega se sama zaključava.

Ovakva specifikacija predstavlja opis originalnog zahteva koji se može realizovati u sistemu.

Radi lakše izrade navedenih dokumenata, u nastavku je dat predlog njihovog sadržaja.

Pri definisanju zahteva treba uraditi sledeće:

- skicirati opštu namenu i opseg sistema, uključujući relevantne ciljeve, veze sa drugim sistemima, uz uvođenje terminologije, oznaka, skraćenica i sl.
- navesti razloge za razvoj sistema (zašto je postojeći sistem nezadovoljavajući, pa treba razvijati novi, koje delove postojećeg sistema treba zameniti, a koje ne i dr.)

- opisati osobine rešenja koje se predlaže kroz prikaz njegovih osnovnih funkcionalnosti na nivou slučajeva korišćenja; opis obuhvata i analizu u pogledu kvaliteta rešenja (tačnost, brzina, prioriteti i sl.)
- opisati okruženje u kome će sistem da radi (navođenje svih hardverskih i softverskih komponentata sa kojima će sistem biti u interakciji, analiza korisničkog interfejsa prema sposobnostima korisnika, njihovom iskustvu, znanju, obrazovanju i dr.)
- navesti pretpostavke vezane za ponašanje okruženja, ukoliko one postoje (uslovi koji mogu dovesti do otkaza sistema, kao i eventualne promene u okruženju koje mogu dovesti do promena zahteva)

Pri izradi specifikacije zahteva potrebno je uraditi sledeće:

- detaljno opisati sve ulaze i izlaze (interfejs sistema), uključujući izvore ulaza, odredišta izlaza, dozvoljene opsege i formate ulaznih i izlaznih veličina, protokole razmene podataka, vremenska ograničenja, i sl.
- prikazati zahtevanu funkcionalnost pomoću ulaza i izlaza korisničkog interfejsa, uključujući provere ispravnosti ulaznih i izlaznih veličina; prikaz mora da bude potpun, odnosno, za sve moguće vrednosti na ulazu, mora biti poznata vrednost na izlazu; ovde se mogu koristiti različite tehnike modelovanja (konačni automati, tragovi događaja i dr.)
- utvrditi usklađenost svakog zahteva sa kriterijumom koji naručilac postavlja u pogledu kvaliteta

Rezultat izrade dokumenata sa definicijom i specifikacijom zahteva je detaljan opis sistema koji razvojni tim treba da proizvede. Neke organizacije, kao što su IEEE (*Institute of Electrical and Electronic Engineers*) i Ministarstvo odbrane SAD poseduju standarde koji definišu sadržaj i format dokumenata o zahtevima. Na slici 3.7 dat je primer IEEE standarda za specifikaciju softverskih zahteva.

1. Uvod u dokument
 - 1.1. Namena proizvoda
 - 1.2. Opseg proizvoda
 - 1.3. Akronimi, skraćenice, definicije
 - 1.4. Reference
2. Opšti opis proizvoda
 - 2.1. Kontekst proizvoda
 - 2.2. Funkcije proizvoda
 - 2.3. Karakteristike korisnika
 - 2.4. Ograničenja
 - 2.5. Pretpostavke i zavisnosti
3. Specifični zahtevi
 - 3.1. Zahtevi spoljašnjih interfejsa
 - 3.1.1. Korisnički interfejsi
 - 3.1.2. Hardverski interfejsi
 - 3.1.3. Softverski interfejsi
 - 3.1.4. Komunikacioni interfejsi
 - 3.2. Funkcionalni zahtevi
 - 3.2.1. Klasa 1
 - 3.2.2. Klasa 2
 - ...
 - 3.3. Zahtevi u pogledu performansi
 - 3.4. Projektna ograničenja
 - 3.5. Zahtevi u pogledu kvaliteta
 - 3.6. Ostali zahtevi
4. Dodaci

Slika 3.7 IEEE standard za specifikaciju zahteva

3.3.2 Kvalitet zahteva

Pošto se prilikom izrade softvera realizuju samo oni zahtevi koji su navedeni u specifikaciji zahteva, vrlo je važan njihov kvalitet. Procena kvaliteta zahteva zasniva se na davanju odgovora na sledeća pitanja:

- *Da li su zahtevi ispravni?* Nakon dokumentovanja zahteva, analitičar i naručilac treba još jednom detaljno da pregledaju zahteve kako bi se uverili da su oni usklađeni sa stvarnim potrebama i odgovaraju shvatanjima naručioca.
- *Da li su zahtevi dosledni?* Doslednim se smatraju dva zahteva koja mogu biti istovremeno ispunjena. U suprotnom, zahtevi su nedosledni. Na primer, ako se u jednom zahtevu navodi da sistem može da koristi najviše deset korisnika u jednom trenutku, a u drugom zahtevu da postoje situacije u kojima sistem može da ima i dvadeset korisnika, onda su ova dva zahteva nedosledna.
- *Da li su zahtevi nedvosmisleni?* Ako više osoba koje analiziraju dokument sa zahtevima dođe do istih interpretacija, tj. shvatanja nekog zahteva, onda je on nedvosmislen. Međutim, može se desiti i da različite osobe ispravnim razmišljanjem dođu do različitih interpretacija zahteva. Tada je zahtev neprihvatljiv jer je višeznačan. U tom slučaju, potrebno je obazbediti nove informacije kako bi zahtev bio precizniji. Na primer, zahtev može da kaže da će neki sklop raditi ako je odstupanje njegove osovine od osnovnog položaja malo. Pojam „malo“ se može različito shvatiti. Stoga je potrebna dodatna informacija koja preciznije definiše moguće odstupanje (na pr. 0,5mm).
- *Da li su zahtevi kompletni?* Skup zahteva se smatra kompletnim ako odlikava zahtevano ponašanje sistema za sve moguće kombinacije ulaza u svim mogućim stanjima sistema, poštujući sva ograničenja. Na primer, u slučaju sistema za obračun plata, potrebno je definisati šta se događa ako radnik uzme neplaćeno odsustvo, dobije povišicu, ili zatraži akontaciju plate.
- *Da li su zahtevi izvodljivi?* Ponekad se dešava da rešenje koje bi odgovorilo potrebama naručioca u suštini i ne postoji. To se obično dešava kada naručilac postavlja dva ili više zahteva u pogledu kvaliteta. Na primer, on može da zahteva jeftin sistem koji analizira ogromne količine podataka za vrlo kratko vreme.
- *Da li je svaki zahtev relevantan?* Ponekad zahtevi nepotrebno otežavaju posao razvojnom timu, pa razvojni tim mnogo vremena troši na realizaciju

nebitnih aspekata sistema, umesto da se usredsredi na suštinsku funkcionalnost. Na primer, ukoliko se razvija simulator tenka, može se zahtevati da se posadi tenka omogućući da šalje elektronsku poštu vojnicima, iako je osnovna namena tenka da savlada neravan teren. Stoga je potrebno da se prilikom definisanja zahteva spreči „eksplozija mogućnosti“ (uvek možemo dodavati nešto novo), kako bi se pomoglo zainteresovanim subjektima da izdvoje suštinske i poželjne zahteve.

- *Da li je zahteve moguće testirati?* Pojedini zahtevi se mogu testirati pre realizacije softvera. To se radi pravljenjem odgovarajućih testova koji jasno demonstriraju kako bi zahtev bio zadovoljen u konačnoj verziji softvera, tj. u realnim uslovima. Na primer, može se napraviti test kojim bi se ispitalo koliko korisnika može istovremeno da radi na sistemu.
- *Da li zahtevi mogu da se prate?* U dokumentima, zahtevi moraju da budu dobro organizovani i sistematizovani, sa jedinstvenim oznakama u cilju lakšeg referenciranja na njih.

Odgovori na navedena pitanja se mogu, takođe, posmatrati kao neka vrsta pratećih zahteva u pogledu kvaliteta. Oni mogu da pomognu analitičaru zahteva da odredi kada je zahtev dobro definisan, a kada je potrebno prikupiti još informacija o njemu. Step en do koga su ovi zahtevi zadovoljeni utiče na sveobuhvatnost rešenja, izbor jezika za specifikaciju zahteva, kao i na proces validacije i verifikacije.

3.4 Validacija i verifikacija zahteva

Dokumenti o zahtevima detaljno opisuju softverski proizvod koji treba da bude isporučen naručiocu. Oni su vrlo bitni, kako za naručioca, tako i za buduće projektante. Pre nego što analitičar zahteva prosledi ove dokumente projektantima, neophodno je da još jednom, zajedno sa naručiocem, proveri njihov sadržaj. Ta provera obuhvata validaciju zahteva i verifikaciju specifikacije. Validacijom se dokazuje da se razvija pravi sistem (onaj koji je potreban naručiocu), dok se verifikacijom potvrđuje da se sistem razvija na pravi način.

3.4.1 Validacija zahteva

Validacija zahteva podrazumeva proveru da li definicije zahteva tačno odražavaju zahteve naručioca. Ovo je složen proces u kome nije dovoljno samo

reći da li je zahtev ispravan ili ne, već treba dati i odgovarajuću argumentaciju za tu tvrdnju.

Validacija zahteva se sprovodi po sledećim kriterijumima:

- ispravnost zahteva
- doslednost zahteva
- nedvosmislenost zahteva
- potpunost zahteva
- relevantnost zahteva
- mogućnost testiranja zahteva
- mogućnost praćenja zahteva

Većina navedenih kriterijuma, kao što su ispravnost, relevantnost, potpunost ili nedvosmislenost zahteva, podležu subjektivnoj proveru. Proveru zahteva po ovim kriterijumima mogu da urade samo zainteresovani subjekti, ako im se da na uvid odgovarajući dokument. Provere po kriterijumima, kao što su doslednost ili mogućnost praćenja zahteva, drugačije su prirode i one se mogu automatizovati.

Postoje različite tehnike za sprovođenje validacije zahteva. U najjednostavnijem slučaju, validacija se može svesti na *čitanje dokumenata* i formiranje izveštaja o uočenim greškama.

Drugi način validacije je *prolazak kroz dokumenta* tako što jedan od autora dokumenata izlaže zahteve svim zainteresovanim subjektima i od njih traži mišljenje i komentare. Ovaj metod je efikasan ako postoji veliki broj zainteresovanih subjekata, pa bi bilo nepraktično da svaki od njih detaljno analizira dokumente.

Formalna inspekcija je postupak validacije u kome se revizori stavljaju u određene uloge (na pr. prodavac, računovođa i dr.) i prate propisana pravila.

Čest način validacije je *ocenjivanje zahteva*. Ocenjivanje se radi po raznim aspektima, kao što su ocenjivanje postavljenih ciljeva, namene softvera, okruženja, predloženih funkcija, rizika, itd. Ocenjivanje rade kako predstavnici naručioca, tako i oni koji su realizovali softver. U predstavnike naručioca spadaju krajnji korisnici, oni koji će obezbediti ulazne podatke za sistem, kao i oni koji će koristiti izlazne rezultate. Ovde mogu biti uključeni i rukovodioci. S druge strane, ocenjivanje rade i članovi razvojnog tima, tima za testiranje i operativnog tima koji se bavi održavanjem.

Izbor validacione tehnike koja će biti primenjena zavisi od notacija koje su korišćene pri definisanju zahteva, kao i od afiniteta i iskustava zainteresovanih subjekata koji rade validaciju.

Kada se tokom validacije uoči neki problem, on se najpre dokumentuje (navede se u čemu je problem i šta mu je uzrok), a zatim se prosledi analitičaru zahteva. Analitičar ima zadatak da reši problem. Na primer, validacijom se može otkriti da postoji kontradiktornost zahteva u smislu da naručilac zahteva pouzdanost i raspoloživost softvera za koje projektni tim smatra da je nemoguće postići. Ovakvi zahtevi moraju da budu razrešeni pre nego što počne projektovanje.

3.4.2 Verifikacija specifikacije

Verifikacija u opštem smislu predstavlja proveru da li je jedan element u projektu saglasan sa drugim. Na primer, može se proveriti da li je programski kôd saglasan sa projektnim rešenjem, ili da li je projektno rešenje u saglasnosti sa specifikacijom zahteva.

U kontekstu projektnih zahteva, verifikacija se radi u cilju provere da li specifikacija zahteva odgovara definiciji zahteva. To je vrlo bitno zato što projektanti razvijaju sistem na osnovu dobijene specifikacije. Verifikacija pruža sigurnost da će sistem koji je realizovan prema datoj specifikaciji u potpunosti zadovoljiti zahteve korisnika.

Verifikaciju nije jednostavno uraditi. Treba dokazati da specifikacija obuhvata sve funkcije, događaje, aktivnosti i ograničenja koji su izneti u zahtevima. Sama specifikacija retko može da pruži dovoljno argumenata za ovo dokazivanje, jer je ona fokusirana na sistemski interfejs, dok je često potrebno dokazati nešto iz okruženja, što je van ovog interfejsa. Na primer, specifikacija sadrži podatak o sili kojom treba delovati na obrtnu rampu da bi ona napravila polukrug i dozvolila posetiocu da uđe u zoološki vrt, a potrebno je da se prebroji koliko je ulazaka u vrt bilo. Ovaj nesklad se prevazilazi tako što se, osim specifikacije, uzimaju u obzir i pretpostavke o ponašanju okruženja, pa se onda proveravaju zahtevi naručioca.

Za verifikaciju se često koriste specijalizovani programi koji pretražuju prostor izvršavanja specifikacije. Ovi programi su prilično složeni i troše značajne računarske resurse.

Nakon obavljene validacije i verifikacije projektnih zahteva, trebalo bi da i naručilac i projektant budu zadovoljni formulisanim zahtevima, tako da projektant može da nastavi sa projektovanjem sistema.

4 Projektovanje sistema

Rezultat analize zahteva, koju analitičar zahteva sprovodi u saradnji sa naručiocima i korisnicima softverskog sistema, predstavljaju dva dokumenta: definicija zahteva i specifikacija zahteva. Kao što je rečeno u prethodnom poglavlju, specifikacija zahteva je dokument namenjen projektantima sistema i u njemu je problem opisan korišćenjem tehničke terminologije bliske projektantima. Ovaj dokument predstavlja osnovu za projektovanje sistema, čiji je cilj generisanje rešenja koje zadovoljava potrebe naručioca. Dakle, pod projektovanjem sistema se podrazumeva kreativni proces prevođenja datog problema u njegovo rešenje.

Iako je problem precizno definisan specifikacijom, broj mogućih rešenja je obično vrlo veliki (ponekad i neograničen). Svaki projektant rešava problem na svoj način i dobija drugačije rešenje. Zajedničko za sva rešenja je da zadovoljavaju sve postavljene zahteve korisnika. Rešenja se međusobno razlikuju po aspektima problema kojima posvećuju veću pažnju. Priroda rešenja se može menjati u svim fazama projekta, ukoliko za to postoji realna potreba. Dešava se da naručioci u toku projekta uoče nešto što bi im bilo bitno da imaju u konačnoj verziji sistema (bilo da su to zaboravili ranije, ili da su u međuvremenu došli do saznanja da bi to bilo korisno dodati), pa imaju potrebu da promene zahteve. Tada bi im trebalo izaći u susret, jer nije dobro da dobiju softver koji im ne odgovara.

Da bi zahteve pretočili u sistem koji radi, projektanti moraju da zadovolje želje i naručioca i tima koji implementira softver. S jedne strane, naručilac zna šta sistem treba da radi, dok sa druge strane, tim koji pravi softver zna kako sistem to treba da radi. Imajući ovo u vidu, projektovanje se može shvatiti kao iterativni proces koji se sastoji iz dve celine: konceptualnog projekta i tehničkog projekta.

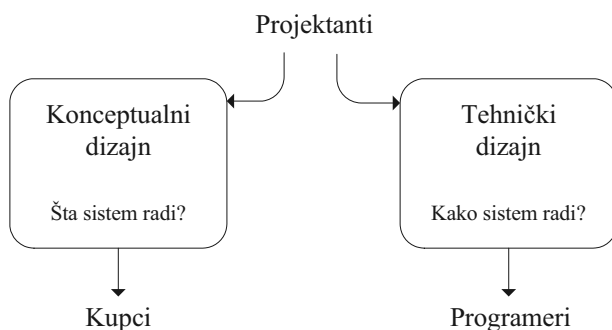
Konceptualni projekat ili konceptualni dizajn detaljno opisuje klijentu šta će sistem da radi. On identifikuje sve entitete u sistemu, njihove atribute i međusobne veze, i pruža odgovore na sledeća pitanja:

- Odakle i na koji način se dobijaju ulazni podaci?

- Kako se obrađuju ulazni podaci?
- Kako će izgledati korisnički interfejs?
- Kakav je vremenski tok događaja u sistemu?
- U kom obliku će biti predstavljeni rezultati?

Konceptualni projekat opisuje funkcije sistema na jeziku razumljivom klijentu i ne zavisi od načina implementacije sistema. Na primer, u njemu se može reći da se neke poruke šalju sa jednog mesta na drugo, ali se ne objašnjava koji će mrežni protokol biti korišćen prilikom tog slanja.

Kada klijent odobri konceptualni projekat, sledi njegovo prevođenje u mnogo detaljniji, tehnički projekat. *Tehnički projekat* ili tehnički dizajn omogućava timu koji razvija softver da utvrdi koji hardver i koji softver su potrebni za implementaciju rešenja, kako će se obavljati komunikacija u sistemu, definiše ulaze i izlaze sistema, opisuje mrežnu arhitekturu sistema i dr. Odnos konceptualnog i tehničkog dizajna prikazan je na slici 4.1.



Slika 4.1 Opšti prikaz procesa projektovanja

Konceptualni i tehnički projekat se mogu objediniti u jedan, sveobuhvatan dokument. To se radi u slučaju kada su klijenti profesionalci u datoj oblasti i mogu da razumeju i „šta sistem radi“ i „kako sistem radi“.

Proces projektovanja je iterativan zato što projektanti naizmenično rade na analizi zahteva, predlaganju mogućih rešenja, testiranju izvodljivosti pojedinih aspekata rešenja i dokumentovanju rešenja za programere.

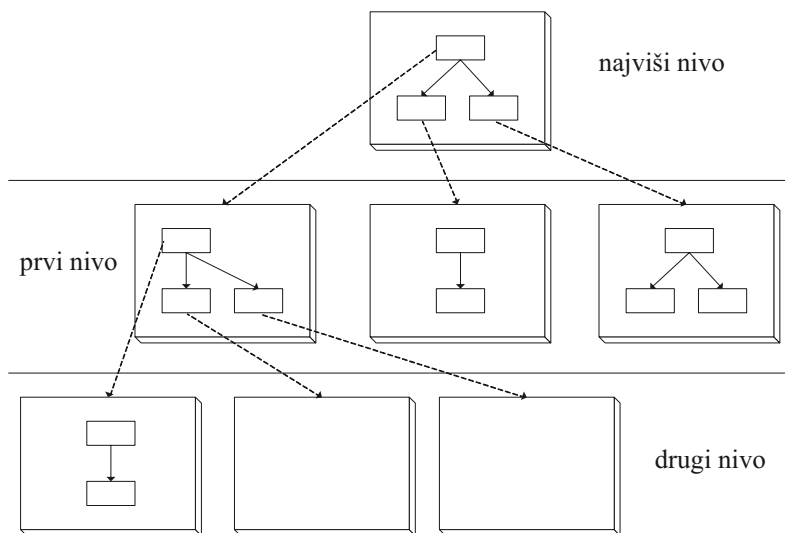
4.1 Modularnost u projektovanju

Proces projektovanja podrazumeva utvrđivanje skupa komponenata od kojih će sistem biti napravljen, kao i definisanje veza koje treba da postoje između njih. Postoji mnogo načina da se napravi dobar projekat koji zadovoljava zahteve korisnika. Izbor projekta koji će biti implementiran zavisi od različitih faktora, kao što su: sklonosti projekatnata, zahtevana struktura sistema, raspoloživ skup ulaznih podataka i dr. Bez obzira na to koji će projekat biti izabran za realizaciju, u svakom projektu postoji neki oblik modularnosti.

Wasserman je 1995.godine predložio da se proces projektovanja izvodi na jedan od sledećih načina:

- *modularnost na nivou funkcionalnosti*. Ovaj metod se zasniva na tome da se izdvoje komponente u sistemu tako da svaka ima svoju funkcionalnost, a da sve zajedno u potpunosti funkcionalno opisuju sistem. Projektant polazi od opšteg opisa funkcionalnosti svake komponente, a zatim ga detaljno razrađuje. Za svaku komponentu, projektant utvrđuje šta ona sadrži, kakva joj je organizacija, u kakvoj je vezi sa ostalim komponentama i sl.
- *modularnost na nivou podataka*. Ovaj pristup se zasniva na strukturama podataka. Projektant na opštem nivou definiše globalnu strukturu podataka, a zatim navodi kako će podaci biti korišćeni i kakve veze postoje među njima.
- *modularnost na nivou događaja*. U ovom pristupu, projektant uočava moguće događaje u sistemu, a zatim analizira kako oni menjaju stanje sistema. Najpre se formira opšti opis koji sadrži skup mogućih stanja, nakon čega se daje detaljan opis kako se prelazi iz jednog stanja u drugo.
- *projektovanje „spolja ka unutra“*. Ovaj metod se zasniva na podacima koje korisnik unosi u sistem. Projektant najpre formira opšti opis u kome navodi koje sve podatke korisnik može da unese u sistem, a zatim detaljno objašnjava šta se u sistemu dešava sa unetim podacima, uključujući kako međurezultate, tako i konačne rezultate.
- *objektno-orijentisano projektovanje*. Ova vrsta projektovanja podrazumeva definisanje klasa objekata u sistemu i njihovih međusobnih veza. Najpre se na opštem nivou opisuju tipovi objekata, a zatim se detaljno daju njihova svojstva i akcije koje mogu da izvrše.

Bez obzira na korišćeni pristup, kao rezultat projektovanja dobija se hijerarhija komponenata ili modula sa više nivoa, kao što je prikazano na slici 4.2.



Slika 4.2 Nivoi modularnosti

Svaki sledeći nivo sadrži više informacija od prethodnog. Za sistem se kaže da je modularan ukoliko svaku aktivnost u njemu obavlja jedna komponenta sa dobro definisanim ulazima, izlazima i osobinama. Ulazi su dobro definisani ako svi imaju uticaja na funkcionisanje komponente, tj. ne postoje ulazi koji se ne koriste. Takođe, ne sme se dozvoliti ni da neki od ulaza nedostaje, jer onda komponenta ne bi bila u stanju da obavi svoju funkciju u celosti. Izlazi su dobro definisani samo ako se svi mogu proizvesti nekom od akcija sistema.

Modularnost pruža mogućnost da se na višem nivou apstrakcije budućim klijentima, nabavljačima, podizvođačima prikaže dizajn sistema, a istovremeno sadrži i detalje potrebne timu koji razvija sistem.

4.2 Strategije projektovanja

Proces projektovanja obično počinje posmatranjem sistema odozgo, tj. sa opšteg aspekta, a zatim se ide na dole uvođenjem sve više detalja koji se odnose na izgled i funkcionisanje sistema. Obrnuti način projektovanja, odozdo na gore, je znatno teži i neizvesniji i često dovodi do nerešivih problema. Shaw i Garlan su 1996.godine predložili da se projektovanje obavlja na tri nivoa:

- projektovanje arhitekture sistema
- projektovanje programskog kôda

- završno projektovanje

Arhitektura sistema podrazumeva povezivanje mogućnosti sistema koje su navedene u specifikaciji zahteva sa komponentama sistema koje će biti implementirane. Komponente obično predstavljaju pojedinačne module u sistemu. Osim modularne strukture, arhitektura opisuje i veze koje treba da postoje između modula, kao i načine kako se od manjih podsistema generišu složeniji sistemi bitni za formiranje konačnog softverskog proizvoda.

Projektovanje programskog kôda obuhvata izbor programskog jezika koji će biti korišćen u implementaciji, izbor struktura podataka u kojima će se čuvati relevantni podaci, kao i izbor algoritama za obradu i manipulaciju podacima. Takođe se utvrđuje skup programskih modula, tj. datoteka pomoću kojih će sistem biti implementiran, potrebne procedure i funkcije sistema.

Završno projektovanje još detaljnije opisuje implementaciju navođenjem informacija o formatima podataka koji će biti korišćeni, šablonima, primenjenim protokolima, dodeli memorije, itd.

Iako bi bilo korisno projektovati po nivoima u redosledu u kome su oni navedeni, iskustva pokazuju da u praksi projektanti naizmenično prelaze sa jednog nivoa na drugi kako više upoznaju rešenje i njegove posledice. Na primer, grupa projekatanta može da odluči da bi sistemom trebalo upravljati pomoću tabela. Međutim, kada naprave prototipove tabela, mogu da zaključe da sistem nema potrebno vreme odziva, tj. da radi presporo. Zato ponovo projektuju brži sistem kojim bi se upravljalo pomoću matrica umesto tabela. Na ovaj način, oni prelaze sa prvog na drugi, a onda se ponovo vraćaju na prvi nivo. Slično, dok projektanti istražuju neke aspekte sistema, mogu u komunikaciji sa programerima ili onima koji testiraju sistem da dođu do nekih zaključaka koji ukazuju na to da bi trebalo promeniti dizajn kako bi se unapredili implementacija, mogućnosti testiranja ili održavanje. Prema tome, projektanti moraju postepeno da rade na arhitekturi, programima i završnom dizajnu u skladu sa sopstvenim poznavanjem rešenja i svojom kreativnošću.

Na početku razvoja sistema potrebno je izabrati arhitekturu koja će biti primenjena. Mnogi sistemi imaju sličnu strukturu. Na primer, distribuirani sistemi obično imaju klijent-server strukturu u kojoj klijent upućuje upite, a server procesira te upite i odgovara na njih. Dosadašnja iskustva po pitanju arhitekture sistema dovela su do pojave različitih stilova u projektovanju. Stil podrazumeva strukturalnu organizaciju softverskog sistema. On uključuje izbor komponentata (podsistema), kao i definisanje njihovih uloga. Na primer, u klijent-server arhitekturi razlikuju se dva podsistema: klijent (kojih može biti više) i server (koji je jedinstven). Uloga klijenta može biti prikazivanje korisničkog interfejsa korisniku, a uloga servera procesiranje brojnih zahteva i zaštita podataka važnih za korisnika. Stil, takođe, odražava i veze između podsistema, tj. način njihovog

povezivanja, uz eventualna ograničenja. Na primer, veza između klijenta i servera se ostvaruje tako što klijent postavlja pitanja, a server odgovara na njih.

Nekoliko najčešće korišćenih stilova su:

- cevi i filtri (*pipe-filter*)
- slojevita (*layers*) arhitektura
- klijent-server (*client-server*) arhitektura
- ravnopravan (*peer-to-peer*) pristup
- arhitektura zasnovana na događajima (*event-bus*)
- objektno-orijentisani (*object-oriented*) pristup

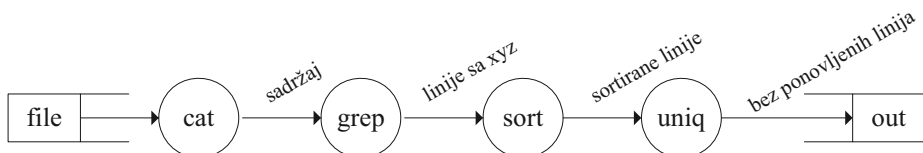
4.2.1 Cevi i filtri

Cevi i filtri predstavljaju strukturu koja se koristi u sistemima baziranim na tokovima podataka. Svaki korak u postupku procesiranja podataka izdvojen je u jednu filtersku komponentu. Filtri rade nezavisno i nisu svesni postojanja drugih filtera u sistemu. Podaci prolaze kroz cevi koje povezuju filtre. Cevi se mogu koristiti za memorisanje ili za sinhronizaciju.

Primer dizajna koji se zasniva na ovom stilu su komande u *Unix* operativnom sistemu:

```
cat file | grep xyz | sort | uniq > out
```

Zadatak u vidu izvođenja niza komandi podeljen je u više uzastopnih koraka. Koraci su povezani tokovima podataka tako da izlaz jednog koraka predstavlja ulaz narednog. U datom primeru, *cat* filter čita datoteku *file* i prosleđuje njen sadržaj *grep* filtru. Filter *grep* selektuje linije koje sadrže *xyz* i šalje ih *sort* filtru. Ovaj filter sortira linije i prosleđuje ih *uniq* filtru koji briše duple linije i šalje rezultat na izlaz *out*. Opisani postupak prikazan je slici 4.3, gde krugovi predstavljaju filtre, a linije sa strelicama cevi.



Slika 4.3 Primer arhitekture sa filtrima i cevima

Arhitektura sa cevima i filtrima ima prednosti, ali i nedostatke. Prednosti se ogledaju u tome što se sistem lako modifikuje dodavanjem novih filtara, ili uklanjanjem postojećih. Filtri predstavljaju nezavisne komponente koje se lako samostalno razvijaju. Dalje, filtri se mogu koristiti više puta, tj. moguće je generisati različite tokove kombinovanjem datog skupa filtara. Ovaj stil omogućava konkurentno procesiranje zato što filtri rade nezavisno i obavljaju svoju funkciju kada prime potrebne podatke, ne čekajući da se svi podaci učitaju ili obrade drugim filtrima. Značajna prednost je i u jednostavnoj analizi ponašanja ovakvih sistema. Na primer, ako je ulaz u sistem x , a ponašanje prvog filtra se može opisati funkcijom g , ponašanje drugog filtra funkcijom f , rezultat procesiranja se može opisati pomoću

$$f(g(x)).$$

Na osnovu ovog izraza, može se analizirati propusna moć sistema (koja je određena najsporijim filtrom), kao i mogućnost pojave nerešivih situacija (*deadlocks*). Ove situacije se mogu desiti ukoliko neki filter, da bi generisao izlaz, mora da ima sve procesirane podatke, a nema dovoljan bafer da ih prihvati. Primer ovakvog filtra je *sort Unix* filter.

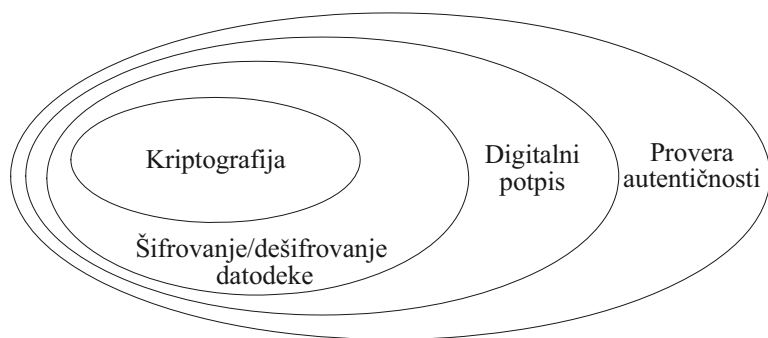
Glavni nedostatak arhitekture sa cevima i filtrima je u tome što podstiče paketnu obradu i nije pogodna za interaktivne aplikacije. Drugi nedostatak je nepotrebn gubitak vremena u transformacijama podataka. Na primer, dešava se da filter procesira realne brojeve, a ulaz i izlaz su mu u tekstualnom formatu. Treći nedostatak je taj što se može desiti da filtri ponavljaju pripremne funkcije drugih filtara, što utiče na pad performansi i povećanje stepena složenosti sistema. Na primer, pošto su realizovani nezavisno, svi filtri u sistemu koji obrađuje datumski objekat, moraju da provere ispravnost datuma, tj. da li je mesec u opsegu od 1 do 12, a dan od 1 do 31, tako da se ova provera nepotrebno vrši više puta.

4.2.2 Slojevita arhitektura

Slojevita arhitektura dekomponuje aplikaciju u više apstraktnih nivoa. Svaki nivo predstavlja jedan sloj koji sadrži grupu zadataka. Slojevi su hijerarhijski raspoređeni tako da svaki sloj pruža usluge sledećem višem sloju u hijerarhiji. Takođe, usluge u jednom sloju implementiraju se korišćenjem usluga koje pruža prvi niži sloj u odnosu na posmatrani.

Ideja iz koje je nastao ovaj stil zasniva se na tome da se konceptualno različiti delovi aplikacije implementiraju odvojeno, s tim da slojevi na višim nivoima apstrakcije koriste samo usluge nižih slojeva.

Primer sistema sa ovom arhitekturom je sistem za bezbednost datoteka dat na slici 4.4. Ovaj sistem se sastoji iz četiri sloja. Najniži sloj je kriptografija koji sadrži funkcije za šifrovanje/dešifrovanje. Drugi sloj koristi ključ za šifrovanje/dešifrovanje datoteke. Treći sloj je zadužen za generisanje digitalnog potpisa i davanje prava pristupa datoteci, dok četvrti sloj identifikuje korisnika putem korisničkog imena i lozinke i proverava autentičnost.



Slika 4.4 Primer slojevite arhitekture

Prednosti slojevite arhitekture su:

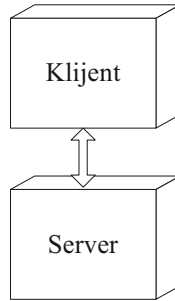
- niži slojevi mogu biti više puta korišćeni od strane različitih viših slojeva
- postojanje slojeva čini lakšom standardizaciju zadataka
- razvoj i testiranje jednog sloja se obavljaju nezavisno od drugih slojeva, pošto se veze (interfejsi) između slojeva ne menjaju, već se sve promene obavljaju unutar sloja

4.2.3 Klijent-server arhitektura

U klijent-server arhitekturi, serverska komponenta pruža usluge većem broju klijentskih komponenata. Klijentske komponente zahtevaju usluge od servera. Serveri su stalno aktivni i osluškiju da li ima zahteva od klijenata. Zahtevi se šalju komunikacionim kanalima koji zavise od mašina na kojima se server i klijent nalaze. Opšti izgled ove arhitekture prikazan je na slici 4.5.

Tipični primeri klijent-server arhitekture su aplikacije sa udaljenim pristupom bazama podataka (kada klijentska aplikacija zahteva uslugu od servera baze podataka), udaljeni fajl sistemi (klijentska aplikacija pristupa datotekama na

serveru i u lokalu transparentno), ili web aplikacije (čitači zahtevaju podatke od web servera).



Slika 4.5 Klijent-server arhitektura

Prispeli zahtevi se obično opslužuju u odvojenim nitima na serveru. U komunikaciji često ima „praznog hoda“ koji nastaje kako zbog saobraćaja na mreži, tako i zbog neophodnog transformisanja zahteva i rezultata u formate koji su često različiti na serverskoj i klijentskog strani. U distribuiranim sistemima sa više servera koji obavljaju istu funkciju, mora se obezbediti transparentnost, što znači da klijenti ne treba da razlikuju servere. Na primer, ako se u Google pretraživaču zahteva neki podatak unošenjem URL adrese, klijent ne treba da zna na kojoj tačno mašini je taj podatak lociran, koja je platforma primenjena, ili koja je putanja korišćena.

4.2.4 Ravnopravni pristup

Arhitektura koja se zasniva na ravnopravnom pristupu može se shvatiti kao simetrična klijent-server arhitektura. U ovoj arhitekturi, ista komponenta može da radi i kao klijent (zahteva usluge od drugih komponentata) i kao server (pruža usluge drugim komponentama). Takođe, komponenta može i da zadrži samo jednu funkcionalnost, bilo klijentsku, ili serversku. Osim toga, komponenta može dinamički da menja svoju ulogu između navedene tri.

Prednost ove arhitekture je u tome što komponente mogu da koriste ne samo svoje, već i kapacitete kompletne mreže u kojoj se nalaze. Takođe, administriranje je jednostavnije, zato što su ovakve, „ravnopravne“ mreže samoorganizujuće. Sistem koji podržava ovu arhitekturu je skalabilan i otporan na otkaze pojedinih komponentata. Konfiguracija sistema se može menjati (uključivanje i isključivanje komponentata) dinamički u toku rada.

Nedostatak ove arhitekture je u tome što nema garancije u pogledu kvaliteta pruženih usluga, pošto komponente sarađuju na dobrovoljnoj osnovi. Zbog ovoga,

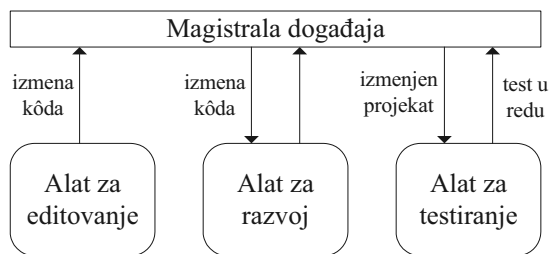
teško je obezbediti sigurnost mreže. Performanse ovakvih sistema rastu sa povećanjem broja komponentata, ali i opadaju sa njegovim smanjenjem.

4.2.5 Arhitektura zasnovana na događajima

Model sistema koji se zasniva na događajima radi na principu difuzionog emitovanja poruka. Sistem funkcioniše tako što komponente (izvori događaja) šalju poruke o tome da je nastupio neki događaj na magistralu događaja. Ostale komponente koje su priključene na magistralu mogu događajima da pridružuju procedure i taj proces se naziva registrovanjem procedura. Zatim sistem poziva sve registrovane procedure.

Generisanje poruka i pridruživanje procedura su asinhronog karaktera. Nakon što neka komponenta generiše poruku o događaju, ona prelazi na neki drugi posao, ne čekajući da ostale komponente prime poruku.

Arhitektura zasnovana na događajima koristi se u aplikacijama za monitorisanje, trgovinu, u okruženjima za razvoj softvera (videti sliku 4.6) i dr.



Slika 4.6 Primer arhitekture zasnovane na događajima

4.2.6 Objektno-orijentisani pristup

Veliki broj savremenih sistema je razvijen delimično, ili u potpunosti na osnovu objektno-orijentisanog (OO) pristupa. Po ovom pristupu, problem i njegovo rešenje se organizuju kao skup objekata kojima se opisuju ne samo podaci, već i ponašanja. OO reprezentacija se prepoznaje po sledećim karakteristikama:

- *identitet*. Proističe iz činjenice da su podaci organizovani u diskretne celine koje se nazivaju objektima. Objekat ima svoja stanja i ponašanja (na primer, ako je objekat *brana*, stanja mogu biti *potpuno_otvorena* ili *potpuno_zatvorena*, dok ponašanje može biti *zvučno_upozorenje* kada branu treba otvoriti). Objekat ima ime koje ga identifikuje.

- *apstrakcija*. Predstavlja različite aspekte sistema. Skup apstrakcija formira hijerarhiju koja prikazuje međusobne odnose različitih pogleda na sistem.
- *klasifikacija*. Podrazumeva grupisanje objekata sa zajedničkim svojstvima i ponašanjem. Jedna grupa se može smatrati klasom kojoj se mogu dodeliti određena svojstva (atributi) i ponašanja (operacije). Način grupisanja zavisi od shvatanja osobe ili tima koji grupiše objekte (na primer, avioni i bicikli mogu predstavljati zasebne grupe, a mogu pripadati i istoj grupi – prevozna sredstva). Dva potpuno različita grupisanja mogu se pokazati podjednako ispravnim i korisnim. Za svaki objekat se kaže da je primerak ili instanca neke klase. Svaka instanca ima svoje vrednosti atributa, a sa ostalim instancama iz klase ima samo zajednička imena atributa.
- *enkapsulacija*. Predstavlja tehniku pakovanja informacija tako da se spolja vidi ono što treba da bude vidljivo, a ne vidi se ono što treba da bude sakriveno. Klasa enkapsulira attribute i ponašanja objekta skrivajući implementacione detalje.
- *nasleđivanje*. Podrazumeva organizovanje klasa u hijerarhiju na osnovu njihovih sličnosti i razlika. Najpre se definiše klasa sa sveobuhvatnim osobinama, a zatim se rafinira u specijalizovane potklase. Potklasa može da nasleđuje kako strukturu i attribute, tako i ponašanje nadređene klase.
- *polimorfizam*. Predstavlja osobinu da se isto ponašanje različito manifestuje kod različitih klasa i potklasa. Pod ponašanjem se podrazumeva akcija koju vrši objekat ili akcija koja se vrši nad objektom. Metod klase predstavlja implementaciju operacije klase. U sistemu sa polimorfizmom, više različitih metoda implementira istu operaciju.
- *perzistencija*. Predstavlja sposobnost da ime, stanje i ponašanje objekta opstanu u vremenu, tj. da se očuvaju i nakon promene objekta. Takav objekat je trajan. Ovo se koristi ukoliko se vrednost nekog atributa često menja, a potrebno je sačuvati sve njegove vrednosti (na primer, kako bi se kasnije napravio dijagram promene vrednosti tog atributa).

OO pristup se koristi u celom procesu razvoja, počevši od definisanja zahteva, projektovanja sistema, programske implementacije, pa do testiranja. Ta konzistentnost u terminologiji predstavlja važnu razliku OO pristupa od ostalih tradicionalnijih pristupa. Pošto OO pristup koristi enkapsuliranje informacija, mnogi projektanti posmatraju klase i objekte sa stanovišta mogućnosti njihovog ponovnog korišćenja u drugim projektima.

OO analiza zahteva obično se iskazuje jezikom bliskim korisniku i opisuje pojmove i moguća scenarija iz posmatranog domena. Pojmovi obuhvataju razne

informacije, usluge i odgovornosti. Specifikacija zahteva koja odražava OO pristup može da predstavlja prve korake u projektovanju sistema. To je zato što se identifikuju objekti u sistemu i definišu njihove međusobne veze. Pri projektovanju sistema, važno je da se prepoznaju objekti realnog sveta, tj. objekti iz domena problema, njihova svojstva i ponašanja. Takođe, treba uočiti i međusobna dejstva i odnose među objektima, kao što su asocijacije, nasleđivanje i sl. Projekat sistema predstavlja visok nivo apstrakcije onoga što će biti sadržano u projektu programa. Projektanti programa na osnovu projekta sistema uvode niz detalja bitnih za implementaciju, kao što su razna izračunavanja, proračun performansi, analiza bezbednosti i dr. Nakon projektovanja programa, sistem je opisan na vrlo niskom nivou apstrakcije pomoću objekata. Sledi proces programiranja kojim se modeli prevode u kôd na nekom OO programskom jeziku. Prilikom kodiranja, teži se uopštavanju sistema kako bi iste klase i objekti mogli kasnije ponovo da se koriste. Po završetku kodiranja, sledi testiranje programa. Najpre se testira ispravnost pojedinačnih modula, zatim se proverava ispravnost integrisanog sistema, i na kraju se vrši završno testiranje, kako bi se utvrdilo da li sistem ispunjava postavljene zahteve.

Programeri implementiraju sistem (pišu programski kôd) na osnovu projekta programa koji sadrži ne samo klase i objekte iz projekta sistema, već i neke dodatne objekte. Ovi objekti nastaju kao rezultat dodatnih razmatranja:

- nefunkcionalnih zahteva sistema, kao što su potrebne performanse ili ograničenja po pitanju izgleda korisničkog interfejsa
- ponovne upotrebljivosti komponenti iz ranije projektovanih sistema
- primenljivosti razmatranih komponenti u budućim sistemima
- zahteva po pitanju korisničkog interfejsa
- struktura podataka koje će biti korišćene i načinima upravljanja podacima

U projektu programa pojavljuju se mnogi objekti koje korisnik ne vidi, jer za njih ne postoje odgovarajući objekti u realnom svetu. Na primer, u projektu sistema može da postoji opis na višem nivou kako su podaci organizovani i kako im se pristupa. Međutim, u fazi projektovanja programa, moraju se doneti konkretne odluke o strukturama u kojima će se čuvati podaci (na primer, o matricama, nizovima i sl.) jer su one neophodne kako bi programeri mogli da pristupe kodiranju. Ovde se vidi jasna razlika između nivoa apstrakcije prisutnog u projektovanju sistema (visok nivo) i projektovanju programa (nizak nivo).

5 UML modelovanje

UML (*Unified Modeling Language*) je nastao 1996.godine objedinjavanjem tri, u to vreme najuticajnije metode modelovanja: Booch-ove, OMT (*Object Modeling Technique*, Rumbaugh) i OOSE (*Object Oriented Software Engineering*, Jacobson). Ideja je bila da se prevaziđe problem neusaglašenosti postojećih metoda i formira jedinstven jezik za modelovanje koji bi bio opšte prihvaćen kao standard.

UML predstavlja skup grafičkih notacija zasnovanih na jedinstvenom metamodelu. Pod grafičkom notacijom podrazumeva se skup grafičkih elemenata koji definišu sintaksu jezika. Metamodel je dijagram koji opisuje koncepte jezika za modelovanje. UML je široko prihvaćen u praksi, između ostalog, i zbog toga što se ni grafička notacija, ni metamodel ne moraju strogo primenjivati. Oni predstavljaju samo pokušaj uvođenja discipline, ali je korisnost modela uvek na prvom mestu.

UML standard obuhvata veći broj dijagrama za modelovanje (u veziji UML 2.4 iz 2011.godine ih je 14) i OCL jezik za specifikaciju ograničenja.

U nastavku su opisani osnovni UML dijagrami koji se koriste u različitim fazama procesa razvoja softvera. Za svaki dijagram navedena je njegova uloga, osnovni elementi i način njegovog generisanja.

5.1 Dijagrami slučajeva korišćenja

Svaki softverski sistem ima svoje korisnike iz spoljašnjeg okruženja. To mogu biti pojedinci, grupe ljudi, ali isto tako i drugi računari ili uređaji. Zato je u procesu modelovanja neophodno utvrditi granice sistema i precizno definisati načine na

koje sistem ostvaruje interakciju sa spoljašnjim svetom. Tako se prikupljaju funkcionalni zahtevi sistema. Nakon identifikacije svih mogućih učesnika u interakciji, za svakog od njih treba ustanoviti postupak same interakcije.

Slučaj korišćenja predstavlja opis postupka interakcije između korisnika i sistema. Treba ga razlikovati od funkcije sistema, jer mu je namena drugačija. Funkcije opisuju sistem, a slučajevi korišćenja opisuju kako se sistem koristi.

Svaki slučaj korišćenja ima svoj jedinstveni cilj. Do tog cilja ne može se uvek doći samo na jedan način, tj. jednom putanjom. Stoga je uveden pojam scenarija. *Scenario* predstavlja niz sukcesivnih koraka kojima se opisuje interakcija korisnika sa sistemom. Slučaj korišćenja može da sadrži više scenarija, pri čemu svi scenariji moraju da budu vođeni istim ciljem.

Način pisanja slučaja korišćenja nije standardizovan, tako da se mogu primenjivati različiti formati. Međutim, uobičajeno je da se slučajevi korišćenja pišu na prirodnom jeziku u obliku priče. Najpre se navodi osnovni, uspešni scenario u vidu numerisanih koraka, a zatim sva odstupanja od osnovnog scenarija (takođe u vidu numerisanih koraka) sa jasnim referisanjem na mesta povratka u osnovnom scenariju (ukoliko ih ima). Svaki korak predstavlja jedan deo interakcije. Opis koraka mora da bude jasan i da ukazuje na njegovog izvršioca. Korak pokazuje nameru učesnika, a ne način na koji se nešto ostvaruje. Postupak pisanja slučaja korišćenja biće ilustrovan na jednom primeru.

Neka slučaj korišćenja *Kupovina proizvoda* treba da modelira postupak kako kupac kupuje neki proizvod. Najpre treba napisati osnovni uspešan scenario. On sadrži uobičajene korake u kupovini, kao što je dato na slici 5.1. Nakon toga treba uočiti moguća odstupanja od osnovnog scenarija, bilo da je reč o mogućim greškama (pogrešno uneti podaci) ili o dodatnim mogućnostima. U datom primeru, u okviru proširenja, dodata su dva nova scenarija: jedan opisuje šta treba raditi ukoliko je kupac redovan, pa ima određene povlastice, dok drugi scenario govori o tome šta se dešava u slučaju kada kupac unese pogrešan broj platne kartice. Pri pisanju slučaja korišćenja, bitno je da se preciziraju koraci u osnovnom scenariju u kojima dolazi do odstupanja, kao i mesta povratka iz dodatnih scenarija.

Slučaj korišćenja može da bude prilično složen u zavisnosti od postavljenog cilja. U tom slučaju je pogodno razložiti ga na više jednostavnijih slučajeva korišćenja, koji se onda uključuju u polazni slučaj korišćenja. Ne postoji standardan način za uključivanje jednog slučaja korišćenja u drugi. Ipak, za povezivanje se često koristi hiperveza u vidu teksta koji je podvučen punom linijom (videti korak 1 u osnovnom scenariju). Kada se u nekom slučaju korišćenja pojavi hiperveza, to znači da je negde definisan uključen slučaj korišćenja na koji ukazuje hiperveza. U datom primeru, postoji posebno definisan slučaj korišćenja koji opisuju pregled kataloga.

Kupovina proizvoda

Osnovni uspešan scenario:

1. Kupac pregleda katalog sa proizvodima.
2. Kupac iz kataloga bira proizvod koji hoće da kupi.
3. Kupac unosi podatke o isporuci (adresu i uslove isporuke).
4. Sistem prikazuje podatke o troškovima.
5. Kupac unosi podatke o platnoj kartici.
6. Sistem proverava podatke o načinu plaćanja.
7. Sistem potvrđuje prodaju.
8. Sistem šalje kupcu priznanicu elektronskom poštom.

Proširenja:

- 3a. Kupac je redovan.
 - 1: Sistem prikazuje podatke o isporuci, cenama, troškovima.
 - 2: Kupac potvrđuje ili menja vrednosti, povratak na korak 6.
- 6a. Podaci o platnoj kartici nisu ispravni.
 - 1: Sistem nudi kupcu mogućnost da ponovo unese podatke o kartici ili se rad prekida, povratak na korak 7.

Slika 5.1 Sadržaj slučaja korišćenja *Kupovina proizvoda*

Uključeni slučajevi korišćenja su pogodni za opisivanje složenih koraka koji bi zauzeli puno prostora u osnovnom scenariju (detaljni opisi se teško čitaju). Takođe se koriste i ukoliko se isti koraci javljaju u više različitih slučajeva korišćenja. Da se isti koraci ne bi više puta puta ponavljali, mogu se izdvojiti u poseban, uključeni slučaj korišćenja, koji bi se onda povezao sa svim ostalim nadređenim slučajevima korišćenja. Međusobnim povezivanjem slučajeva korišćenja može se ostvariti više nivoa ugnježdavanja, ali preglednosti radi, ne sme se preterati u broju nivoa.

Osim sadržaja, slučajevi korišćenja mogu da poseduju i neke dodatne informacije, kao što su:

- *preduslov (pre-condition)* koji opisuje šta treba da bude ispunjeno pre nego što sistem dopusti izvršavanje slučaja korišćenja
- *garancija (guarantee)* koja opisuje stanje sistema nakon izvođenja slučaja korišćenja
- *okidač (trigger)* koji definiše događaj koji dovodi do započinjanja izvođenja slučaja korišćenja

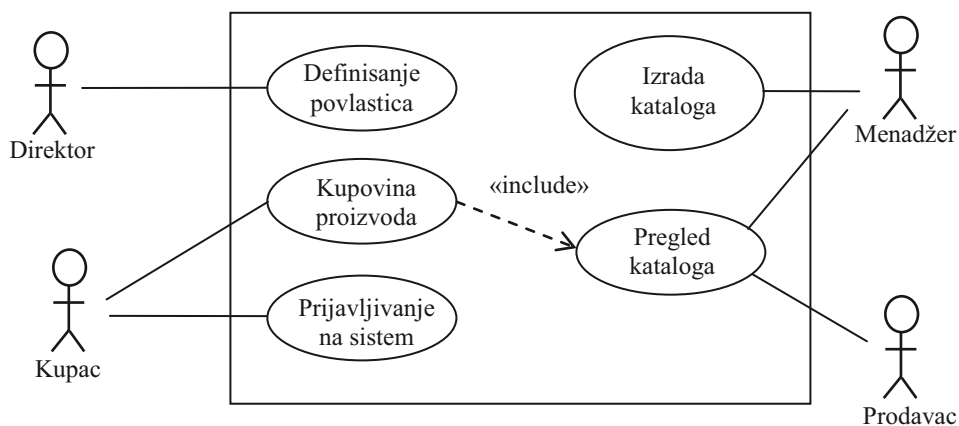
Uključivanje ovih dodatnih mogućnosti može da doprinese efikasnijem radu, ali direktno ima uticaja i na potencijalne rizike. Stoga dodatne informacije treba oprezno uvoditi, jer se one onda moraju strogo poštovati.

Svaki softverski sistem podržava veći broj opcija, što znači da se može koristiti na različite načine. Zato je prilikom njegovog modelovanja potrebno definisati veći broj slučajeva korišćenja. Osim sadržaja ovih slučajeva korišćenja, važno je naglasiti i veze koje postoje između njih, kao i njihove izvršioci. Jedan slučaj korišćenja može da ima više izvršioca (ili učesnika – *actors*). Tada je jedan učesnik glavni (*primary actor*) i to je onaj koji traži uslugu od sistema. Ostali učesnici sa kojima sistem komunicira su sporedni (*secondary actors*). Takođe, jedan učesnik u sistemu može da izvodi više slučajeva korišćenja.

Dijagram slučajeva korišćenja predstavlja grafički prikaz skupa svih slučajeva korišćenja u sistemu ili delu sistema. On ukazuje na granice sistema i njegovu interakciju sa spoljašnjim svetom. Na dijagramu se mogu uočiti sledeći grafički elementi:

- *elipse* koje prikazuju slučajeve korišćenja
- *pojednostavljeni simboli čoveka* koji prikazuju učesnike, tj. izvršioci slučajeva korišćenja
- *pune linije* koje povezuju slučaj korišćenja sa njegovim izvršiocom
- *isprekidanje linije sa strelicom* koje povezuju uključene sa ostalim slučajevima korišćenja

Na slici 5.2 dat je primer jednog dijagrama slučajeva korišćenja koji sadrži slučaj korišćenja *Kupovina proizvoda* čiji je sadržaj ranije opisan.



Slika 5.2 Primer dijagrama slučajeva korišćenja

Zbog mogućeg velikog broja slučajeva korišćenja u sistemu, oni se razvrstavaju u tri nivoa:

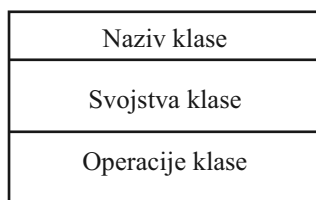
- *osnovni nivo (sea-level)*. Ovom nivou pripadaju centralni slučajevi korišćenja koji obično predstavljaju interakcije glavnog učesnika sa sistemom.
- *niži nivo (fish-level)*. Na ovom nivou se nalaze slučajevi korišćenja uključeni u slučajeve korišćenja na osnovnom nivou.
- *viši nivo (kite-level)*. Ovaj nivo obuhvata uglavnom poslovne slučajeve korišćenja koji pokazuju kako se slučajevi sa osnovnog nivoa uklapaju u širi kontekst poslovnih interakcija.

Najveći broj slučajeva korišćenja u sistemu pripada osnovnom nivou.

5.2 Dijagrami klasa

Dijagrami klasa opisuju tipove entiteta u sistemu, različite vrste statičkih veza koje postoje između entiteta, kao i ograničenja u načinu njihovog povezivanja. Zbog svog sadržaja, ovi dijagrami se generišu praktično u svim fazama razvoja, tako da predstavljaju najčešće korišćene UML dijagrame. Ovi dijagrami imaju najbogatiji skup tehnika za modelovanje od svih UML dijagrama.

Osnovni grafički element u dijagramu klasa je model klase. On se sastoji od tri komponente: imena klase, svojstva klase i operacija klase, kao što je prikazano na slici 5.3.



Slika 5.3 Model klase

Svojstva klase opisuju strukturne karakteristike klase. Svojstva se pojavljaju u dva oblika, kao atributi klase i kao asocijacije. Između atributa i asocijacija postoje velike sličnosti, tako da se većina informacija može modelovati i na jedan i na

drugi način. Ipak, postoje i male razlike koje idu u prilog asocijacijama, a koje će biti iznete u nastavku.

Pošto se ista informacija može modelovati na dva načina, postavlja se pitanje kada koristiti koji od njih. Atributi prikazuju svojstvo u tekstualnom obliku, dok asocijacije daju grafički prikaz koji je uvek ilustrativniji. Stoga se atributi obično koriste za opis jednostavnijih i manje važnih svojstava, kao što su datumi, vrednosti logičkih promenljivih i dr., dok su asocijacije pogodnije za vizuelno naglašavanje važnijih osobina klasa. Prema tome, izbor načina za prikaz nekog svojstva zavisi prvenstveno od toga da li želimo da to svojstvo posebno istaknemo (tada koristimo asocijacije) ili ne (tada koristimo attribute).

Atribut opisuje svojstvo u vidu reda teksta unutar klase. Potpuni oblik zapisa atributa je

vidljivost ime:tip kardinalnost = podrazumevana_vrednost {opis_svojstva}

pri čemu:

vidljivost označava da li je atribut javni (+), privatni (-), paketni (~) ili zaštićeni (#)

ime predstavlja naziv atributa u klasi

tip ukazuje na to da postoji ograničenje po pitanju vrste objekata koji imaju atribut

kardinalnost pokazuje na koliko objekata se odnosi svojstvo

podrazumevana_vrednost predstavlja vrednost atributa u novom objektu, ukoliko se drugačija vrednost ne zada tokom generisanja objekta

opis_svojstva omogućava definisanje novih osobina atributa (na primer, ako se koristi *{readOnly}* znači da nije dozvoljeno menjanje svojstva)

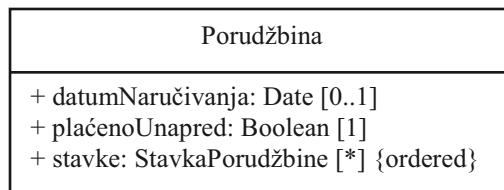
Kardinalnost se definiše zadavanjem donje (DG) i gornje (GG) granice u obliku DG..GG. DG može biti pozitivan ceo broj ili nula, a GG pozitivan ceo broj ili * koja označava da nema ograničenja sa gornje strane. Najčešće se sreću sledeći oblici kardinalnosti.

- 1, što označava da su DG i GG jednake; ekvivalentan zapis je 1..1 (na primer, ovako se definiše kardinalnost za slučaj „Jedna porudžbina mora da ima tačno jednog kupca.“)
- 0..1, što označava da se svojstvo primenjuje na jedan ili nijedan objekat (na primer, na ovaj način se modeluje da „Firma može da ima posebnog predstavnika, ali i ne mora.“)

- *, što predstavlja skraćeni zapis za 0..*; (na primer, ovako se modeluje da „Kupac ne mora da pošalje porudžbinu, ali ako želi, može da pošalje neograničen broj porudžbina.“)

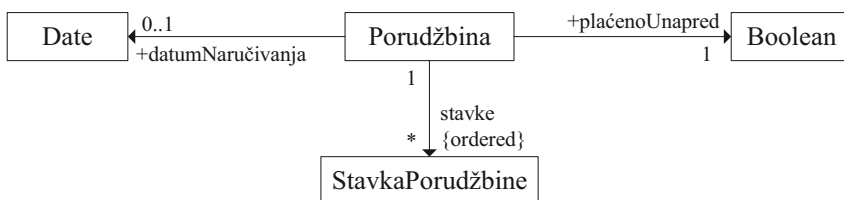
Asocijacija opisuje svojstvo punom linijom između dve klase, usmerenom od izvorne ka odredišnoj klasi. Ime svojstva, kao i kardinalnost, navode se na odredišnom kraju asocijacije. Sama odredišna klasa određuje tip svojstva.

Definisanje atributa i asocijacija će biti ilustrovano na jednom primeru. Neka klasa *Porudžbina* poseduje tri svojstva: *datumNaručivanja*, (opisuje da li je i kada porudžbina poslata), *plaćenoUnapred* (ukazuje na to da li je uplata izvršena) i *stavke* (definiše skup naručenih stavki, pri čemu je svaka stavka opisana klasom *StavkaPorudžbine* koja mora biti posebno definisana). Ova svojstva se mogu u okviru klase definisati javnim atributima kao što je dato na slici 5.4.



Slika 5.4 Modelovanje svojstava primenom atributa

Isti skup svojstava može se predstaviti i na drugi način, tj. pomoću asocijacija kao na slici 5.5.



Slika 5.5 Modelovanje svojstava primenom asocijacija

Kao što se vidi sa slike, kardinalnost može biti zadata na oba kraja asocijacije što, u odnosu na opis pomoću atributa, predstavlja izvesnu prednost. Tako je u datom primeru kardinalnošću 1 opisano da jedna stavka porudžbine može da pripada samo jednoj porudžbini, a kardinalnošću * (na drugom kraju asocijacije) da porudžbina može da ima neograničen broj stavki.

Operacije klase opisuju aktivnosti koje klasa može da obavi. Sintaksa operacija je sledeća:

vidljivost ime (lista_parametara) : tip_rezultata {opis_svojstva}

pri čemu:

vidljivost ima isto značenje kao u slučaju atributa

ime predstavlja naziv operacije

lista_parametara ukazuje na parametre operacije

tip_rezultata pokazuje kog je tipa rezultat operacije, ukoliko operacija ima rezultat

opis_svojstva opisuje svojstva operacije

Parametri iz liste parametara definišu se sledećom sintaksom:

smer ime: tip = podrazumevana_vrednost

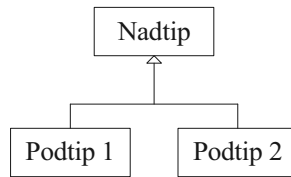
pri čemu:

ime, *tip* i *podrazumevana_vrednost* imaju isto značenje kao u sintaksi za definisanje atributa

smer ukazuje na to da li je parametar ulazni (*in*), izlazni (*out*) ili ulazno-izlazni (*inout*); ako *smer* nije označen, podrazumeva se da je parametar ulazni (*in*)

U UML-u se razlikuju dve vrste operacija: upiti i modifikatori. *Upiti* samo čitaju neku vrednost iz klase ne menjajući vidljivo stanje sistema (stanje koje se može uočiti spolja). Oni nemaju sporednih efekata i obično se označavaju opisom svojstva {*query*}. Redosled izvršavanja upita se može promeniti, a da se ne promeni ponašanje sistema. *Modifikatori* su operacije koje menjaju stanje sistema. Uobičajeno je da oni ne vraćaju nikakav rezultat.

Dijagramom klasa može se modelovati i *generalizacija*. Ona podrazumeva smeštanje zajedničkih osobina više klasa u opštu klasu koja predstavlja nadtip. Sve što važi da klasu koja je nadtip, važi i za klase koje su podtipovi (videti sliku 5.6).



Slika 5.6 Generalizacija

Generalizacija se realizuje pomoću nasleđivanja (*inheritance*). Važne posledice nasleđivanja su:

- *zamenljivost*, koja omogućava da se umesto objekta klase koja je nadtip može koristiti objekat bilo kog podtipa te klase
- *polimorfizam*, koji omogućava da objekat klase koja je podtip može na neke komande odgovoriti drugačije od objekta klase koja je nadtip, pri čemu je to transparentno za klijenta

Dijagramom klasa se mogu predstaviti i *zavisnosti* između elemenata. Dva elementa su zavisna ako promena definicije jednog elementa (davaoca) prouzrokuje promenu definicije drugog elementa (klijenta). Zavisnost između klasa se javlja u slučajevima kada:

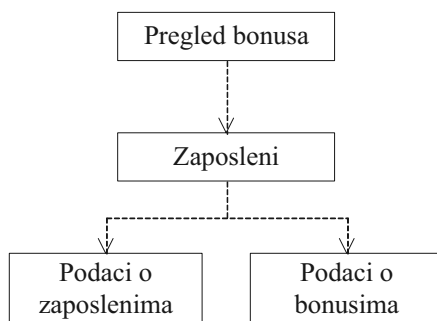
- jedna klasa šalje poruku drugoj klasi (ako se promeni interfejs neke klase, može se desiti da poruke koje su joj upućene ne budu više ispravne)
- jedna klasa sadrži drugu klasu
- objekat jedne klase prosleđuje objekat druge klase kao parametar neke operacije

Upravljanje zavisnostima je od izuzetnog značaja za sistem. Pošto su promene u sistemu česte, one se propagiraju kroz sistem tako što utiču na pojedine elemente. Što je veći broj elemenata koje treba promeniti, to je proces izmena teži. Pošto se izmene u klasama uglavnom reflektuju samo preko interfejsa, zavisnosti na nivou interfejsa treba vrlo pažljivo definisati.

U dijagramima klasa, zavisnosti se koriste uvek kada treba pokazati kako promene jednog elementa menjaju drugi element. Na slici 5.7 prikazane su zavisnosti (date isprekidanim linijama) u primeru višeslojne aplikacije.

Definisane su četiri klase: *Pregled bonusa* (koja pripada prezentacionom sloju), *Zaposleni* (iz oblasti problema), *Podaci o zaposlenima* i *Podaci o bonusima*

(iz oblasti ponašanja sistema, tj. pravila poslovanja). Jednosmerna zavisnost povezuje klasu *Pregled bonusa* sa klasom *Zaposleni*. To znači da promene u klasi *Pregled bonusa* neće uticati na klasu *Zaposleni* (kao ni na ostale klase), dok promene u klasi *Zaposleni* utiču na klasu *Pregled bonusa* samo ako se radi o promeni interfejsa klase *Zaposleni*. Klasa *Pregled bonusa* nema direktnu zavisnost od klasa sa podacima. Ukoliko se klase sa podacima promene, to može dovesti do promene klase *Zaposleni*. Međutim, uticaj ovih promena se tu završava, ukoliko se menja samo realizacija klase *Zaposleni*, a ne i njen interfejs.



Slika 5.7 Primer definisanja zavisnosti

Ograničenja u sistemu se uglavnom modeliraju osnovnim elementima u dijagramu klasa, kao što su: atributi, asocijacije, ili generalizacija. Međutim, ne mogu se sva ograničenja prikazati na ovaj način. UML dopušta definisanje proizvoljnih ograničenja, uz poštovanje samo jednog pravila, a to je da opis ograničenja mora biti dat unutar vitičastih zagrada (`{}`).

Za predstavljanje ograničenja mogu se koristiti:

- prirodni jezik - ovaj način je neprecizan jer može da dovede do različitih tumačenja, ali se ipak preporučuje
- neki od programskih jezika
- OCL (*Object Constraint Language*) – UML-ov formalni jezik ograničenja koji je dovoljno precizan, ali je neophodno poznavanje ovog jezika da bi se razumelo ograničenje

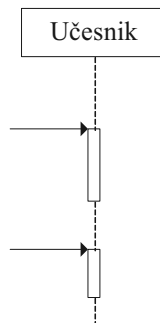
Ograničenju se može dodeliti ime koje je odvojeno dvotačkom. Primer ograničenja napisanog na prirodnom jeziku je:

{onemogućiti neregularnost ispita: student mora položiti ispit pre unosa ocene}

5.3 Dijagrami sekvence

Dijagrami sekvence opisuju saradnju objekata u obavljanju neke aktivnosti vodeći računa o vremenskoj komponenti. Obično prikazuju scenario u okviru slučaja korišćenja koji obuhvata izvestan broj objekata i poruka koje objekti međusobno razmenjuju.

Na dijagramu sekvence, svaki učesnik u poslu prikazuje se pravougaonikom sa nazivom učesnika. Iz pravougaonika polazi vertikalna isprekidana linija života (*lifeline*) koja označava da je učesnik kreiran, tj. da postoji u sistemu. Učesniku stižu poruke (*messages*) koje se na dijagramu predstavljaju horizontalnim punim linijama sa strelicom na vrhu i čitaju se odozgo na dole. Kada učesniku stigne poruka, to znači da on treba da se uključi u neki posao. Od mesta prispeća poruke, duž linije života se generiše pravougaonik koji predstavlja traku aktivnosti (*activation bar*). Traka aktivnosti pokazuje da je učesnik aktivan u interakciji. Trake odgovaraju izvršavanju metoda učesnika. Na slici 5.8 dati su osnovni elementi dijagrama sekvence.

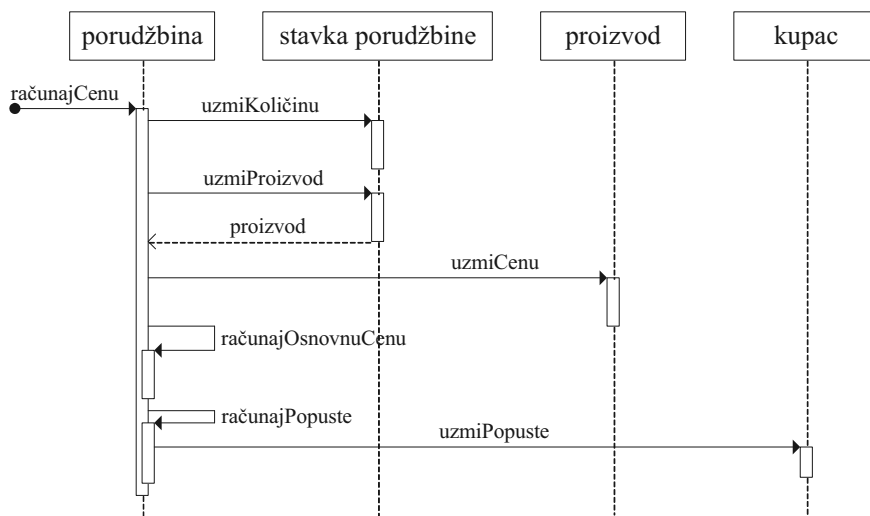


Slika 5.8 Elementi dijagrama sekvence

Isti scenario slučaja korišćenja može se pomoću dijagrama sekvence modelovati na različite načine. Mogu se izdvojiti dva opšta pristupa: centralizovano i distribuirano upravljanje. Pristupi se razlikuju po načinu saradnje učesnika u realizaciji datog scenarija.

Navedeni pristupi biće predstavljeni na primeru jednog scenarija. Neka je data porudžbina koja obuhvata više stavki. Svaka stavka sadrži: naziv proizvoda koji je naručen, količinu koja je naručena i jediničnu cenu proizvoda. Po posmatranom scenariju, porudžbina prima poruku kojom se traži izračunavanje vrednosti cele porudžbine. Da bi izvršila ovaj zahtev, porudžbina mora da pregleda sve svoje stavke i izračuna njihove vrednosti na osnovu naručenih količina i jediničnih cena proizvoda. Na kraju, porudžbina treba da izračuna popust na osnovu pravila koja

važe za konkretnog kupca. Ovaj scenario se može modelovati kao što je prikazano na slici 5.9.



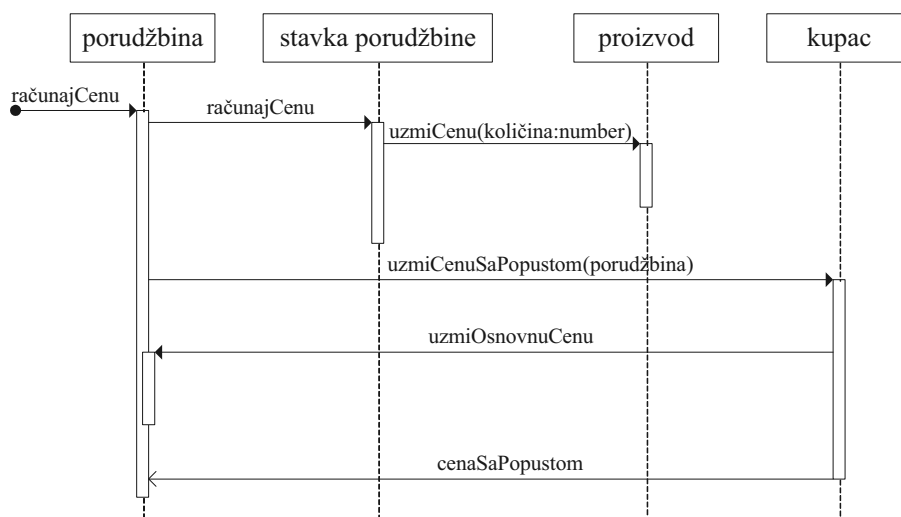
Slika 5.9 Primer centralizovanog upravljanja

Kao što se vidi, na dijagramu postoje četiri učesnika: *porudzbina*, *stavka porudzbine*, *proizvod* i *kupac*. Primljena poruka *računajCenu* formira traku aktivnosti na liniji života učesnika *porudzbina*. Ovaj učesnik šalje poruku *uzmiKoličinu* učesniku *stavka porudzbine* i traži od njega podatak o naručenoj količini proizvoda. Zatim, na sličan način zahteva i podatak o vrsti proizvoda. Nakon toga, šalje učesniku *proizvod* poruku *uzmiCenu* da bi imao na raspolaganju i taj podatak. Pozivom svoje metode, na osnovu dobijenih podataka, učesnik *porudzbina* najpre računa cenu porudžbine, a zatim i popust. Pošto mu nedostaje informacija o popustima, zahteva je od učesnika *kupac* slanjem poruke *uzmiPopuste*.

Može se videti da je na dijagramu nacrtana samo jedna povratna strelica (isprekidana linija). Iako slične veze postoje za svaku poslatu poruku, preglednosti radi, povratne veze ne moraju da budu nacrtane na dijagramu.

Opisani postupak modelovanja naziva se centralizovanim upravljanjem zato što jedan učesnik (*porudzbina*) obavlja obradu, a drugi učesnici ga snabdevaju potrebnim podacima. To znači da učesnici nisu podjednako opterećeni, tj. nemaju isto učešće u poslu. Ipak, ovakav način modelovanja je vrlo pregledan i jednostavan, pa je pogodan za početnike.

Isti primer može se modelovati i na način prikazan na slici 5.10.



Slika 5.10 Primer distribuiranog upravljanja

U ovom slučaju, učesnici sarađuju na drugačiji način. Nakon prijema poruke *racunajCenu*, učesnik *porudzbina* prosleđuje poruku učesniku *stavka porudzbine* i traži od njega da izračuna sopstvenu cenu. Međutim, i ovaj učesnik prepušta izračunavanje cene drugom učesniku (*proizvod*), s tim što mu u poruci prosleđuje parametar o količini naručenog proizvoda. Kada dobije informaciju o vrednosti porudžbine, učesnik *porudžbina* šalje poruku učesniku *kupac* tražeći od njega da izračuna popust. Pošto nema informaciju o vrednosti porudžbine, *kupac* upućuje povratnu poruku učesniku *porudžbina* kako bi dobio potrebni podatak.

Ovakav način modelovanja naziva se distribuiranim upravljanjem zato što je obrada raspoređena između više učesnika, od kojih svaki izvršava svoj deo posla. Kao što se vidi, preglednost dijagrama je manja nego u slučaju centralizovanog upravljanja, ali je postignuto ravnomernije opterećenje učesnika. Osim toga, ovaj način upravljanja je efikasniji, ali zahteva iskusnije projektante.

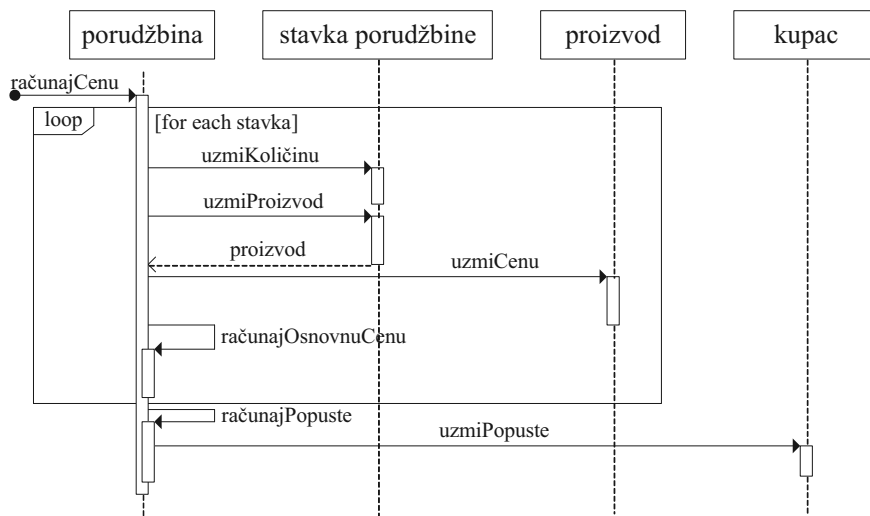
Pažljivom analizom, može se primetiti da na oba data dijagrama sekvence postoji jedan nedostatak. Naime, u oba slučaja se računa cena samo jedne stavke u porudžbini. To je zato što nigde ne postoji petlja u okviru koje bi se prolazilo kroz sve stavke porudžbine.

Dijagrami sekvence mogu da modeluju petlje, ali se ne preporučuje njihovo korišćenje u tu svrhu. Oni su prvenstveno namenjeni vizualizaciji interakcije između objekata, a ne modelovanju logike upravljanja.

Ipak, u UML-u je definisana notacija za modelovanje petlji u dijagramu sekvence. Petlje se opisuju pomoću *okvira interakcije*. Okvirom se ograničava deo prostora na dijagramu sekvence koji obuhvata ono što treba da se izvrši u petlji. U

uglu okvira navodi se operator *loop*, a desno od njega (u uglastim zagradama) opis iteracije.

Kompletan dijagram sekvence za dati primer prikazan je na slici 5.11.



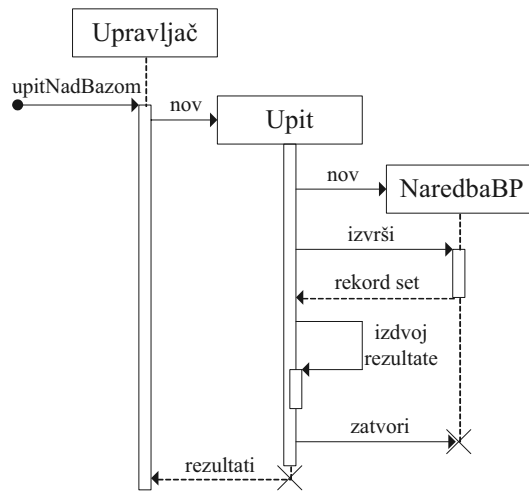
Slika 5.11 Modelovanje petlje

Na dijagramu sekvence, učesnici se mogu kreirati i brisati. Objekti se brišu kada nisu više potrebni kako ne bi opterećivali sistem.

Kreiranje učesnika se vrši tako što se nacrtava poruka povezana neposredno sa nazivom učesnika. Ako kreirani učesnik odmah započinje neku aktivnost, traka aktivnosti se crta kao da izlazi iz učesnika, a ako ne, crta se linija života kao što je uobičajeno.

Brisanje učesnika se označava ukrštenim linijama X. Učesnika može da izbriše drugi učesnik ako pošalje poruku čija strelica ulazi u X na kraju linije života učesnika koji se briše. Učesnik može i samog sebe da izbriše navođenjem X na kraju svoje linije života.

Na primer, neka je potrebno da se iz baze podataka pročita neki podatak (slika 5.12). Učesnik *Upravljač* dobija poruku da izvrši ovu aktivnost. Da bi to uradio, on može da kreira učesnika *Upit* koji odmah počinje sa radom. Kreira novog učesnika *NaredbaBP* i šalje mu poruku *izvrši*. Ovaj učesnik izvršava upit i dobijeni rekord set prosleđuje učesniku *Upit*. On poziva svoj metod koji iz rekord seta izdvaja traženi podatak. Nakon toga šalje poruku *zatvori* kojom briše nadalje nepotrebnog učesnika *NaredbaBP*. Pošto prosledi nađeni podatak učesniku *Upravljač*, *Upit* briše i samog sebe.



Slika 5.12 Generisanje/brisanje učesnika

5.4 Dijagrami aktivnosti

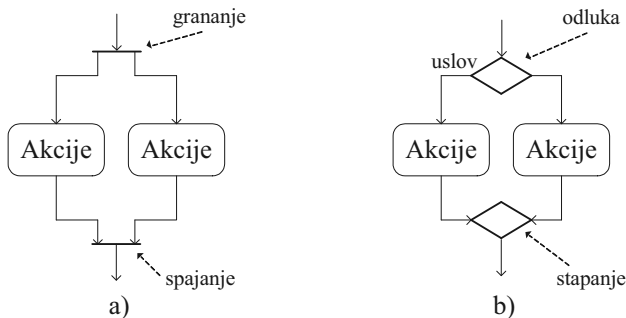
Dijagrami aktivnosti prikazuju tok posla koji treba obaviti kroz opisivanje njegovih logičkih procedura i postupaka.

Dijagram modelira aktivnost koja se sastoji od niza akcija. To znači da je aktivnost širi pojam od akcije. Akcije se izvršavaju u određenom redosledu koji zavisi od izgleda dijagrama. Dijagram aktivnosti počinje početnim, a završava se završnim čvorom.

Najvažnija uloga dijagrama aktivnosti je ta što omogućava modelovanje i paralelnih i uslovnih ponašanja. Pod paralelnim ponašanjem podrazumeva se izvršavanje više poslova istovremeno, dok uslovno ponašanje označava izbor posla koji će se izvršiti prema zadatom uslovu.

Paralelno ponašanje se na dijagramu predstavlja pomoću *grananja* (*fork*) i *spajanja* (*join*), čiji su grafički simboli dati na slici 5.13 a). Grananje ima jedan ulazni tok i više paralelnih izlaznih tokova. Svaki izlazni tok sadrži akcije koje treba izvršiti u redosledu u kome su navedene. Međutim, redosled izvršavanja akcija koje pripadaju različitim tokovima nije bitan (paralelno ponašanje). Kada se izvrše sve akcije u jednom izlaznom toku, dolazi se do spajanja. Spajanje ima više ulaznih tokova i jedan izlazni. Ulazni tokovi spajanja odgovaraju izlaznim tokovima grananja. Da bi se prošlo kroz spajanje, potrebno je ostvariti

sinhronizaciju svih ulaznih tokova, tj. sve akcije u svim tokovima moraju da budu izvršene.

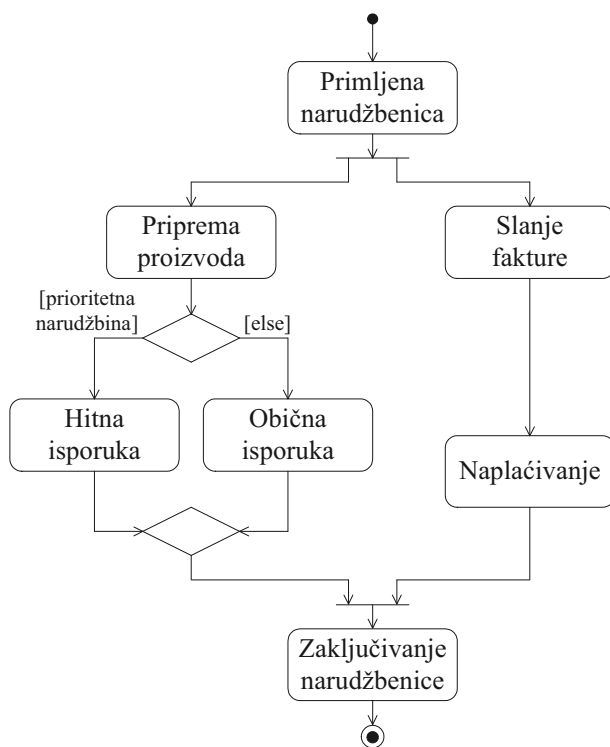


Slika 5.13 Osnovni elementi dijagrama aktivnosti

Za predstavljanje *uslovnog ponašanja* na dijagramu aktivnosti, koriste se *odluka* (*decision*) i *stapanje* (*merge*), čiji su grafički simboli dati na slici 5.13 b). Odluka ima jedan ulazni i više izlaznih tokova. Svakom izlaznom toku je pridružen uslov u vidu logičkog izraza. Pri svakom nailasku na odluku, moguće se nastaviti samo jednim od izlaznih tokova, tako da uslovi moraju da budu međusobno isključivi. Rezervisana reč *else* koristi se kao uslov koji ukazuje na tok kojim treba nastaviti ukoliko su svi ostali uslovi odluke netačni. Kroz stapanje se prolazi čim se izvrše sve akcije u toku čiji je uslov ispunjen.

Paralelno i uslovno ponašanje ilustrovani su primerom datom na slici 5.14.

Primer modelira aktivnost obrade narudžbenice. Kada se primi narudžbenica, da bi se obradila, potrebno je obaviti dva posla: isporučiti naručene proizvode kupcu i naplatiti ih. Navedeni poslovi su međusobno nezavisni (obavljaju ih različite službe) i mogu se obavljati istovremeno. Zato se modeluju pomoću dva paralelna toka, što je na dijagramu aktivnosti predstavljeno grananjem i spajanjem. Isporuka proizvoda (tok levo) zahteva najpre pripremu naručenih proizvoda, a zatim i njihovo slanje kupcu. Međutim, pošto se, u zavisnosti od prioriteta narudžbenice, proizvodi mogu slati hitnom ili običnom isporukom, to se modeluje uvođenjem odluke. Nakon izbora načina isporuke, prolazi se kroz stapanje i dolazi do spajanja. Naplata vrednosti narudžbenice (tok desno) izvodi se izvršavanjem akcija *Slanje fakture* i *Naplaćivanje*. Kroz spajanje se prolazi tek kada su oba posla izvršena (akcije iz oba toka). Tek tada postoje uslovi za zaključivanje narudžbenice, čime se aktivnost završava.

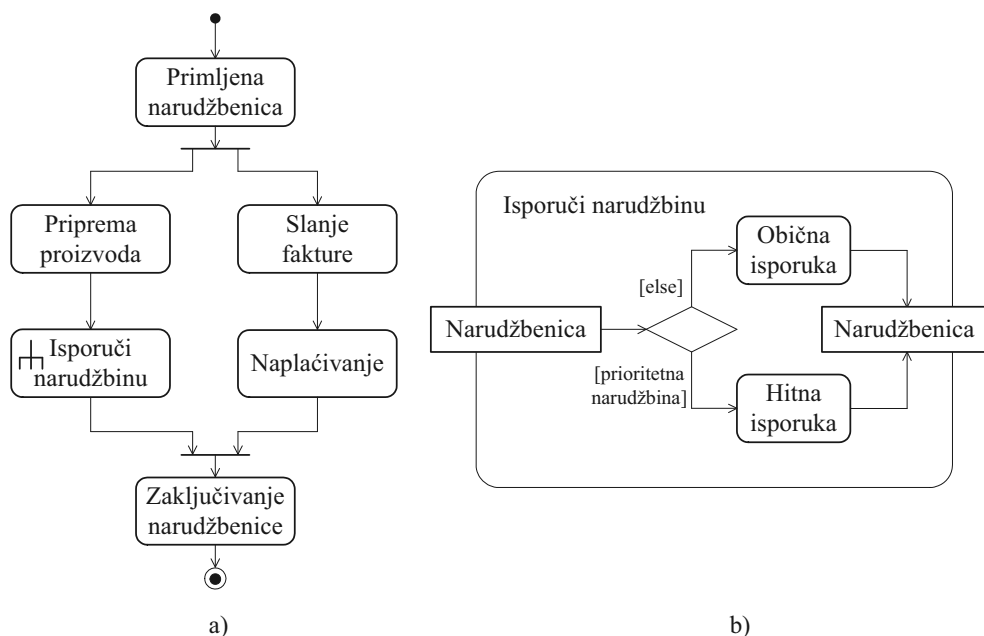


Slika 5.14 Primer dijagrama aktivnosti

Akcije u dijagramu aktivnosti mogu same po sebi da budu složene. Modelovanje ovakvih akcija dovodi do usložnjavanja dijagrama aktivnosti i smanjenja njegove preglednosti i jasnoće. Zato UML pruža mogućnost *razlaganja akcija na podaktivnosti*. Razlaganje podrazumeva izradu novog dijagrama podaktivnosti kojim se opisuje akcija. Naziv dijagrama podaktivnosti odgovara imenu akcije. Dijagram podaktivnosti ima iste ulazne i izlazne parametre kao i akcija koju opisuje. U glavnom dijagramu aktivnosti, nazivu akcije koja se razlaže pridružuje se simbol *račve*. On označava da za tu akciju postoji posebno definisan dijagram podaktivnosti.

U primeru obrade narudžbenice, može se razložiti akcija koja se bavi izborom načina isporuke proizvoda, kao što je prikazano na slici 5.15.

Dijagram pod a) odgovara ranije datom dijagramu aktivnosti u kome je simbolom račve naglašeno da je akcija *Isporuči narudžbinu* razložena. Dijagram podaktivnosti koji modelira razloženu akciju je dat pod b).

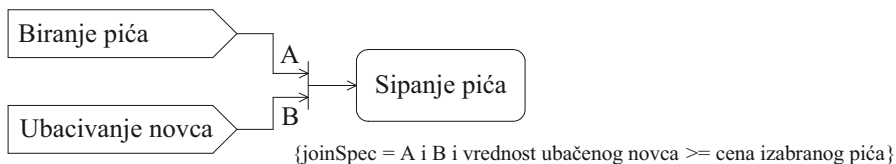


Slika 5.15 Primer razlaganja akcije

Kao što je ranije rečeno, obično se podrazumeva da spajanje dozvoljava izvršavanje izlaznog toka kada svi ulazni tokovi dođu do tačke spajanja. Međutim, nekad je korisno uvesti neko složenije pravilo. Ono se može predstaviti specifikacijom spajanja.

Specifikacija spajanja je logički izraz pridružen simbolu spajanja. Vrednost izraza se računa svaki put kada neki tok dođe u fazu spajanja. Ukoliko je uslov dat u specifikaciji ispunjen (a svi tokovi su stigli do spajanja), prelazi se na sledeću akciju.

Primer upotrebe specifikacije spajanja dat je na slici 5.16.

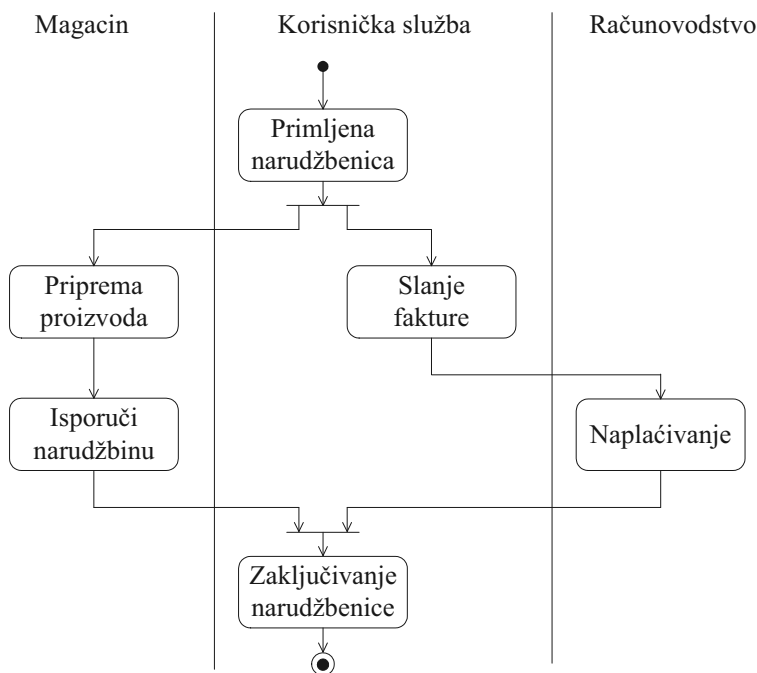


Slika 5.16 Primer specifikacije spajanja

Pretpostavimo da treba modelovati aktivnost uzimanja pića iz automata. Da bi automat dao piće, osluškuje se da li je neko izabrao vrstu pića i da li je ubacio

novac u automat. Kada se izvrše ove dve akcije, dolazi se do spajanja. Međutim, ne sme se preći na sledeću akciju sipanja pića sve dok se ne proverí da li ubačeni novac odgovara ceni željenog pića. Ovaj uslov je modelovan specifikacijom spajanja datom u vitičastim zagradama.

Dijagrami aktivnosti opisuju poslove koji treba obaviti, ali ne kazuju ko će te poslove uraditi. Pošto to može da bude vrlo važna informacija, uvedena je mogućnost podele dijagrama aktivnosti na particije. *Particije* pokazuju koja klasa ili organizaciona celina izvršava koje akcije. Postoje različite mogućnosti podele na particije. Najjednostavnija je jednodimenzionalna podela prikazana na slici 5.17 (za primer obrade narudžbenice). Ova podela se, zbog sličnosti, često naziva „plivačka staza“.



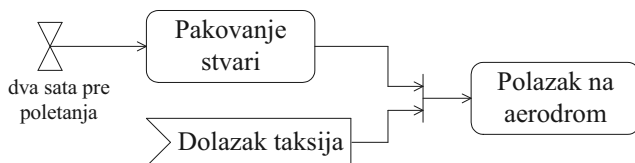
Slika 5.17 Primer podele na particije

Kao što se vidi, magacinska služba se bavi pripremom i slanjem naručenih proizvoda, korisnička služba prijemom i zaključivanjem narudžbenice i slanjem fakture, dok je računovodstvo zaduženo za naplatu.

Osim navedene, postoji i podela na particije u vidu dvodimenzionalne mreže, ili jednodimenzionalna hijerarhijska podela.

Mnogi sistemi ostvaruju interakciju sa spoljašnjim procesima. U dijagramu aktivnosti, ova interakcija se modelira pomoću *signala* koje akcije primaju iz spoljašnjeg sveta ili ih šalju u svet.

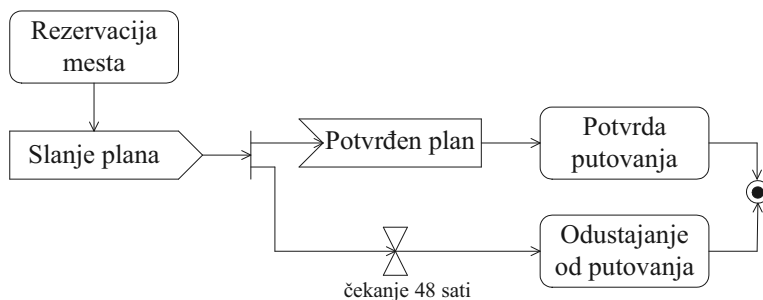
Aktivnost koja prima signal spolja, neprestano osluškuje signale, a na dijagramu se prikazuje kako će aktivnost reagovati na prispeli signal. Primer prijema signala dat je na slici 5.18.



Slika 5.18 Primer prijema signala

Primer modeluje odlazak na putovanje avionom. Kao što se vidi, dva sata pre poletanja treba početi sa pakovanjem stvari. To je predstavljeno peščanim satom koji označava vremenski signal, tj. signal koji nastaje zbog protoka vremena (ne dobija se iz spoljašnjeg sveta). Istovremeno se očekuje signal iz spoljašnjeg sveta da je stigao taksij, što je predstavljeno posebnim grafičkim simbolom za prijem. Spajanje se prolazi tek kada je stigao taksij, a i stvari su spakovane, pa postoje uslovi da se krene na aerodrom.

Primer na slici 5.19 ilustruje slanje signala u okruženje. Slanje signala je korisno kada treba sačekati odgovor pre nego što se posao nastavi. To je standardan način za opisivanje isticanja dozvoljenog vremena.

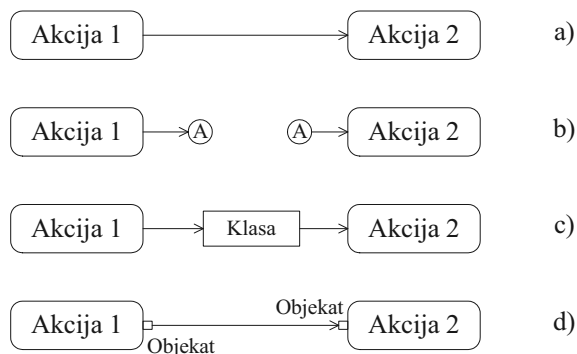


Slika 5.19 Primer slanja signala

U datom primeru, nakon rezervisanja mesta, šalje se plan i očekuje signal o njegovoj potvrdi. Kao što se vidi, simbol za prijem signala može da ima i ulazni tok koji označava da osluškivanje započinje tek kada ulazni tok aktivira prijem. Po prijemu potvrde plana, potvrđuje se i putovanje. Istovremeno (paralelno), čeka se 48 sati da stigne potvrda plana. Ako u tom periodu potvrda ne stigne, odustaje se od putovanja. Prema tome, paralelni tok koji prvi stigne do završetka, prekida drugi tok.

Veze između akcija u dijagramu aktivnosti predstavljaju se *tokovima* ili *ivicama*. Postoje četiri ekvivalentna načina za povezivanje akcija:

- punom linijom koja se završava strelicom (slika 5.20 a)). U ovom slučaju ivici se može dodeliti ime, ali ne mora.
- parovima konektora (slika 5.20 b)). Konektori u jednom paru moraju biti obeleženi istom oznakom da bi se znalo da čine par. U svakom paru, jedan konektor ima ulazni, a drugi izlazni tok. Konektori se koriste u velikim dijagramima aktivnosti kada treba povezati dve akcije na udaljenim krajevima dijagrama. Da se ne bi povlačile linije preko celog dijagrama, crtaju se konektori koji direktno povezuju akcije.
- prosleđivanjem objekata duž ivice crtanjem simbola klase (slika 5.20 c)).
- prosleđivanjem objekata duž ivice uz dodavanje nožica simbolima akcija (slika 5.20 d)). O nožicama će biti reči u nastavku.



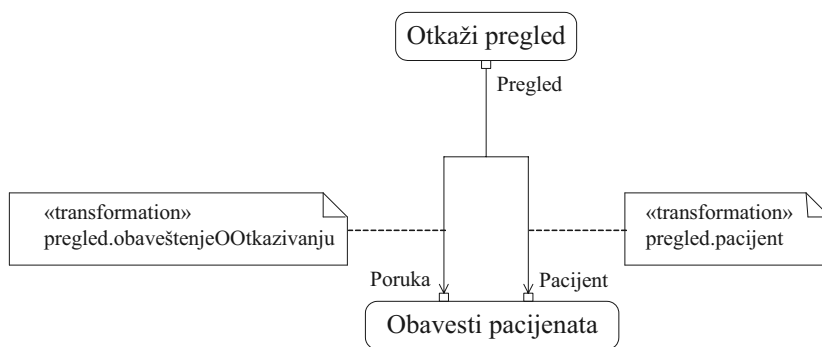
Slika 5.20 Načini povezivanja akcija

Akcije mogu da imaju parametre. Parametri se mogu prikazivati na dijagramu aktivnosti, ali i ne moraju. Ukoliko je potrebno prikazati informacije o

parametrima, za njihovo modelovanje se koriste *nožice*. Na dijagramu, nožice se crtaju u vidu kvadratića oslonjenih na simbol akcije.

Pri preciznom crtanju dijagrama aktivnosti, izlazni parametri jedne akcije moraju da odgovaraju ulaznim parametara naredne akcije. Ukoliko to nije slučaj, neophodno je definisati *transformaciju* parametara. Ona ukazuje na postupak dobijanja ulaznih parametara naredne akcije na osnovu izlaznih parametara prethodne akcije. Transformacija ne sme da proizvodi sporedne efekte.

Na slici 5.21 prikazan je primer korišćenja nožica.



Slika 5.21 Primer upotrebe nožica

Primer opisuje aktivnost otkazivanja lekarskog pregleda. Prilikom zakazivanja pregleda, definiše se objekat koji sadrži osnovne informacije o pregledu: identifikacioni broj pregleda, ime pacijenta, termin pregleda, itd. U slučaju da pregled treba da bude otkazan, potrebno je o tome obavestiti pacijenta odgovarajućom porukom. Otkazivanje pregleda se u dijagramu aktivnosti modeluje pomoću dve akcije: *Otkazi pregled* i *Obavesti pacijenta*. Prva akcija ima jedan izlazni parametar, *Pregled*, koji sadrži informacije o pregledu. Druga akcija ima dva ulazna parametra: poruku koju treba poslati i ime pacijenta. Pošto se parametri ovih akcija razlikuju, definisane su transformacije koje iz skupa informacija o pregledu izdvajaju samo one koje je potrebno proslediti akciji *Obavesti pacijenta*.

U mnogim aktivnostima javlja se situacija u kojoj nakon neke akcije treba više puta izvršiti neku drugu akciju. Najbolji način za modelovanje ove situacije je korišćenje oblasti primene.

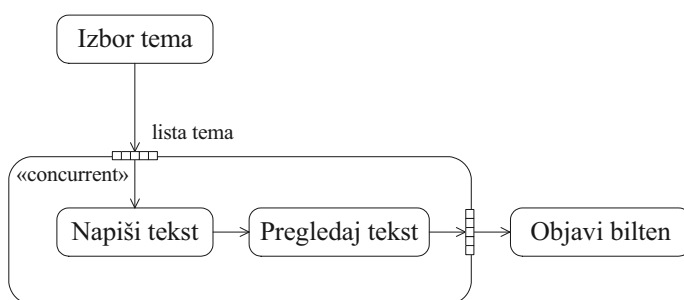
Oblast primene predstavlja deo dijagrama aktivnosti u kome se akcije izvršavaju po jednom za svaki element ulazne kolekcije. Svako pojedinačno izvršavanje akcija proizvodi odgovarajući element izlazne kolekcije. Oblast primene se napušta nakon kompletiranja izlazne kolekcije. Broj elemenata u ulaznoj kolekciji može, ali ne mora da odgovara broju elemenata u izlaznoj

kolekciji. Ako su ovi brojevi elemenata različiti, oblast primene se ponaša kao filter.

U oblasti primene, elementi ulazne kolekcije se mogu obrađivati na dva načina:

- paralelnom obradom, u kojoj se elementi obrađuju istovremeno (označava se rezervisanom rečju *concurrent*)
- iterativnom obradom, u kojoj se elementi obrađuju jedan za drugim

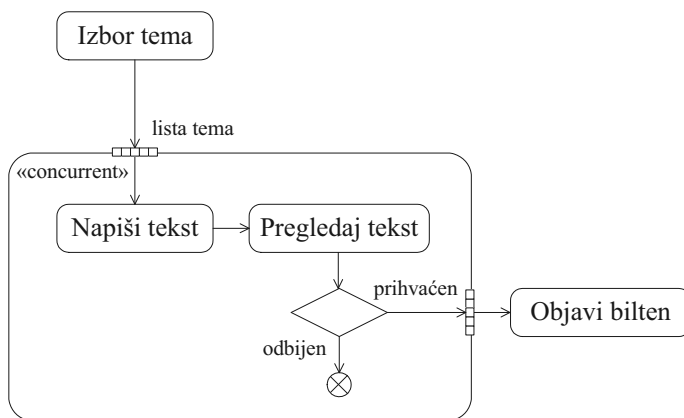
Na slici 5.22 dat je primer korišćenja oblasti primene.



Slika 5.22 Primer upotrebe oblasti primene

Primer modeluje aktivnost objavljivanja biltena. Da bi se objavio bilten, potrebno je na zadate teme napisati odgovarajuće članke, pregledati ih i od prihvaćenih tekstova sačiniti bilten. Skup tema predstavlja ulaznu kolekciju. Elementi (pojedinačne teme) ulazne kolekcije obrađuju se paralelno, zato što članke pišu različiti autori. Svaka tema se obrađuje kroz dve akcije: *Napiši tekst* i *Pregledaj tekst*. One se nalaze u oblasti primene (ovalni pravougaonik) zato što se ponavljaju više puta. Svaki prihvaćeni članak uvršćuje se u izlaznu kolekciju. Dakle, u ovom primeru brojevi elemenata u ulaznoj i izlaznoj kolekciji ne moraju biti jednaki. Kada se kompletira izlazna kolekcija, prelazi se na akciju objavljivanja biltena.

U dijagramu aktivnosti postoji mnoštvo tokova. Svaki tok može da se prekine *završetkom toka*, bez prekidanja cele aktivnosti. Zahvaljujući završetku toka, oblasti primene ostvaruju svoju filterarsku ulogu. Na slici 5.23 prikazano je kako se u prethodnom primeru objavljivanja biltena postiže filtriranje.



Slika 5.23 Primer upotrebe završetka toka

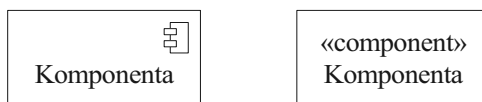
Kao što se vidi, ukoliko se članak ne prihvati za objavljivanje, tok se završava.

5.5 Dijagrami komponenata

Dijagrami komponenata opisuju fizičku organizaciju softverskih komponenata u sistemu i zavisnosti koje postoje između njih. Oni daju statički pogled na implementaciju sistema. Komponente mogu biti biblioteke, paketi, datoteke, i sl. Jednim dijagramom se obično ne može predstaviti ceo sistem, pa se zato pravi kolekcija ovih dijagrama. Ova vrsta dijagrama se najčešće koristi u fazi implementacije, ali je korisna i prilikom održavanja sistema.

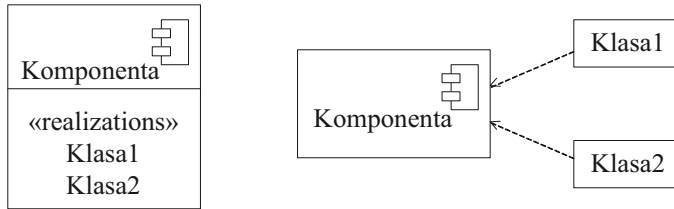
Na dijagramima komponenata se pojavljuju *stvari* i *relacije*. Stvari čine komponente, klase, atrefakti, interfejsi, portovi, podsistemi, dok relacije obuhvataju zavisnosti, generalizacije, asocijacije, realizacije.

Komponenta je apstrakcija fizičkog, zamenljivog dela sistema. Ona kapsulira neki sadržaj koji je okruženju vidljiv samo kroz interfejse. Na slici 5.24 data je grafička notacija komponente (verzija UML 2) u dve varijante.



Slika 5.24 Grafički prikaz komponente

Komponenta može da sadrži deo u kome su navedene klase koje realizuje, kao što je prikazano na slici 5.25.



Slika 5.25 Komponenta sa klasama koje realizuje

Između komponente i klase postoje sličnosti i razlike. Sličnosti su u tome što obe imaju imena, realizuju interfejsse i imaju elemente zavisnosti generalizacije i asocijacije veza. Razlike su u tome što je klasa logička apstrakcija, dok je komponenta fizička realizacija (u svetu bitova). Komponente predstavljaju fizičko pakovanje različitih logičkih apstrakcija. Osim ovoga, klase imaju attribute i operacije, a komponente samo operacije koje su dostupne preko interfejsa.

Mnoge vrste sistemskog softvera (na primer, operativni sistemi ili programski jezici) podržavaju koncept komponentata. Primeri komponentata su *JavaBean*, *EJB*, *CORBA*, *.NET assembly*, itd.

U UML-u su definisani sledeći standardni stereotipovi za komponente:

- *executable* – označava komponentu koja se može izvršavati
- *library* – označava statičku ili dinamičku objektnu biblioteku
- *file* – označava datoteku sa proizvoljnim sadržajem
- *document* – označava dokument
- *script* – označava skript
- *source* – označava datoteku sa izvornim kôdom

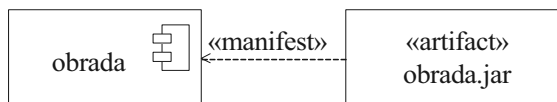
Artefakt je fizička informacija koja nastaje ili se koristi u procesu razvoja, procesu izvršavanja programa, ili pri isporuci softvera. Na primer, u procesu razvoja, nastaju artefakti u vidu modela, izvornog kôda, raznih dokumenata, a koriste se resursi. Prilikom izvršavanja programa, mogu da nastanu objekti kreirani iz DLL-a. Pri isporuci, artefakti mogu biti datoteke sa ekstenzijom .dll, .exe, .jar, razne tabele i sl.

Primer grafičkog prikaza artefakta (u dve varijante) dat je na slici 5.26.



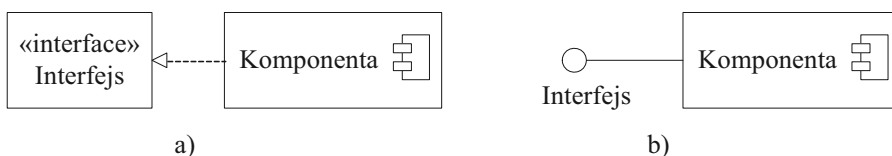
Slika 5.26 Primer artefakta

Artefakt može da predstavlja manifestaciju komponente, kao što je prikazano na slici 5.27.



Slika 5.27 Artefakt kao manifestacija komponente

Interfejs je skup operacija koji specificira servise komponente. Jedna komponenta može da realizuje jedan ili više interfejsa. Interfejs se na dijagramu komponenata predstava u vidu kanoničke (slika 5.28 a)) ili skraćene forme (slika 5.28 b)).

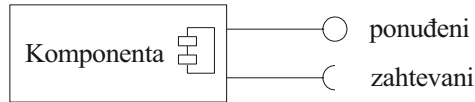


Slika 5.28 Grafički prikaz interfejsa

Komponente, kao osnovni gradivni elementi softverskog sistema, treba da budu slabo povezane, kako se izmena u jednoj komponenti ne bi odražavala na ostatak sistema. Da bi se to postiglo, uvedeno je da se komponentama može pristupati samo preko interfejsa. Interfejsi, u stvari, odvajaju implementaciju od ponašanja komponente. Jedna komponenta može, bez ikakvih popravki, da bude zamenjena drugom komponentom sa istim interfejsom.

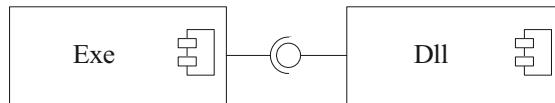
Postoje dve vrste interfejsa: eksport i import interfejsi. *Eksport interfejs* je interfejs koji komponenta realizuje. On obezbeđuje servis drugih komponenata. Za razliku od njega, *import interfejs* predstavlja interfejs koji komponenta koristi i on omogućava njenu nadgradnju. Zato se eksport interfejs naziva i ponuđenim, a import interfejs, zahtevanim interfejsom. *Ponuđeni interfejs* pokazuje koje servise komponenta nudi drugim komponentama. *Zahtevani interfejs* ukazuje na servise koji su datoj komponenti potrebni od strane drugih komponenata za njeno

funkcionisanje. Komponenta može da ima više interfejsa obe vrste. Na slici 5.29 prikazane su obe vrste interfejsa.



Slika 5.29 Vrste interfejsa

Kombinacijom grafičkih oznaka za ponuđeni (loptica) i zahtevani (postolje) interfejs, dobija se veznik sklopa (*assembly connector*). Primer interakcije ostvarene na ovaj način dat je na slici 5.30.



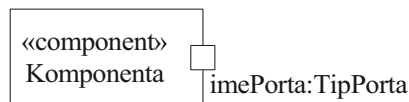
Slika 5.30 Primer primene veznika sklopa

Zavisnosti opisuju veze između komponenata i na dijagramu se predstavljaju usmerenom isprekidanom linijom, kao na slici 5.31. Prema slici, komponenta *Modul 1* koristi neke servise koji su obezbeđeni u komponenti *Modul 2*.



Slika 5.31 Zavisnosti između komponenata

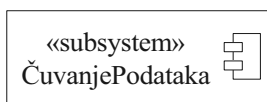
Port predstavlja tačku interakcije sa okruženjem. Komponenta može da ispoljava interfejs preko porta. Naime, portu se mogu pridružiti interfejsi koji specificiraju prirodu interakcije komponente sa okruženjem. Na dijagramu komponenata, port se grafički predstavlja kao na slici 5.32.



Slika 5.32 Grafički prikaz porta

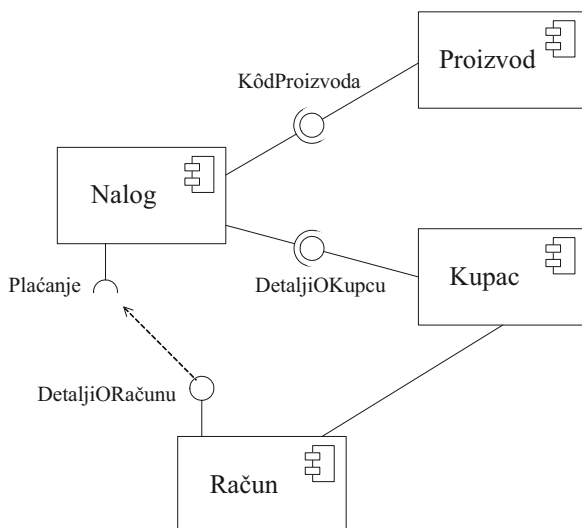
Podsistemi predstavljaju fizičke podsisteme u sistemu. Oni mogu da sadrže komponente ili druge podsisteme. Najčešće reprezentuju direktorijume u sistemu

datoteka. Grafički prikaz podsistema koji obuhvata komponente za čuvanje podataka dat je na slici 5.33. Sličan je prikazu komponente, s tom razlikom što se koristi druga rezervisana reč.



Slika 5.33 Grafički prikaz podsistema

Primer dijagrama komponenata prikazan je na slici 5.34. U datom sistemu postoje četiri komponente: *Proizvod*, *Kupac*, *Nalog* i *Račun*. Eksport interfejse realizuju komponente *Proizvod* i *Kupac*, a import interfejse, komponenta *Nalog*. Zavisnošću se mapiraju detalji koji se odnose na račun pridružen kupcu sa import interfejsom *Plaćanje* koga realizuje komponenta *Nalog*.



Slika 5.34 Primer dijagrama komponenata

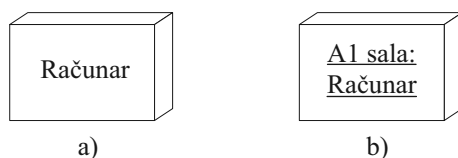
5.6 Dijagrami raspoređivanja

Dijagrami raspoređivanja opisuju fizičku organizaciju sistema, prikazujući njegovu hardversku i softversku arhitekturu. Na njima je jasno navedeno koje hardverske i softverske komponente postoje u sistemu, kao i koja softverska

komponenta se izvršava na kojoj hardverskoj komponenti. Takođe su definisane i veze između različitih delova sistema.

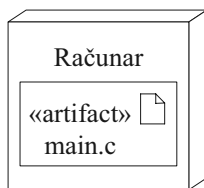
Osnovni elementi dijagrama su čvorovi, artefakti i veze.

Čvor je apstrakcija fizičkog objekta koji predstavlja resurs obrade. To može da bude neki uređaj (na primer, računar ili mobilni telefon) ili neko izvršno okruženje, tj. softver koji opslužuje neki drugi program (na primer, operativni sistem, kontejnerski procesi, vituelna mašina i sl.). Čvorove često predstavljaju web serveri, serveri baze podataka, aplikativni serveri, itd. Čvorovi se na dijagramu raspoređivanja prikazuju u vidu kutija sa imenom čvora. Oni mogu da imaju i instance, što se naznačava podvlačenjem imena instance. Na slici 5.35 pod a) je dat primer čvora, dok je pod b) prikazana instanca čvora koja predstavlja određeni računar.



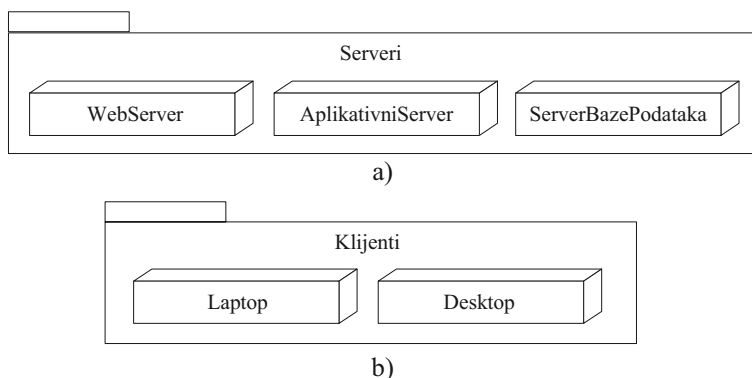
Slika 5.35 Primer čvora i instance čvora

Artefakti su softverske komponente koje se izvršavaju u čvorovima. To su obično datoteke, koje mogu biti izvršne (.exe, .dll, .jar, itd), datoteke sa podacima, konfiguracione datoteke, HTML dokumenti i dr. Artefakti se na dijagramu predstavljaju u vidu pravougaonika unutar čvorova (kutija) na koje su alocirani, na način prikazan na slici 5.36.



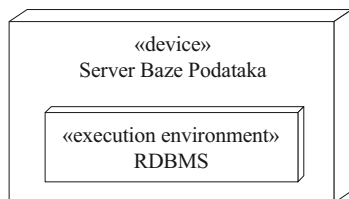
Slika 5.36 Primer artefakta

Često se dešava da više fizičkih čvorova obavljaju logički srodne poslove. Oni se mogu grupisati u pakete. Na slici 5.37 prikazani su paketi servera (pod a)) i klijenata (pod b)) koji se koriste u sistemu.



Slika 5.37 Primeri paketa čvorova

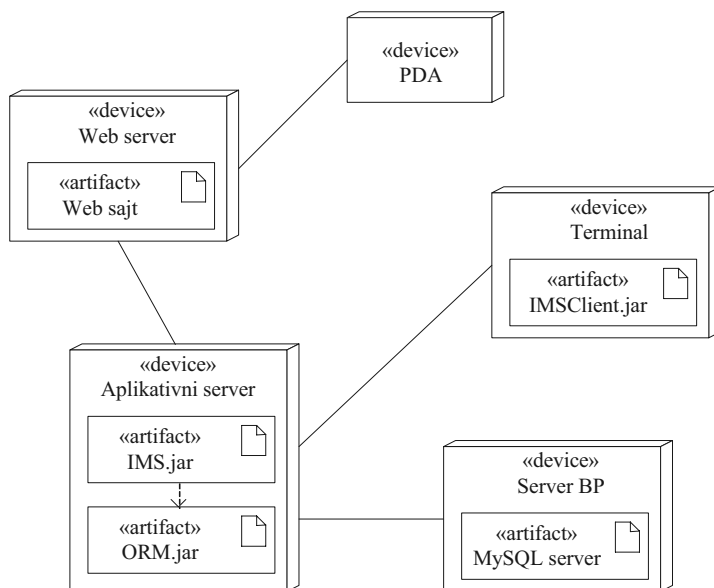
Čvorovi mogu da budu i ugnježdeni, tj. jedan čvor može da sadrži drugi. To se na dijagramu predstavlja kao na slici 5.38.



Slika 5.38 Ugnježdeni čvorovi

Veze predstavljaju komunikacione putanje između različitih delova sistema. Za njihovo prikazivanje na dijagramu raspoređivanja, koriste se asocijacije.

Primer dijagrama raspoređivanja dat je na slici 5.39. On modeluje arhitekturu web aplikacije kojoj može da se pristupa preko PDA (*Personal Digital Assistant*) uređaja za upravljanje ličnim informacijama. Aplikativni server je povezan sa MySQL bazom podataka.



Slika 5.39 Primer dijagrama raspoređivanja

6 Implementacija softvera

Implementacija ili kodiranje podrazumeva softversku realizaciju rešenja razmatranog problema. Rezultat implementacije je skup programa koji ispunjavaju zadate funkcionalne zahteve. Osnovu za implementaciju predstavlja projekat programa. Međutim, sistem se uvek može realizovati na mnogo različitih načina: mogu se koristiti različiti programski jezici, razvojna okruženja, hardverske platforme, itd. Cilj ovog poglavlja je da ukaže na opšte principe implementacije, tj. na praktične elemente softverskog inženjerstva koje treba imati na umu prilikom pisanja programa.

Iako na prvi pogled proces implementacije izgleda jednostavan, jer je sve ranije definisano modelima, obično to nije tako. Najpre, treba imati u vidu da postoji mogućnost da projektanti nisu uzeli u obzir sve specifičnosti platforme ili razvojnog okruženja koji će biti korišćeni. Drugo, strukture podataka je ipak jednostavnije predstaviti tabelama ili graficima nego napisati ekvivalentan programski kôd. Treće, programi moraju da budu razumljivi ne samo onima koji su ih pisali, već i drugima koji učestvuju u procesu razvoja (onima koji testiraju sistem, onima koji ga održavaju i dr.). Četvrto, treba težiti ka tome da se napisani kôd može jednostavno ponovo iskoristiti, ukoliko se za to ukaže prilika.

U zavisnosti od vrste projekta, proces implementacije može da bude različit. Programeri mogu da budu u prilici da analiziraju postojeći tuđi kôd kako bi ga izmenili, dogradili, ispravili uočene greške, ili ga iskoristili u sklopu neke druge aplikacije. Takođe, mogu i da analiziraju ili generišu sopstveni kôd u okviru realizacije nekog novog modula. Bez obzira na okolnosti u kojima rade, programeri su uvek deo tima koji razvija softver. To znači da je neophodno obezbediti visok nivo saradnje i koordinacije između svih članova tima kako bi softverski proizvod bio kvalitetan.

U razvoju softvera moraju se poštovati standardi koji važe u konkretnom okruženju. Mnoge kompanije zahtevaju da programski kôd bude usklađen po pitanju stila, formata, sadržaja i drugih parametara, kako bi sâm kôd i prateća dokumentacija bili jasni svakome ko ih čita. Ovakav pristup ima brojne prednosti, kao što su:

- primena standarda i procedura pomažu programerima da uspostave sistematičnost u svom radu
- standardizovana dokumentacija omogućava lakše i brže tumačenje kôda
- olakšano je pronalaženje grešaka u kôdu
- novonastale izmene se mogu jednostavnije uneti u sistem, jer je jasno koji deo kôda implementira koju funkcionalnost
- strukturiranjem kôda prema standardima održava se usklađenost elemenata iz dizajna sa elementima iz implementacije
- olakšan je nastavak rada na projektu ukoliko je iz nekog razloga projekat morao da bude prekinut na neko vreme

U procesu implementacije najvažnije je poštovati standard koji se tiče usklađenosti elemenata iz projekta sa elementima iz implementacije. Ukoliko se modularnost pristupa iz faze projektovanja ne prenese na fazu implementacije, onda dolazi do degradacije celog projekta. Osobine projekta, kao što su slaba spregnutost komponenata, ili dobro definisane veze između komponenata treba da budu i osobine napisanih programa. Samo na taj način je moguće lako praćenje implementiranih algoritama, struktura podataka i funkcija unutar kôda u smislu njihove usklađenosti sa projektom.

Tokom pisanja svog dela kôda, programer treba da vodi računa o tome da će njegov kôd najverovatnije koristiti druge osobe na različite načine. Na primer, ako programer nije član tima za testiranje, neko drugi će testirati njegov kôd. Ili, možda će neko drugi integrisati njegov modul sa ostalim programima u sistemu. Čak i u slučaju da je napisani kôd samostalan, može se javiti potreba za nekim izmenama, bilo zbog neke greške, ili zbog promena koje zahteva kupac. U svakom slučaju, bitno je da kôd bude dobro organizovan, jasan i dobro dokumentovan odgovarajućim komentarima.

6.1 Pisanje programa

Pisanje programa je kreativan proces u kome programer prevodi opise sistema date u projektu u programski kôd na konkretnom programskom jeziku. Pri implementaciji, programer ima veliku slobodu, jer se ista funkcionalnost uvek može realizovati na različite načine. Ponekad, specificirani zahtevi ili projekat uslovljavaju izbor programskog jezika.

Da bi se postiglo da programski kôd odražava dizajn sistema, potrebno je poštovati nekoliko opštih smernica za pisanje programa:

- *lokalizacija ulaza i izlaza*
- *korišćenje pseudokôda*
- *revidiranje kôda*
- *višekratna upotrebljivost*

Pod lokalizacijom ulaza i izlaza podrazumeva se izdvajanje delova programskog kôda koji se odnose na učitavanje ulaznih podataka ili generisanje izlaznih veličina u posebne specijalizovane komponente u okviru sistema. Na taj način se dobija sistem koji se lakše može modifikovati, ukoliko se za to ukaže potreba. Ove specijalizovane komponente odražavaju osobine trenutno korišćenog hardvera i softvera. Ukoliko dođe do promene hardvera ili softvera, najverovatnije će biti potrebno da se i komponente koje opisuju ulaze i izlaze izmene, pa je zato dobro da budu izdvojene od ostatka sistema.

Lokalizacija vodi uopštavanju sistema. Osim unosa podataka, ulaznim komponentama mogu se dodati i druge funkcionalnosti, kao što su provera ispravnosti unetih podataka, formatiranje unetih podataka i sl. Na ovaj način se ostale komponente u sistemu rasterećuju poslova vezanih za ulaze. Slično ovome, objedinjavanje izlaznih funkcija u jednu komponentu vodi lakšem razumevanju i menjanju sistema.

Prilikom projektovanja neke komponente, njen dizajn ne mora da bude strogo povezan sa nekim programskim jezikom, ili konkretnom strukturom podataka kojom će ona biti predstavljena. Dizajn je obično samo okvir koji ukazuje na to šta ta komponenta treba da radi. Da bi se dizajn pretočio u kôd, potrebni su znanje i individualna kreativnost programera. Da ne bi došlo do loše realizacije, korisno je da se od dizajna ka kôdu ne ide naglo, već u fazama.

Pre konačne realizacije komponente u izabranom programskom jeziku, dobro je ispitati više alternativa kako bi se utvrdilo koja je implementacija najbolja. Za to se može koristiti pseudokôd. Pod pseudokôdom se podrazumeva kôd napisan običnim jezikom koji se koristi u datoj oblasti. Pseudokôd je slobodnijeg stila, i u

njemu se, osim opisa, često koriste i sintaksni elementi postojećih programskih jezika (zgrade, strukture za kontrolu toka, itd). Sledi primer pseudokôda koji realizuje funkciju za računanje faktoriijela.

```
funkcija faktoriijel (ceo broj n)
{
    ceo broj rezultat = 1
    dok je n veće od 1 {
        rezultat = rezultat * n
        smanjiti n za 1
    }
    vratiti rezultat
}
```

Zbog svoje deskriptivnosti, pseudokôd se jednostavno i brzo analizira i menja, pa se njegovom primenom značajno štedi vreme potrebno za donošenje odluke o tome kako će komponenta zaista biti realizovana. Tokom analize različitih varijanti pisanih pseudokôdom, mogu se uočiti i korisne izmene u dizajnu. O tome treba obavestiti projektante, kako bi se dobila njihova saglasnost o predloženim izmenama. Na ovaj način se održava usklađenost između dizajna i kôda. Pseudokôd predstavlja dobru osnovu za lako pisanje kôda na izabranom programskom jeziku.

Prilikom pisanja kôda najpre se pravi gruba skica, a zatim se taj kôd revidira i ponovo piše sve dok se ne postigne željeni rezultat. Ukoliko se programeru čini da je kôd nerazumljiv i zamršen, ili ne može da reši neke probleme (grananja, petlji i sl.), neophodno je da ponovo razmotri dizajn. Tako će utvrditi da li je problem u dizajnu (tada treba da konsultuje projektante) ili u njegovom prevođenju dizajna u kôd. Ako je problem u prevođenju, onda programer treba da preispita način predstavljanja i strukturiranja podataka, primenjene algoritme i dr.

Dobra praksa pri pisanju kôda je da se komponente realizuju tako da mogu ponovo da se primene, a takođe je dobro i da se koriste ranije implementirane komponente. Višekratno upotrebljive komponente ubrzavaju proces pisanja kôda i smanjuju broj potencijalnih grešaka (ranije realizovane komponente su već istestirane).

Prilikom generisanja višekratne komponente, treba voditi računa o sledećem:

- komponenta treba da bude opšta i da radi u što fleksibilnijim uslovima
- u okviru komponente treba izdvojiti delove podložne promenama (zbog promene radnih uslova) od delova koji se najverovatnije nikada neće menjati

- interfejs komponente treba da bude opšti i dobro definisan
- pri imenovanju veličina treba primenjivati jasne i logički intuitivne konvencije
- neophodno je dokumentovati strukture podataka i algoritme korišćene pri izgradnji komponente
- potrebno je navesti informacije o otkrivenim i ispravljenim greškama

Da bi se ranije implementirana komponenta ponovo koristila, potrebno je proveriti sledeće:

- da li ona obavlja funkciju koja je potrebna
- ako je potrebna manja izmena komponente, da li je bolje uraditi tu promenu ili napraviti novu komponentu
- da li je komponenta dobro dokumentovana, tako da je potpuno jasna njena funkcija
- da li postoji evidencija o testiranju i revizijama komponente koja bi garantovala da u njoj nema grešaka

Pri ugrađivanju višekratne komponente, treba proceniti i koliko kôda bi trebalo napisati da bi sistem mogao da sarađuje sa ovom komponentom.

Bez obzira na korišćeni programski jezik, realizacija svake programske komponente zahteva utvrđivanje struktura podataka, algoritama i kontrolnih struktura koji će biti primenjeni.

6.1.1 Strukture podataka

Pre pisanja programa, potrebno je uraditi neke pripremne radnje kako bi se sâm proces pisanja pojednostavio, a napisani kôd bio efikasniji. Jedna od njih je osmišljavanje struktura podataka u kojima će se čuvati podaci bitni za rad komponente koju program realizuje. Strukture podataka se biraju tako da se podacima u njima lako upravlja i manipuliše. Nakon utvrđivanja struktura podataka koje će biti korišćene u realizaciji, program se organizuje u skladu sa njima.

Izbor struktura podataka može se preuzeti iz dizajna, tj. projekta (ukoliko su tamo definisane odgovarajuće strukture), ili se može napraviti u fazi implementacije na osnovu manipulacija koje treba izvršiti nad podacima u okviru komponente.

Izabrane strukture podataka direktno utiču na složenost i efikasnost programa. Na primer, pretpostavimo da komponenta računa porez na dohodak. Ulazni podaci u komponentu su iznos dohotka koji treba da se oporezuje i sledeća pravila:

- a) Za prvih 100000 din. dohotka, poreska stopa iznosi 10%.
- b) Za sledećih 100000 din. dohotka (iznad 100000 din.), poreska stopa iznosi 12%.
- c) Za sledećih 100000 din. dohotka (iznad 200000 din.), poreska stopa iznosi 15%.
- d) Za sledećih 100000 din. dohotka (iznad 300000 din.), poreska stopa iznosi 18%.
- e) Za dohodak iznad 400000 din., poreska stopa iznosi 20%.

Na osnovu datih pravila sledi da neko ko je imao oporezovani dohodak u iznosu od 350000 din. plaća 10% od prvih 100000 din. (što iznosi 10000 din.), zatim 12% od sledećih 100000 din. (12000 din.), 15% od sledećih 100000 din. (15000 din.) i 18% od preostalih 50000 din. (9000 din.), što ukupno iznosi 46000 din.

Kôd u programskom jeziku C koji realizuje komponentu za računanje poreza na dohodak može da ima sledeći izgled:

```
int racunanje_poreza(int dohodak) {
    int porez = 0;

    if (dohodak == 0) return porez;
    if (dohodak > 100000) porez = porez + 10000;    // 10% na prvih 100000 din.
    else {
        porez = porez + 0.1 * dohodak;
        return porez;
    }
    if (dohodak > 200000) porez = porez + 12000;    // 12% na drugih 100000 din.
    else {
        porez = porez + 0.12 * (dohodak - 100000); // 12% na preko 100000 din.
        return porez;
    }
    if (dohodak > 300000) porez = porez + 15000;    // 15% na trecih 100000 din.
    else {
        porez = porez + 0.15 * (dohodak - 200000); // 15% na preko 200000 din.
        return porez;
    }
    if (dohodak > 400000)
        // 18% na cetvrtih 100000 din. i 20% na preko 400000 din.
        porez = porez + 18000 + 0.2 * (dohodak - 400000);
    else
        porez = porez + 0.18 * (dohodak - 300000); // 18% na preko 300000 din.
    return porez;
}
```

Neka je definisana tabela sa podacima o poreskim stopama u zavisnosti od intervala u kome se nalazi iznos oporezovanog dohotka, kao što je prikazano u tabeli 6.1.

Donja granica opsega (dgo)	Stopa (s)
0	0.10
100000	0.12
200000	0.15
300000	0.18
400000	0.20

Tabela 6.1 Tabela sa podacima o poreskim stopama

Algoritam za računanje poreza može se znatno uprostiti ukoliko se koristi data tabela:

```
int racunanje_poreza(int dohodak) {
    int porez = 0;
    if (dohodak == 0) return porez;
    for (i = 0; i < 4; i++) {
        if (dohodak > dgo[i+1]) {
            porez = porez + s[i]*100000;
            if (i == 3) porez = porez + s[i+1] * (dohodak - dgo[i+1]);
        }
        else {
            porez = porez + s[i] * (dohodak - dgo[i]);
            return porez;
        }
    }
    return porez;
}
```

Usvojene strukture podataka, u nekim slučajevima, mogu da utiču i na izbor programskog jezika. Na primer, programski jezik LISP je projektovan za rad sa listama, za predikatski račun pogodan je Prolog, dok Ada i Eiffel omogućavaju obradu nedozvoljenih stanja, tj. izuzetaka. Pascal je pogodan za implementaciju rekurzivnih struktura podataka, kao što su stabla.

6.1.2 Algoritmi

U projektu programa obično su definisani algoritmi koje treba upotrebiti za realizaciju date komponente. To mogu, na primer, da budu algoritmi za sortiranje,

optimizaciju, višekriterijumsko odlučivanje, itd. Iako su algoritmi poznati, postoji velika fleksibilnost u pogledu njihove realizacije, zavisno od ograničenja koje postavljaju korišćen programski jezik i raspoloživa hardverska platforma. Različito realizovani algoritmi imaju različite performanse, kao i efikasnost.

Prilikom implementacije algoritama, većina programera pokušava da generiše kôd koji zadati algoritam izvršava što je moguće većom brzinom. Iako je brzina važan pokazatelj od koga svakako zavise performanse komponente, ipak treba obratiti pažnju i na moguće posledice ovog ubrzanja:

- brže izvršavanje algoritma može da prouzrokuje složeniji kôd koji zahteva više vremena za generisanje
- složeniji kôd zahteva više primera za testiranje za koje treba obezbediti odgovarajuće ulazne podatke
- potrebno je više vremena za tumačenje i razumevanje kôda što je on složeniji
- buduće potencijalne izmene je teže sprovesti ako je kôd složeniji

Imajući u vidu navedeno, pitanje vremena potrebnog za izvršavanje algoritma trebalo bi razmatrati zajedno sa postavljenim zahtevima i kvalitetom urađenog projekta. Posledica ubrzanja ne sme da bude smanjenje jasnoće i ispravnosti rada komponente.

Ukoliko je brzina značajan faktor u implementaciji, potrebno je detaljno proučiti na koji način prevodilac za izabrani programski jezik optimizuje kôd. Na taj način se izbegava situacija da optimizacija koju je programer primenio u cilju ubrzanja, u stvari uspori naizgled brži kod. Na primer, pretpostavimo da treba napisati kôd koji implementira trodimenzionalni niz. Da bi ubrzao postupak, programer može da koristi jednodimenzionalni niz i sâm izračunava indeks, tj. poziciju elementa u trodimenzionalnom nizu. Neka se *indeks*, na primer, računa kao:

$$\text{indeks} = 2*i + 4*j + k;$$

Dakle, u programu se svaka pozicija u trodimenzionalnom nizu računa množenjem i sabiranjem nekih vrednosti. Međutim, može se desiti da prevodilac, da bi skratio vreme, za indeksiranje niza koristi registre, a ne izračunavanja. U tom slučaju, korišćenje jednodimenzionalnog niza bi produžilo vreme izvršenja, iako je polazna ideja bila da ga smanji.

6.1.3 Kontrolne strukture

Kontrolne strukture upravljaju tokom izvršavanja programa. Veliki broj kontrolnih struktura je već predložen u dizajnu sistema. Nezavisno od primenjene arhitekture, prilikom prevođenja dizajna u programski kôd treba očuvati što je moguće više kontrolnih struktura, jer je tada lakše pratiti usklađenost dizajna i kôda.

Prilikom pisanja kôda, preporučljivo je da se koriste one kontrolne strukture koje omogućavaju lako čitanje kôda odozgo nadole. To znači da treba izbegavati velike skokove sa jednog na drugo mesto u programu, praćeno obeležavanjem mesta za povratak, jer se onda uvek postavlja pitanje da li se ide pravom putanjom. U tumačenju kôda, primarno treba da bude šta program radi, a ne sâm tok kontrole.

Kontrolne strukture mogu značajno da utiču na razumljivost programa. Na primer, neka je dat program:

```

dobit = d;
if (prihod < 20000) goto A;
dobit = d + 5*nagrada;
goto C;
A: if (prihod < 15000) goto B;
dobit = d + 2*nagrada;
goto C;
B: if (prihod < 10000) goto C;
dobit = d + nagrada;
C: return dobit;
```

Kao što se vidi, u ovom programu ima mnogo skokova, što ga čini nepreglednim, pa je teško pratiti njegovo izvršavanje. Isti rezultat se može postići primenom drugačije strukture toka na sledeći način:

```

if (prihod < 10000) dobit = d;
elseif (prihod < 15000) dobit = d + nagrada;
elseif (prihod < 20000) dobit = d + 2*nagrada;
else dobit = d + 5*nagrada;
```

Iako se pri pisanju programa uvek teži da on bude čitljiv odozgo nadole, to nije uvek moguće postići. Na primer, čest je slučaj da izlazak iz petlje naruši ovakav poredak. Međutim, kad god je to moguće, korisno je da naredna akcija bude što bliže uslovu koji do nje vodi.

Pri pisanju kôda treba voditi računa o tome da on ne bude previše specijalizovan i prilagođen konkretnoj primeni, već da ima opštiji karakter. Na primer, neka je potrebno napraviti komponentu koja ispituje da li se u prvih 10

karaktera nekog teksta pojavljuje „*“. Ova komponenta može da se realizuje kao funkcija čiji je jedini argument tekst, a onda se u telu funkcije proverava prvih 10 karaktera i vraća odgovarajuća logička vrednost. Međutim, ovako realizovana funkcija teško da bi mogla da se primeni u bilo kojoj drugoj situaciji. Stoga je bolje da realizacija navedene funkcije bude opštijeg karaktera, tj. da, osim argumenta koji predstavlja tekst, ima još dva ulazna parametra: broj karaktera u okviru kojih se obavlja pretraživanje i sâm karakter koji se traži. Ovako realizovana funkcija bi mogla kasnije da se koristi za pretraživanje delova teksta proizvoljne dužine (zadate argumentom koji predstavlja broj karaktera) i za proizvoljan znak (zadat argumentom koji predstavlja karakter). Ipak, težnja ka uopštavanju ne sme da ugrozi performanse komponente ili razumljivost kôda.

Kao i u fazi projektovanja, i u fazi implementacije, modularnost predstavlja poželjnu osobinu. Izgradnjom programa u modularnim blokovima dobija se razumljiviji sistem koji se lakše testira i održava. Svaka komponenta je odgovorna za svoj deo posla (koji je skriven za ostatak sistema), dok ostale komponente samo koriste rezultat njenog rada. Modularne komponente se mogu ponovo iskoristiti u drugim aplikacijama, a njihove izmene i eventualna prilagođenja su lokalnog karaktera. Između komponenata treba da postoje dobro definisane veze. Dobra praksa u pisanju kôda je da postoji konzistentnost u imenovanju parametara kojima se komponente povezuju (izlazni parametar jedne komponente je ulazni parametar druge komponente). Ovakvim pristupom, zavisnost između komponenata postaje vidljivija.

6.2 Programska dokumentacija

Pod programskom dokumentacijom podrazumeva se skup opisa u tekstualnoj formi kojima je objašnjeno šta program radi i na koji način. Postojanje odgovarajuće programske dokumentacije omogućava kasnije korišćenje, održavanje ili unapređenje softvera.

Programska dokumentacija obuhvata, uslovno rečeno, unutrašnju i spoljašnju dokumentaciju. Pod unutrašnjom dokumentacijom podrazumevaju se opisi pridruženi programskom kôdu u vidu komentara i oni se nalaze u datotekama sa programima. Spoljašnja dokumentacija sadrži sve ostale opise (van datoteka) koji se odnose na dati sistem.

6.2.1 Unutrašnja dokumentacija

Unutrašnja ili interna dokumentacija ima veliki značaj, ne samo za programera koji je pisao dati program, već i za programere koji će ga u budućnosti koristiti. Ona je namenjena svakome ko čita napisani kôd, jer sadrži informacije koje treba da pomognu da se kôd lakše shvati i protumači.

Da bi programski kôd bio čitljiv, mora, pre svega, da bude jasno i pregledno napisan. To se postiže sistematičnim pristupom pisanju koji podrazumeva uvlačenje redova u zavisnosti od mesta naredbe u hijerarhiji programa. Na primer, kôd koji sledi ne prati navedenu preporuku:

```
if (prihod < 1000000)
    rezultat = 1;
elseif (prihod == 1000000)
    if (cena < 50)
        rezultat = 2;
    else rezultat = 3;
    else rezultat = -1;
```

Kao što se vidi, dati kôd nije lak za praćenje. Isti kôd se može napisati na mnogo jasniji i čitljiviji način:

```
if (prihod < 1000000)
    rezultat = 1;
elseif (prihod == 1000000)
    if (cena < 50)
        rezultat = 2;
    else rezultat = 3;
    else rezultat = -1;
```

Dobra praksa je da se na početku svakog dokumenta sa kôdom ispiše komentar u vidu zaglavlja u kojem se opisuje funkcionalnost programskog kôda koji sledi, njegove veze sa okruženjem, očekivani ulazni i izlazni podaci. Zaglavlje uglavnom sadrži sledeće informacije:

- ulogu komponente i njeno mesto u sistemu
- naziv komponente
- ime autora komponente
- datum kada je komponenta napisana ili revidirana
- način na koji se pristupa komponenti

Na početku zaglavlja bi trebalo ukratko opisati svrhu komponente i eventualno njeno mesto u sistemu, posmatrano u odnosu na druge komponente. Naziv komponente mora da bude uočljiv da bi se lakše ustanovilo gde se ona poziva u ostatku sistema. Ime autora ukazuje na to kome se treba obratiti ako ima nekih pitanja u vezi sa realizacijom date komponente. Komponente se često ažuriraju i revidiraju bilo zbog izmena u zahtevima, ili zbog ispravljanja uočenih grešaka. Bilo bi dobro da u dokumentaciji postoji hronologija izvršenih promena sa naznačenim vremenima i imenima autora koji su te promene izvršili. Važan deo zaglavlja čini objašnjenje kako se pristupa komponenti. Ono treba da sadrži informaciju o broju ulaznih i izlaznih parametara, uključujući i njihove tipove. Pogodno je za svaki parametar dati kratak opis koji objašnjava njegovu ulogu u komponenti. Osim ovoga, u dužim zaglavljima navode se i ime, tip i svrha svake veće strukture podataka ili promenljive, kratak opis logičkog toka i algoritama, načini obrade grešaka i dr.

Primer uobičajenog zaglavlja sa minimalnim skupom informacija dat je na slici 6.1.

```
*****
* Komponenta za sabiranje elemenata niza
* Ime komponente: SUBARR
* Programer: E.Maric
* Verzija: 1.0 (2.februar 2012.)
*
* Poziv procedure: CALL SUBARR(A, R)
* Ulazni parametar: numericki niz A ciji su elementi celi brojevi
* Izlazni parametar: R - zbir elemenata niza A
*
*****
```

Slika 6.1 Primer zaglavlja

Prednosti pisanja zaglavlja su sledeće:

- zaglavlje pruža informacije o funkcionalnosti kôda koji sledi, što naročito pomaže onima koji održavaju softver
- u situaciji kada se traži gotova komponenta koja bi se ponovo iskoristila, na osnovu zaglavlja može se zaključiti da li kôd koji sledi realizuje traženu komponentu
- u slučaju otkaza, zaglavlje daje dovoljno elemenata za procenu da li je uzrok otkaza u posmatranom dokumentu ili ne

Nakon zaglavlja, koje predstavlja uvod u program, slede dodatni komentari koji usmeravaju čitaoca u tumačenju programa. Oni mu pomažu da shvati kako su navodi iz zaglavlja implementirani u kôdu. Količina dodatnih komentara zavisi od procene sâmog programera. Ukoliko se u programu koriste ilustrativna imena promenljivih, ako su naredbe jasne i dobro strukturirane, program je sâm po sebi izvor informacija prilikom tumačenja koda. U tom slučaju, broj dodatnih komentara ne mora da bude veliki. Međutim, komentari imaju svoju ulogu čak i u dobro napisanom kôdu, jer se njima mogu opisati dodatne informacije koje nisu vidljive iz kôda. Na primer, ako kôd implementira neki postupak, u komentaru se može navesti odakle je taj postupak preuzet:

```
// Postupak obracunavanja cene je definisan u saradnji sa kupcem.  
int obracun(...) {  
    ....  
}
```

Osnovna uloga komentara je da daje korisnu informaciju o delu programskog kôda kome je pridružen. To znači da svaka promena u kôdu zahteva ažuriranje postojećeg komentara. Pod korisnom informacijom podrazumeva se informacija koja nije očigledna iz kôda, već daje neko važno objašnjenje. Na primer, komentar

```
// sabrati a sa 1  
a = a + 1;
```

je nepotreban, jer je njegov sadržaj očigledan iz kôda. Za razliku od ovog, komentar

```
// podesiti indeks narednog clana u nizu  
a = a + 1;
```

pruža važnu informaciju o tome da se očekuje korišćenje narednog člana niza.

Dobra praksa je da se komentari pišu istovremeno sa kôdom, jer će tada u njima biti najviše korisnih informacija. Naknadno pisanje komentara vodi ka izostanku nekih bitnih zamisli programera koje je on u međuvremenu zaboravio. Ukoliko je programeru teško da iskomentariše kôd, to je znak da je dizajn previše složen i da ga treba uprostiti.

Prilikom pisanja kôda, posebnu pažnju treba posvetiti izboru imena struktura podataka. Imena treba da opisuju ulogu strukture. Na primer, programski kôd

```
vrednost = cena_proizvoda * kolicina_proizvoda;
```

je mnogo jasniji od kôda

```
v = c * k;
```

tako da je za poslednji primer svakako potrebno napisati odgovarajući komentar.

Weinberg (1971.g.) preporučuje da se programski kôd i komentari ne mešaju, tj. da se u dokumentu formatiraju tako da se kôd nalazi na levom delu stranice, a komentari na desnom. Na primer, sledeći kôd je dobro iskomentarisan:

```
void zamena (int a[], int i, int j) {  
    int tmp = a[i];    // privremeno cuvanje kolicine proizvoda u i-tom magacinu  
    a[i] = a[j];       // prebacivanje proizvoda iz j-tog u i-ti magacin  
    a[j] = tmp;       // prebacivanje proizvoda iz i-tog u j-ti magacin  
}
```

6.2.2 Spoljašnja dokumentacija

Spoljašnja ili eksterna dokumentacija opisuje sistem sa opšteg aspekta i daje odgovore na pitanja ko, šta, kako, zašto, kada i gde u sistemu nešto radi. Za razliku od unutrašnje dokumentacije koja je prvenstveno namenjena programerima, spoljašnja dokumentacija je, osim programerima, namenjena i projektantima. Na osnovu spoljašnje dokumentacije, projektanti mogu da analiziraju sistem i predlažu njegove buduće izmene i unapređenja. Spoljašnja dokumentacija sadrži znatno šire i detaljnije opise od onih koji se mogu naći u komentarima u programu.

Pošto se softverski sistem sastoji od većeg broja međusobno povezanih komponentata, spoljašnja dokumentacija obično sadrži sledeće:

- pregled svih komponentata u sistemu
- podelu komponentata po grupama, ukoliko ima potrebe za tim (u složenijim sistemima mogu se izdvojiti komponente sa sličnom funkcijom, na primer, komponente korisničkog interfejsa, komponente za pristup bazama podataka, komponente za obradu podataka, ulazno/izlazne komponente, itd.)
- dijagrame kojima se opisuju pojedinačne komponente sa odgovarajućim tekstualnim objašnjenjima
- dijagrame iz kojih se jasno vidi kako se podaci koriste u sistemu, tj. koje komponente koriste koje podatke, kako ih razmenjuju i modifikuju
- opis klasa objekata i njihovu hijerarhiju nasleđivanja (ukoliko se primenjuje objektno-orijentisani pristup u rešavanju problema)

Spoljašnja dokumentacija komponente se piše u skladu sa strukturom komponente koja je već opisana u dizajnu sistema, uz dodavanje tekstualnih opisa o pojedinostima iz programskog kôda kojim je komponenta realizovana.

Na početku ove dokumentacije daje se opis problema koji komponenta rešava. Ovaj opis ne podrazumeva ponavljanje ranije postavljenih zahteva, već postavljanje nove osnove u smislu kada se komponenta poziva i zašto je ona potrebna. Zatim slede razmatranja različitih opcija mogućih rešenja problema, uz navođenje razloga zbog kojih je konkretno rešenje izabrano.

Nakon izbora rešenja, sledi opis algoritama koji se koriste u komponenti. U opisu se navode primenjene formule, postavljeni uslovi i ograničenja, reference na korišćenu literaturu, itd. Opisi algoritama su vrlo detaljni i uključuju analizu svih posebnih slučajeva do kojih u algoritmu može da dođe. Svaki slučaj se posebno razmatra. Opisuje se kako se slučaj obrađuje ili, ako se smatra da se neki slučaj nikada ne može pojaviti, onda se obrazlaže zašto je to tako. Na primer, ako se u nekom algoritmu javlja formula sa deljenjem dve promenljive, u dokumentaciji bi trebalo navesti u kom slučaju imenilac može da ima vrednost 0, i kako se u kôdu reaguje na njega.

Spoljašnja dokumentacija komponente treba da odslikava tok podataka na nivou komponente, što se obično predstavlja odgovarajućim dijagramima.

6.3 Kvalitet programiranja

Sve do nedavno se smatralo da će na osnovu dobrog dizajna svaki programer napisati dobar kôd. Međutim, Whittaker i Atkin (2002.g.) ističu da kvalitet programskog kôda u mnogome zavisi i od znanja, veštine, maštovitosti i iskustva programera u rešavanju problema.

Smatra se da pronalaženje dobrog rešenja zahteva prolazak kroz sledeće faze:

- razumevanje problema
- osmišljavanje plana rešenja
- izvršavanje plana i provera rešenja

Razumevanje problema podrazumeva njegovu detaljnu analizu, tj. precizno utvrđivanje uslova u kojima problem mora da se rešava. Pre svega, treba ustanoviti gde je granica sistema u odnosu na okruženje, tj. šta su ulazni podaci (u kom su obliku i kako se do njih može doći), a šta izlazni (u kom obliku i kome se oni isporučuju). Osim toga, moraju se uočiti i sva ograničenja i uslovi koje sistem treba da ispuni. Programeri često koriste grafički prikaz kako bi bolje razumeli problem. Crtaju dijagrame tokova koji im pomažu u prepoznavanju različitih uslova koji

moraju da budu ispunjeni. Ovi dijagrami mogu da ukažu i na mogućnost dekomponovanja problema na jednostavnije podprobleme koji se lakše rešavaju.

Sledeći korak u pronalaženju rešenja je *osmišljavanje plana*. Plan se može relativno lako napraviti ako je problem dobro shvaćen i ako je očigledno kako se postavljeni uslovi mogu ispuniti. Međutim, često nisu sve veze u sistemu odmah vidljive, pa se javljaju mnoge nepoznanice. U tom slučaju, da bi se pronašao pravi plan, preporučuje se isprobavanje sledećih tehnika:

- prepoznavanje sličnih problema (ispitivanje da li se već postojeći algoritmi, bibliotečke funkcije, podaci i sl. mogu upotrebiti za rešavanje razmatranog problema)
- preformulisanje problema (ispitivanje da li se mogu uvesti neke pretpostavke koje bi pojednostavile rešenje, ili da li se problem može konkretizovati, ili čak uopštiti u istom cilju)
- razlaganje problema (ispitivanje da li se mogu izdvojiti delovi koji bi se obrađivali zasebno jedan od drugog)

Prilikom pravljenja plana, dobro je da se omogući grupna diskusija zainteresovanih učesnika u projektu, čiji bi cilj bio analiziranje mogućih opcija i prepoznavanje najboljeg rešenja.

Nakon utvrđivanja plana, treba ga sprovesti u delo. To podrazumeva *proveru ispravnosti rešenja*, korak po korak. Za svaki korak se procenjuje da li se on može sprovesti na osnovu prethodnog koraka i da li omogućava postizanje uslova za izvršenje narednog koraka. Na primer, ako se u koracima pojavljuju neki logički iskazi, treba proveriti da li su oni dobro povezani, tj. da li se mogu izvršavati jedan posle drugog. Kada se dođe do rešenja, ponovo se pregleda svaki deo plana i analizira da li je rešenje višekratno, tj. da li se može primeniti više puta, da li je korisno i u drugim situacijama (i kojim), da li može da postane osnova za neki šablon, itd.

7 Testiranje softvera

U procesu razvoja softvera, nakon faze implementacije, sledi faza testiranja napisanih programa. Testiranje ima veliki značaj, jer se nakon ove faze softver isporučuje naručiocima. Obaveza svakog proizvođača softvera je da isporuči kvalitetan softver. To je u interesu ne samo krajnjeg korisnika, već i samog proizvođača, jer kvalitet isporučenog softvera direktno utiče na ugled proizvođača na tržištu softvera i mogućnosti dobijanja novih poslova u budućnosti.

Postoje različite tehnike testiranja softvera koje obezbeđuju da se kupcima isporuči kvalitetan sistem koji zadovoljava njihove zahteve. Cilj testiranja je otkrivanje grešaka u softveru. Međutim, ova faza nije jedina u procesu razvoja koja se bavi identifikovanjem i ispravljanjem grešaka. Skoro u svim fazama koje prethode testiranju ulažu se naponi da se smanji broj nedostataka u sistemu. Tako, potencijalni skup grešaka u softveru uključuje greške u projektnim zahtevima, u dizajnu, implementaciji, greške u dokumentaciji, kao i loše ispravke detektovanih problema u softveru. U svim fazama se teži da se potencijalni problemi uoče kako bi se što pre prevazišli. Dokazano je da otklanjanje nekog nedostatka više košta što se on kasnije otkrije.

Iako svaki proizvođač teži da napravi softver bez grešaka, to je praktično nemoguće. Programeri mogu da budu izuzetno dobri (imaju znanje i veliko iskustvo), ali ne mogu da garantuju da će njihov program raditi ispravno svaki put kada se pokrene i u svim situacijama. Greške se javljaju iz sledećih razloga:

- mnogi softverski sistemi su složeni, tako da prolaze kroz veliki broj stanja, koriste zahtevne algoritme, složene formule i sl. i nije jednostavno obezbediti ispravan rad sistema u svim situacijama u kojima on može da se nađe

- pri implementaciji sistema često se koriste alati koji su na raspolaganju programerima, ili alati koje programeri dobro poznaju, a možda nisu najpogodniji za realizaciju te vrste sistema
- dešava se da ni kupcima, tj. naručiocima nije potpuno jasno šta im je potrebno, pa samim tim može da dođe do neusklađenosti funkcija pojedinih delova sistema
- u složenim projektima učestvuje veliki broj ljudi (projektnata, programera, onih koji testiraju programe,...), pa su veće mogućnosti pojave međusobnog nerazumevanja

Kao što se vidi, na većinu od navedenih razloga programer ne može da utiče, pa samim tim ne može ni da bude siguran da će njegov program raditi u svim mogućim uslovima. U svakom slučaju, programer se uvek trudi da dobro sagleda kontekst problema koji rešava i da što pre uoči što više nedostataka.

7.1 Greške i otkazi

Nedostaci koji se pojavljuju u sistemu manifestuju se kroz greške i otkaze. Terminološki posmatrano, ova dva pojma se međusobno razlikuju. Greška predstavlja uzrok zbog koga je došlo do pojave nekog neželjenog efekta, a otkaz je sâm taj neželjeni efekat. Neželjeni efekat je situacija kada softver ne radi ono što je predviđeno projektnim zahtevima. Može se reći da je otkaz događaj koji ukazuje na to da u softveru postoji greška, jer je on u stvari posledica, tj. manifestacija te greške. Na primer, ako je u projektnim zahtevima definisano da sistem treba da pruži neku informaciju samo za to ovlašćenom korisniku, a on je pruža i neovlašćenim korisnicima, onda je reč o otkazu. Ukoliko je do ovakvog ponašanja sistema došlo zato što neki parametar u softveru nije dobro postavljen, onda je to greška u programu. Prilikom testiranja, greške se moraju ispraviti, dok je otklanjanje otkaza poželjno, ali nije uvek u potpunosti izvodljivo. Treba napomenuti da nije uvek jednostavno otkriti grešku (ili greške) koja je prouzrokovala neki otkaz.

Mnogi otkazi se detektuju tek nakon isporuke sistema. Uzroci otkazivanja mogu da budu različiti:

- nepotpuna ili pogrešna specifikacija zahteva (na primer, kupac nije eksplicitno izneo svoj zahtev da u sistemu treba da postoji više nivoa ovlašćenja)

- nemogućnost implementacije nekog zahteva na postojećoj hardverskoj ili softverskoj platformi
- greška pri projektovanju sistema (projekat ne odgovara u potpunosti postavljenim zahtevima)
- greška u programskom kôdu, tj. neodgovarajuća implementacija zahteva

Pošto razlozi otkaza mogu da budu vrlo raznovrsni i da potiču iz različitih faza razvoja, da bi se utvrdilo šta je pravi izvor otkaza, najjednostavnije je poći od provere ispravnosti napisanog kôda. To bi najverovatnije zahtevalo najmanje izmena u postojećem sistemu.

Mnogi programeri testiranje programa shvataju kao dokazivanje da ti programi ispravno rade. Međutim, svrha testiranja je upravo obrnuta. Cilj testiranja je nalaženje grešaka u programima, pa se uspešnim smatra onaj test tokom koga je identifikovana neka greška, ili je došlo do nekog otkaza. Ispravno shvatanje uloge testiranja doprinosi poboljšanju kvaliteta softvera, jer u potpunosti angažuje programera u njegovim naporima da pronađe grešku u softveru.

U slučaju pojave otkaza, problem se prevazilazi tako što se najpre prepozna greška koja je dovela do otkaza, a zatim se u sistem unesu potrebne izmene koje otklanjaju tu grešku. Pri detekciji grešaka mora se voditi računa o tome da su one mogle da nastanu kako u softveru, tako i u hardveru. Softverske greške su uvek u programskom kôdu i one su postojane. To znači da one postoje od trenutka kada je taj deo kôda napisan, a postojaće i dalje sve dok se taj deo kôda ne modifikuje. Greške u hardveru su drugačije prirode. Naime, hardverske komponente su podložne habanju, tako da vremenom mogu da promene svoju funkcionalnost i dovedu do neke vrste otkaza.

7.1.1 Klasifikacija grešaka

Nakon implementacije neke programske komponente, programer najpre pregleda napisani kôd kako bi video da li u njemu postoji neka očigledna greška. Zatim pristupa testiranju komponente, pri čemu pokušava da pronađe grešku tako što osmišljava, najpre jednostavne, a onda sve složenije uslove u kojima očekuje da program ne radi na odgovarajući način. Pri osmišljavanju uslova, važno je da programer zna koju vrstu grešaka traži.

S obzirom da je broj mogućih grešaka veoma veliki, pokazalo se korisnim da se one klasifikuju. Tako se mogu izdvojiti:

- *sintaksne greške*. Ove greške nastaju zbog pogrešne upotrebe iskaza programskog jezika koji je korišćen. Na primer, to može da bude izostanak zarez na kraju naredbe, pogrešan naziv kontrole toka, pokušaj korišćenja nepostojećeg tipa podataka, nepoštovanje propisane strukture petlje, itd. Mnoge od ovih grešaka prevodioci otkrivaju u toku kompajliranja programa i o njima obaveštavaju programera, navodeći vrstu greške i njenu tačnu poziciju u datoteci. Međutim, može se desiti da je sintaksna greška takva da samo menja funkcionalnost. Takva greška ne može biti otkrivena prevođenjem, a može da dovede do težih posledica (Mayers je 1976.god. istakao da je prvi let SAD na Veneru doživeo neuspeh zato što je u jednoj petlji napisanoj u programskom jeziku Fortran nedostajao zarez).
- *greške u postupku obrade*. Ove greške nastaju kada način obrade ulaznih podataka nije ispravno implementiran. Uobičajene greške ovog tipa su: prevremeno ili prekasno grananje u programu, ispitivanje pogrešnih uslova, neodgovarajuća inicijalizacija promenljivih, pogrešno zadati parametri petlje, izostanak razmatranja nekih slučajeva (na primer, deljenje nulom), poređenja promenljivih različitih tipova i sl. Ove greške se mogu otkloniti detaljnim čitanjem programa, ili testiranjem programa za ulazne podatke izabrane tako da su za njih poznati izlazni podaci koje program treba da generiše. U poslednjem slučaju, programer može da otkrije grešku praćenjem izvršavanja programa i poređenjem dobijenih međurezultata sa očekivanim.
- *greške u preciznosti*. Ukoliko se u programu koriste formule koje ne mogu da izračunaju rezultate sa potrebnom preciznošću, to može da dovede do pogrešnog rada programa. Takođe, do neočekivanih rezultata može da dođe i ako je formula dobra, ali su podaci nad kojima ona operiše neodgovarajućeg tipa, pa dolazi do odsecanja i smanjenja preciznosti izračunatih vrednosti. Ove greške se prevazilaze pažljivom proverom podataka i analizom načina primene formule.
- *greške zbog prekoračenja*. U mnogim sistemima postoje razna ograničenja koja se moraju ispoštovati. Na primer, može biti ograničen broj korisnika u sistemu, intenzitet njihove međusobne komunikacije, propusnost kanala za prenos podataka, itd. U fazi projektovanja, na osnovu ovih ograničenja, potrebno je podesiti ponašanje sistema za maksimalno opterećenje. To znači da postavljena ograničenja treba preneti na komponente dizajna. Na primer, treba odrediti dimenzije nizova, tabela (ukoliko se koriste), veličine bafera, dužine redova čekanja i sl. Ukoliko prilikom izvršavanja programa dođe do prepunjavanja struktura podataka predviđenih za opsluživanje ograničenja (na primer, prepuni se bafer), onda dolazi do grešaka zbog prekoračenja.

- *greške zbog performansi.* Ove greške nastaju kada sistem ne postiže performanse koje su predviđene projektnim zahtevima. Na primer, sistem može da radi neprihvatljivo sporo, posebno u uslovima koji se približavaju postavljenim granicama po pitanju opterećenja. Da bi se ove greške izbegle, programi moraju da budu istestirani u ekstremnim uslovima. Performanse se svakako moraju proveriti za maksimalno opterećenje, a poželjno je utvrditi (ako konfiguracija sistema to dozvoljava) i šta bi se desilo ako se sistem optereti i preko maksimuma. Na primer, ako je u zahtevima specificirano da sistem treba da opslužuje istovremeno 24 korisnika, sistem se mora testirati u uslovima kada je svih 24 korisnika aktivno. Dobro bi bilo i da se proverí šta bi se desilo kada bi broj korisnika bio i veći, jer bi to bila korisna informacija za eventualno buduće proširenje sistema.
- *greške u dokumentaciji.* Programski kôd se implementira na osnovu odgovarajuće dokumentacije. Međutim, u nekim situacijama može da dođe do njihove međusobne neusklađenosti (na primer, ako neka funkcija nije dobro implementirana). Prilikom traženja grešaka, programeri često analiziraju dokumentaciju kako bi napravili potrebne izmene u kôdu. To može da dovede do propagacije postojećih i pojave novih grešaka. Ove greške se mogu prevazići samo ponovnom proverom i usklađivanjem kôda i programske dokumentacije.
- *greške u vremenskoj koordinaciji.* U sistemima koji rade u realnom vremenu veoma je važna vremenska koordinacija procesa koji se izvršavaju. U zavisnosti od prirode samih procesa, neki od njih se mogu izvršavati istovremeno, dok se drugi moraju izvršavati sukcesivno u tačno zadatom redosledu. U svakom slučaju, deo programskog kôda mora da bude posvećen usklađivanju redosleda izvršavanja procesa. Ukoliko ovaj deo kôda nije dobro implementiran, pri izvršavanju programa se javljaju greške, koje se obično teško identifikuju i ispravljaju. Problemi u detekciji ovih grešaka nastaju iz dva razloga. Prvo, nije lako predvideti sva moguća stanja u kojima sistem može da se nađe, a drugo, zbog velikog broja parametara koji utiču na redosled procesa ponekad nije moguće ponoviti iste uslove u kojima se greška pojavila, pa se neka detaljnija analiza i ne može sprovesti.
- *greške zbog nepoštovanja standarda.* U mnogim kompanijama propisane su standardne procedure koje se moraju poštovati prilikom izrade softvera. Ukoliko se programer na pridržava ovih procedura, mogu se pojaviti greške. Ove greške mogu da utiču na rad programa, ali i ne moraju. Čak i u slučaju da ne remete rad programa, ove greške mogu da stvore probleme

onima koji testiraju ili održavaju program ako ne razumeju logiku rada ili u kôdu ne mogu da pronađu podatke ili funkcije koje su im potrebne.

Sve navedene vrste grešaka direktno su povezane sa programskim kôdom i u nadležnosti su programera. Međutim, postoje i greške koje nisu posledica loše napisanog kôda, već drugih činioca u sistemu. To su, na primer, greške u hardveru ili greške u sistemskom softveru. U mnogim sistemima, u okviru specifikacije zahteva navode se hardverski i softverski resursi koji će biti korišćeni. Sve komponente u sistemu projektuju se u skladu sa raspoloživim resursima. Greške mogu da nastanu ako isporučeni hardver ili softver ne funkcionišu kako se to od njih očekuje. U tom slučaju, treba proveriti u pratećoj dokumentaciji da li dati hardver ili softver uopšte mogu da rade u potrebnim operativnim uslovima (ako ne mogu, onda je greška napravljena prilikom izbora hardvera/softvera), pa ukoliko mogu, onda se treba obratiti proizvođaču kako bi on pomogao u rešavanju problema.

Pri izradi softvera, posebna pažnja se mora posvetiti oporavku sistema nakon različitih vrsta otkaza. Na primer, potrebno je utvrditi šta će se desiti sa sistemom ako tokom neke obrade nestane napajanje. Različiti sistemi mogu da se oporave na različite načine: neki će nastaviti sa radom uz uključivanje sopstvenih strujnih agregata, drugi mogu da zapamte datoteke pre otkaza, treći za sačuvaju skup poslednjih transakcija koje će biti ponovljene nakon dolaska napajanja, i sl. U svakom slučaju, nakon otkaza, sistem bi trebalo da se oporavi na prihvatljiv način. Ukoliko se to ne obezbedi, nastaju nekontrolisane greške usled otkaza čije posledice ne mogu uvek da se sagledaju.

7.2 Vrste testiranja

Testiranje sistema je formalni proces koji izvodi tim za testiranje sa ciljem da utvrdi logičku ispravnost i svrsishodnost testiranog programa. Testiranje se sprovodi na računaru uz poštovanje određenih test procedura koje se primenjuju na određene test slučajeve. Značaj testiranja je veliki zato što se na taj način značajno smanjuju gubici koje proizvođači softvera imaju usled grešaka i otkaza softvera nastalih nakon njegove isporuke kupcu. Glavni zadatak svakog člana tima za testiranje je da otkrije što je moguće više nedostataka, posebno ozbiljnijih koji mogu da imaju katastrofalne materijalne, informacione, bezbednosne i druge posledice. Proces testiranja je skup u dugotajan i zato svi proizvođači softvera ulažu velike napore da ga učine što efikasnijim.

Mnogi programeri programiranje shvataju vrlo lično, kao odraz njihovog znanja, inteligencije i mogućnosti. Stoga svaku kritiku na račun programa koje su

napisali doživljavaju kao kritiku ličnih sposobnosti, pa im je teško da potisnu lična osećanja i sujetu. Ovakvi stavovi mogu da ugroze projekat. Da bi se to izbeglo, mnogi proizvođači softvera (koji imaju mogućnosti za to) formiraju specijalne timove za testiranje. To znači da jedan čovek piše programski kôd, a drugi ga testira. Na ovaj način se izbegava konflikt između vlastitog osećanja odgovornosti za greške i potrebe da se pronađe što više grešaka u kôdu.

Postojanje timova za testiranje pruža još neke prednosti. Prvo, može se desiti da programer nesvesno pogrešno protumači dizajn, tj. napravi grešku u tumačenju rada komponente ili u postupku implementacije nekog algoritma. Naravno, on ne bi dao svoj program na testiranje ako bi mislio da on nije u skladu sa datom specifikacijom. Međutim, pošto je previše blizak svom kôdu, programer nije u stanju da bude objektivan i sagleda napravljenu grešku. Veća je verovatnoća da će to učiniti onaj ko testira dati program. Drugo, postojanje timova za testiranje uvodi paralelizam u radu na projektu, čime se mogu postići značajne vremenske uštede. Naime, kada programer završi implementaciju jedne komponente, on je daje na testiranje, i nastavlja sa radom na sledećoj komponenti. Tako se proces programiranja i proces testiranja uvek obavljaju paralelno, čime se ne samo štedi vreme, već se i ranije uočavaju greške.

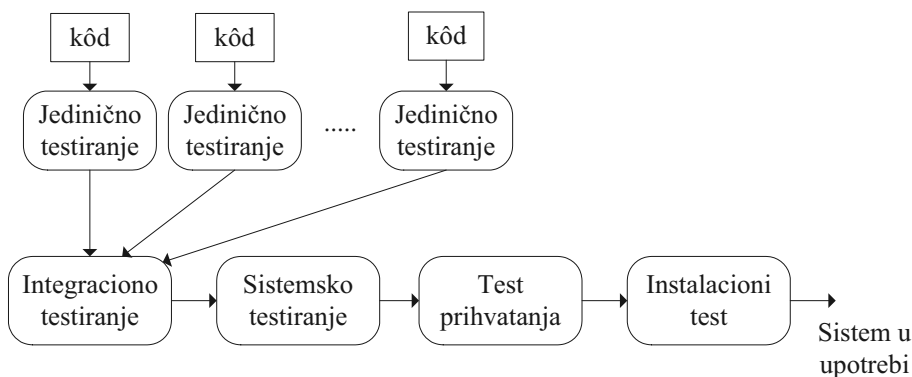
Testiranje se sprovodi imajući u vidu dva cilja: verifikaciju i validaciju sistema. Verifikacija sistema treba da pruži odgovor na pitanje da li se sistem razvija (gradi) na pravi način, dok validacija treba da odgovori na pitanje da li je sistem koji se razvija zaista ono što je korisniku potrebno. Da bi se dobili odgovori na ova pitanja, potrebno je proći kroz sledeće faze u procesu testiranja:

- *jedinično testiranje (testiranje pojedinačnih modula)*. U ovoj fazi se svaka programska komponenta testira nezavisno od ostatka sistema. Testiranje se svodi na proveru funkcionalnosti komponente za unapred definisan skup ulaznih podataka koji reflektuje sve moguće situacije u kojima komponenta može da se nađe. Testiranjem se proverava da li se za dati skup ulaznih podataka dobijaju očekivani rezultati na izlazu ili se izvršavaju očekivane akcije. Ova vrsta testiranja obuhvata i proveru korišćenih struktura podataka, logike primenjenih procedura, kao i opsega ulaznih i izlaznih podataka.
- *integraciono testiranje*. Pošto su pojedinačne komponente u sistemu dobro implementirane, u ovoj fazi se testira saradnja između ovih komponentata. Proverava se da li su veze između komponentata dobro definisane i realizovane, tj. da li komponente komuniciraju na način opisan u projektu sistema i programa. Rezultat ove faze je kompletan (integrisani) sistem koji radi.

- *sistemska testiranje (završno testiranje)*. Nakon dobijanja sistema koji radi, u ovoj fazi se testira da li sistem odgovara zahtevima korisnika. Proverava se da li sistem izvršava sve funkcije opisane u projektnim zahtevima specificiranim u saradnji sa kupcem. Takođe se vrši i testiranje performansi sistema da bi se videlo da li su one u skladu sa postavljenim zahtevima po pitanju hardvera i softvera. Rezultat ove faze je kompletan sistem koji se može isporučiti korisniku.

Jedinično testiranje i integraciono testiranje omogućavaju verifikaciju sistema, dok završno, sistemsko testiranje proverava validnost sistema.

Pre nego što se sistem isporuči, proizvođač softvera treba da obavi i *završni test prihvatanja* sistema, u kojem, sada zajedno sa kupcem, proverava usklađenost sistema sa postavljenim zahtevima. Nakon ovog testa, prihvaćeni sistem se instalira u realnom radnom okruženju i konačno, *instalacionim testom* se proverava da li sistem i dalje ispravno funkcioniše. Na slici 7.1 prikazane su sve faze u procesu testiranja kroz koje treba proći da bi sistem bio u upotrebi.



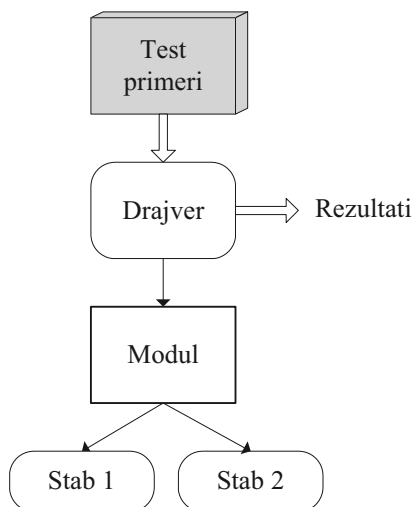
Slika 7.1 Proces testiranja

7.2.1 Jedinično testiranje

Modul predstavlja najmanju programsku celinu koja se testira izolovano od ostatka sistema. Jedinično testiranje je metod u kome se individualni moduli programskog kôda testiraju kako bi se ustanovilo da li odgovaraju svojoj nameni.

Iako se testiraju izolovano, pojedinačni moduli nisu nezavisni od sistema. Njih pozivaju neki drugi moduli, a i oni sami pozivaju neke module. Prema tome, da bi

se obavilo samostalno testiranje jednog modula, neophodno je razviti odgovarajuće programsko okruženje, kao što je prikazano na slici 7.2.



Slika 7.2 Okruženje za testiranje modula

Drajver (driver) predstavlja glavni program koji prihvata ulazne podatke iz test primera, prosleđuje ih modulu koji se testira i generiše rezultate testa. Dakle, on simulira komponentu koja poziva modul koji se testira. *Stab (stub)* je program koji simulira module koje poziva modul koji se testira. On koristi interfejse pozivajućih modula, izvršava minimalnu obradu podataka, daje rezultat i vraća kontrolu modulu koji se testira.

Da bi se pronašle greške u nekom programskom modulu, treba uraditi sledeće:

- pročitati kôd sa ciljem pronalaženja očiglednih grešaka u sintaksi, podacima, algoritmima, itd.; pri ovome se može pogledati i specifikacija dizajna kako bi se proverilo da li kôd uzima u obzir sve relevantne slučajeve
- prevesti kôd pomoću prevodioca (kompajlera), čime se uklanjaju preostale sintaksne greške
- generisati test procedure za test slučajeve kojima se može proveriti da li se ulazni podaci pravilno konvertuju u željene izlazne podatke

Programer piše programski kôd na osnovu dizajna, tj. dokumentacije u kojoj je rečima i slikom opisano šta konkretna komponenta treba da radi. To znači da

programski kôd predstavlja programerovo tumačenje dizajna. Pošto ovo tumačenje može da bude i pogrešno, dobro je da se formira objektivna grupa koja bi još jednom pregledala kôd. U njoj bi se, osim autora kôda, nalazilo i nekoliko nezavisnih stručnjaka. Grupa bi trebalo da utvrdi da li je kôd napisan u skladu sa dokumentacijom, ili je došlo do nekih nesporazuma, pogrešnih tumačenja, nekonzistentnosti ili drugih propusta.

Pregled kôda može da se obavi na dva načina: letimičnim pregledom ili inspekcijom kôda.

Letimični pregled se odvija u nezvaničnoj atmosferi. Programer prezentira kôd koji je napisao ostatku grupe i onda dolazi do diskusije. Članovi grupe postavljaju pitanja, programer na njih odgovara, zajednički analiziraju pojedine aspekte realizacije koje smatraju bitnim i pokušavaju da pronađu greške. Tokom ovog pregleda, rasprava je usmerena isključivo na kôd, a ne na ocenjivanje sposobnosti njegovog autora.

Inspekcija kôda je metod koji je prvi uveo Fagan 1976.godine u IBM-u. Ova vrsta pregleda odvija se u zvaničnoj atmosferi. Grupa za pregled proverava programski kôd i dokumentaciju prema prethodno pripremljenoj listi pitanja. Pitanja se odnose na različite aspekte realizacije. Na primer, ona mogu da budu u vezi sa proverom:

- definicija i načina korišćenja struktura i tipova podataka u komponenti
- ispravnosti i efikasnosti primenjenih algoritama i drugih izračunavanja
- ispravnosti interfejsa komponente
- korektnosti napisanih komentara, tj. njihove usklađenosti sa kôdom
- performansi (brzina obrade, efikasnost korišćenja memorije i dr.)

Da bi se obavila inspekcija kôda, grupa se najpre sastaje da utvrdi ciljeve inspekcije i grubo pregleda kôd. Nakon toga, svaki član grupe pojedinačno proučava kôd i prateću dokumentaciju i pokušava da pronađe greške. Na kraju, svi članovi grupe se ponovo sastaju i podnose svoje izveštaje o uočenim nedostacima. Zatim cela grupa analizira pronađene greške i donosi odluku o tome koji nedostaci predstavljaju greške koje treba ispraviti, a koji ne predstavljaju problem. Svim sastancima rukovodi vođa grupe koji vodi računa i o tome da se (kao i kod letimičnog pregleda) diskutuje isključivo o kôdu, a ne o autoru kôda.

Istraživanja pokazuju da je inspekcija kôda efikasnija od letimičnog pregleda (jer je i detaljnija). Fagan je radio studiju u kojoj su dve grupe pregledale kôd korišćenjem navedenih metoda. Dobijeno je da je u toku prvih sedam meseci rada aplikacije 38% manje grešaka uočeno kod grupe koja je radila inspekciju kôda

nego kod grupe koja je radila letimičan pregled. U drugom Faganovom eksperimentu, od ukupnog broja grešaka otkrivenih tokom razvoja sistema, 82% je pronađeno tokom inspekcije.

Iako autorima kôda svakako nije prijatno da neko drugi pregleda njihov kôd, pregledanje kôda se pokazalo vrlo korisnim, tako da su ga mnoge kompanije uvrstile u svoje standardne procedure. Pregled na nivou komponenata olakšava nalaženje grešaka, jer su uzroci problema jasniji u ovoj, nego u kasnijim fazama razvoja. Ispravljanje grešaka u ranim fazama razvoja je, takođe, mnogo jeftinije nego u kasnijim.

Nakon što je programer napisao kôd koji realizuje neku komponentu, i nakon što je taj kôd pregledala odgovarajuća grupa stručnjaka i utvrdila da on odgovara dizajnu, prelazi se na sledeći korak u procesu testiranja. To je analiza kôda na sistematičan način, pomoću test procedura, kako bi se utvrdila njegova ispravnost. Kôd je ispravan ako implementira funkcionalnost i veze prema drugim komponentama zahtevane u dizajnu.

U opštem slučaju, procesu testiranja modula može se pristupiti na dva načina: metodom „crne kutije“ ili metodom „bele kutije“.

Metodi „crne kutije“ i „bele kutije“ predstavljaju dve krajnosti, jer se u prvom slučaju ništa ne zna o procesu obrade podataka, dok se u drugom slučaju zna sve. Pri donošenju odluke o načinu testiranja, ne mora se izabrati samo jedan metod. Metodi se mogu kombinovati u zavisnosti od mogućnosti koje postoje u konkretnom slučaju. Na izbor načina testiranja utiče više faktora:

- priroda ulaznih podataka
- broj mogućih logičkih putanja koje treba proveriti
- količina izračunavanja
- složenost primenjenih algoritama

7.2.1.1 Metod „crne kutije“

Po metodu „crne kutije“, program se shvata kao zatvorena (crna) kutija nepoznatog sadržaja kod koje su vidljivi samo ulazi i izlazi. Funkcionalnost programa se određuje samo posmatranjem dobijenih izlaznih podataka na osnovu odgovarajućih, poznatih ulaznih podataka. Prilikom testiranja, na osnovu zadatih ulaznih podataka, dobijeni izlazni podaci se upoređuju sa unapred očekivanim i na taj način se proverava ispravnost programa. Kao što se vidi, u ovom pristupu svi

testovi potiču iz specifikacije programa i ne vrši se nikakvo razmatranje programskog kôda.

Prednost metoda „crne kutije“ je u tome što je testiranje oslobođeno brige o ograničenjima koja proističu iz unutrašnje strukture komponente i logike njenog rada. Glavni nedostatak ovog metoda je u tome što je detaljno testiranje svih kombinacija različitih ulaznih podataka za većinu programa praktično neizvodljivo. Ovo važi čak i ako se posmatra ispravnost samo osnovnih elemenata, kao što su ispravnost unetih vrednosti, vreme unosa, redosled unosa, itd.

Na primer, neka je napisan program koji ima tri ulazna parametra (a , b i c), a kao rezultat daje dva rešenja kvadratne jednačine

$$ax^2 + bx + c = 0 \quad \text{ili poruku} \quad \text{„Nema realnih rešenja.“}$$

Očigledno je da se ovaj program ne može testirati proverom svih mogućih vrednosti ulaznih parametara, jer ima beskonačno mnogo kombinacija. Zato onaj ko testira program mora da izabere reprezentativne kombinacije (a , b , c) tako da može da dokaže da će i ostale moguće kombinacije program pravilno obraditi. U posmatranom slučaju, kao reprezentativne kombinacije mogu se uzeti sve kombinacije pozitivnih, negativnih vrednosti i nule za ulazne parametre (ukupno 27 kombinacija). Ili, ako onaj ko testira program bolje poznaje problematiku problema (a uz to zna i koji problem implementira komponenta), može uzeti triplete (a , b , c) tako da je diskriminanta $b^2 - 4ac$ pozitivna, negativna ili jednaka nuli. Međutim, i u slučaju da uspešno prođu svi testovi nad reprezentativnim ulaznim podacima, nema garancije da komponenta zaista ne sadrži greške. Na primer, može se desiti da komponenta nekad da pogrešan rezultat zbog greške u zaokruživanju, ili zbog neusklađenosti tipova podataka i sl.

Moguće su i situacije kada onaj ko testira program nije u stanju da napravi skup reprezentativnih slučajeva na osnovu koga bi dokazao pravilno funkcionisanje komponente generalno. To se dešava kada je nedovoljno poznat način obrade koji se odvija u unutrašnjosti komponente. Na primer, ako komponenta računa porez na prihod, a ulazni podatak je vrednost bruto prihoda koji se oporezuje, teško je definisati reprezentativne vrednosti zato što nisu poznate poreske kategorije (one su ugrađene u komponentu), pa se ne zna šta treba da se očekuje kao rezultat. Ovaj problem se može prevazići metodom „bele kutije“.

Metod „crne kutije“ je bio osnova za formulisanje više tehnika funkcionalnog testiranja, od kojih će tri biti opisane u nastavku:

- podela na klase ekvivalencije
- analiza graničnih vrednosti
- uzročno-posledični grafovi

Podela na klase ekvivalencije

Tehnika podele na klase ekvivalencije polazi od ideje da se ulazni podaci mogu razvrstati u reprezentativne klase tako da se za sve pripadnike jedne klase program ponaša na sličan način. Te reprezentativne klase su nazvane klasama ekvivalencije. U idealnom slučaju, klase ekvivalencije su međusobno disjunktne i pokrivaju ceo prostor vrednosti ulaza.

Testiranje se obavlja samo za jednu reprezentativnu vrednost ulaza iz svake klase ekvivalencije, zato što se smatra da je to jednako delotvorno kao i testiranje bilo kojom drugom vrednošću iz iste klase. Naime, očekuje se da bi se u svim ovim slučajevima pronašla ista greška u programu.

Klase ekvivalencije se formiraju na osnovu svih uslova iz specifikacije koji se odnose na ulaze programa. Za svaki uslov se posmatraju dve grupe klasa prema zadovoljenosti tog uslova:

- legalne klase koje obuhvataju dozvoljene situacije
- nelegalne klase koje obuhvataju sve ostale situacije

Na primer, ako je u specifikaciji naznačeno da ulazni podatak p treba da bude broj između 0 i 100 (uključujući i njih), definišu se:

- jedna legalna klasa ekvivalencije ($1 \leq p \leq 100$) i
- dve nelegalne klase ekvivalencije ($p < 1$) i ($p > 100$).

Ako ulazni podatak p treba da ima fiksnu vrednost 15, definišu se:

- jedna legalna klasa ekvivalencije ($p = 15$) i
- dve nelegalne klase ekvivalencije ($p < 15$) i ($p > 15$).

Ako ulazni podatak p treba da uzme vrednost iz zadatog skupa $\{a, b, c\}$, pri čemu se za svaku vrednost iz skupa program različito ponaša, definišu se:

- tri legalne klase ekvivalencije ($p = a$), ($p = b$) i ($p = c$), i
- jedna nelegalna klasa ekvivalencije (van skupa), na primer ($p = d$).

Ukoliko se javi sumnja da se program ne ponaša isto za svaki element klase ekvivalencije, onda tu klasu treba podeliti na više manjih.

Nakon određivanja klasa ekvivalencije, formiraju se test primeri za:

- sve legalne klase ekvivalencije (cilj je da se jedan test primer primeni na što više legalnih klasa)
- sve nelegalne klase ekvivalencije (za svaku nelegalnu klasu mora se napisati poseban test primer, kako bi se izbeglo da jedan neregularan ulazni podatak maskira neki drugi, takođe neregularan, zbog provera unetih u kôd).

Pretpostavimo da razmatranom tehnikom treba testirati deo programa koji ispituje ispravnost unetog telefonskog broja. Neka je dozvoljeni format broja:

+|00 (nnn) nn – nnn – nnn

gde n predstavlja numerički simbol, a $|$ „ili“ operator. Grupa (nnn) označava državu i pripada skupu {381}, a grupa nn označava telefonskog operatera i pripada skupu {11,12,13}. Analizom svakog od navedenih uslova, mogu se dobiti legalne i nelegalne klase ekvivalencije (oznake klasa date su u zagradama) date u tabeli 7.1:

Uslov	Legalne klase	Nelegalne klase
Početni simbol	(1) +, 00	(8) sve ostalo
Oznaka države	(2) 381	(9) sve ostalo
Broj simbola u oznaci operatera	(3) 2	(10) > ili < od 2
Prvi simbol u oznaci operatera	(4) 1	(11) sve ostalo
Drugi simbol u oznaci operatera	(5) 1,2,3	(12) $\neq$ 1,2,3 ili nenum. simbol
Separatori između grupa simbola	(6) –	(13) bilo koji drugi simbol
Br. simb. u poslednje dve grupe	(7) 3	(14) $\neq$ 3

Tabela 7.1 Klase ekvivalencije

Na osnovu date tabele, mogu se napraviti sledeći test primeri:

+ (381) 11 – 284 – 667	zadovoljava sve legalne klase, (1) do (7)
– (381) 12 – 224 – 017	zadovoljava nelegalnu klasu (8)
00(368) 18 – 284 – 667	zadovoljava nelegalne klase (9) i (12)
+ (381) 111 – 284 – 667	zadovoljava nelegalnu klasu (10)
+ (381) 21 – 284 – 667	zadovoljava nelegalnu klasu (11)
+ (381) 11 / 284 / 667	zadovoljava nelegalnu klasu (13)
+ (381) 11 – 24 – 67	zadovoljava nelegalnu klasu (14)

Analiza graničnih vrednosti

U radu sa klasama ekvivalencije često se dešavaju greške zbog neadekvatnog definisanja njihovih granica. Na primer, programer može umesto oznake $\leq$ pogrešno da iskoristi oznaku $<$. Stoga je uvedena analiza graničnih vrednosti koja se primenjuje pri generisanju test primera u cilju potpunije podele opsega ulaznih podataka.

Analiza graničnih vrednosti se radi tako što se test primeri biraju na granicama različitih klasa ekvivalencije. Pri tome se vodi računa, ne samo o ulaznim uslovima, već i o izlaznim podacima iz programa.

Majersove preporuke za obavljanje ove analize su:

- ako ulazni uslov predstavlja opseg vrednosti, testove treba napisati za krajeve opsega i za vrednosti odmah do krajeva; na primer, za ulaz u opsegu -1.0 do 1.0, treba testirati vrednosti: -1.0, 1.0, -1.0001 i 1.0001
- ako ulazni uslov precizira broj mogućih vrednosti, testove treba napisati za minimalan i maksimalan broj vrednosti i za jedan ispod i jedan iznad broj; na primer, ako ulazna datoteka može da sadrži 1 do 255 zapisa, treba napisati testove za 1, 255, 0 i 256 zapisa
- prvu preporuku treba primeniti i na izlazne uslove
- drugu preporuku treba primeniti i na izlazne uslove; na primer, ako na stranu izveštaja staje 65 redova, treba napisati testove za 64, 65 i 66 redova
- ako je ulaz (ili izlaz) programa uređen skup (neka datoteka, lista, niz), treba obratiti pažnju na prvi i poslednji element u skup
- upotrebiti sopstvene mogućnosti za nalaženje drugih graničnih uslova koje bi trebalo testirati

Analiza graničnih vrednosti će biti ilustrovana na jednostavnom primeru pisanja formata datuma u godini koja nije prestupna: *dd.mm*, gde *dd* predstavlja dan, a *mm* mesec u godini.

Kao što je poznato, za mesece januar, mart, maj, jul, avgust, oktobar i decembar, *dd* pripada opsegu 1 do 31, za februar opsegu 1 do 28, a za ostale mesece opsegu 1 do 30. Slično ovome, *mm* pripada opsegu 1 do 12. Ovo su legalne klase u primeru i na osnovu njih mogu se napisati test primeri sa graničnim vrednostima za:

dan	01.04.	30.04.	23.09.
mesec	04.01.	04.12.	24.08.

Nelegalne klase predstavljaju vrednosti van navedenih opsega. Za njih se mogu napraviti sledeći test primeri sa graničnim vrednostima za:

dan	00.05.	32.05.
mesec	06.00.	06.13.

Uzročno-posledični grafovi

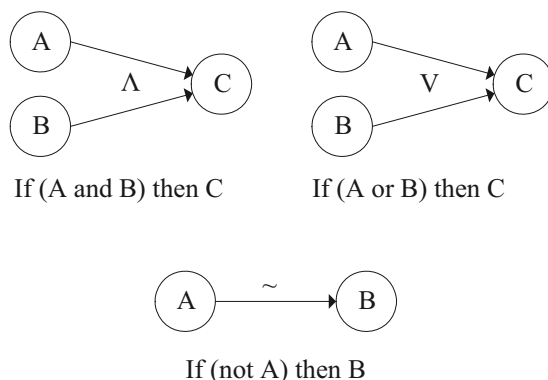
Testiranje softvera bi bilo znatno olakšano ako bi test primeri mogli da se automatski generišu na osnovu zahteva. Da bi se to postiglo, potrebno je analizirati zahteve i preformulisati ih u logičke relacije između ulaza i izlaza. Rezultat ovoga može se predstaviti uzročno-posledičnim grafom. Ova tehnika se ne može primeniti u sistemima sa vremenskim ograničenjima i sistemima sa konkurentnim procesima.

Uzročno-posledični graf se konstruiše na sledeći način:

- specifikacija se najpre podeli na radne delove (čime se ograničava veličina grafa)
- zatim se identifikuju uzroci (u vidu ulaznih uslova ili klasa ekvivalencije ulaznih uslova) i posledice (u vidu izlaznih podataka ili unutrašnjih promena stanja sistema)
- analiziranjem značenja specifikacije, generiše se uzročno-posledični graf

Graf se sastoji od čvorova koji predstavljaju uzroke, posledice i međučvorove. Na slici 7.3 dati su primeri elementarnih uzročno-posledičnih grafova.

Čvorovi uzroka se obično crtaju na levoj strani grafa, a čvorovi posledica na desnoj. Čvorovi su povezani granama u skladu sa relacijama koje postoje između uzroka i posledica. Zavisno od relacija, mogu se dodavati novi čvorovi, tj. međučvorovi. U granama grafa mogu se definisati zavisnosti tipa AND ($\wedge$), OR ($\vee$) i NOT ($\sim$).



Slika 7.3 Tipovi zavisnosti u uzročno-posledičnim grafovima

Da bi se generisao skup test primera, najpre se uzročno-posledični graf mora transformisati u tabelu odlučivanja. Ova tabela predstavlja dvodimenzionalnu strukturu koja mapira uzroke u posledice.

Tabela odlučivanja ima po jednu vrstu za svaki uzrok i posledicu. Po vrstama, podeljena je u dva dela. U prvom delu se navode uzroci (uslovi), a u drugom posledice (akcije). U poljima tabele mogu se pojaviti sledeće vrednosti: T (*true*), F (*false*) i * (označava da sadržaj polja nije bitan). Značenje T i F zavisi od dela tabele u kome se pojavljuju. U prvom delu tabele:

- T označava da uslov mora da bude ispunjen kako bi se postiglo pravilo
- F označava da uslov ne sme da bude ispunjen kako bi se postiglo pravilo

U drugom delu tabele, značenje T i F je sledeće:

- T označava da će akcija biti izvršena
- F označava da akcija neće biti izvršena

Kolone tabele odgovaraju pravilima na osnovu kojih se generišu test primeri. Broj kolona u tabeli zavisi od broja uzroka, a sadržaj kolona se dobija analizom čvorova koji predstavljaju posledice u grafu. Naime, u kolonama se navode sve kombinacije uzroka koje mogu da izazovu posmatranu posledicu.

Korišćenjem tabele odlučivanja, broj test primera se značajno smanjuje, što štedi vreme potrebno za testiranje komponente.

U nastavku je dat primer generisanja skupa testova primenom uzročno-posledičnog grafa. Neka je data funkcija koja obrađuje podizanje novca sa računa. Funkcija ima tri ulaza (količina novca koji se podiže, tip računa i trenutno stanje na

računu) i dva izlaza (novo stanje na računu i kôd akcije). Tip računa može biti p (poštanski) i c (klasični). Kôd akcije može da bude $D\&L$ (obrađi podizanje novca i pošalji pismo), D (obrađi podizanje novca), $S\&L$ (blokiraj račun i pošalji pismo) i L (pošalji pismo). Funkcija radi prema sledećoj specifikaciji:

Ukoliko ima dovoljno novčanih sredstava na računu ili bi novo stanje bilo u granicama dozvoljenog minusa, podizanje novca se obrađuje. Ukoliko bi podizanje novca dovelo do prekoračenja dozvoljenog minusa, podizanje novca nije moguće. U tom slučaju, ukoliko je u pitanju poštanski račun, vrši se njegovo privremeno blokiranje. Pismo se šalje za sve obavljene transakcije u slučaju poštanskog računa, kao i za klasični račun ukoliko na njemu nema dovoljno novčanih sredstava (tj. račun više nije u plusu).

Na osnovu specifikacije, mogu se identifikovati sledeći uzroci:

U1. Novo stanje je u plusu.

U2. Novo stanje je u dozvoljenom minusu.

U3. Račun je poštanski.

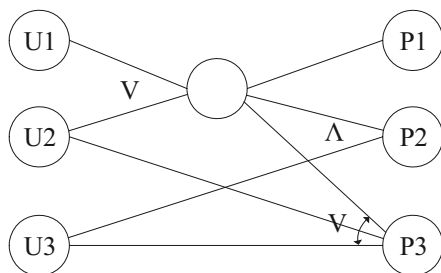
Uočene posledice su:

P1. Obrada podizanja novca.

P2. Privremeno blokiranje računa.

P3. Slanje pisma.

Veze između uzroka i posledica mogu se predstaviti uzročno-posledničnim grafom prikazanim na slici 7.4.



Slika 7.4 Uzročno-posledični graf za primer uzimanja novca sa računa

Da bi se generisao skup test primera, dati uzročno-posledični graf mora se transformisati u tabelu odlučivanja 7.2.

Čvorovi	1	2	3	4	5	6	7	8
U1. Novo stanje je u plusu.	F	F	F	F	T	T	T	T
U2. Novo stanje u dozvoljenom minusu.	F	F	T	T	F	F	T	T
U3. Račun je poštanski.	F	T	F	T	F	T	F	T
P1. Obrada podizanja novca.	F	F	T	T	T	T	*	*
P2. Privremeno blokiranje računa.	F	T	F	F	F	F	*	*
P3. Slanje pisma.	T	T	T	T	F	T	*	*

Tabela 7.2 Tabela odlučivanja za primer uzimanja novca sa računa

Kao što se vidi, tabela ima 8 (2^3) kolona za pravila zato što ima 3 uzroka. U prvom delu tabele navedene su sve moguće kombinacije za uzroke. Drugi deo tabele je popunjen na osnovu vrednosti uzroka i logike funkcije. Za pravila 7 i 8 akcije nisu definisane zato što te kombinacije uzroka nisu moguće (novo stanje ne može istovremeno da bude i u plusu i u dozvoljenom minusu).

Na osnovu pravila u tabeli odlučivanja, mogu se generisati test primeri dati u tabeli 7.3:

Test primer	Tip računa	Dozvoljeni minus (£)	Trenutno stanje (£)	Suma za podizanje (£)	Novo stanje (£)	Kôd akcije
T1	c	100	-70	50	-70	L
T2	p	1500	420	2000	420	S&L
T3	c	250	650	800	-150	D&L
T4	p	750	-500	200	-700	D&L
T5	c	1000	2100	1200	900	D
T6	p	500	250	150	100	D&L

Tabela 7.3 Tabela odlučivanja za primer uzimanja novca sa računa

Ovi test primeri u potpunosti pokrivaju prostor uzroka-posledica. Pošto su pravila 7 i 8 nemoguća, za njih nisu definisani test primeri.

7.2.1.2 Metod „bele kutije“

Metod „bele kutije“ testira i analizira izvorni programski kôd i zahteva dobro poznavanje programiranja, korišćenog programskog jezika, kao i dizajna konkretne programske komponente. U ovom metodu, program se shvata kao otvorena (bela) kutija čija je unutrašnjost poznata. Plan testiranja se formira na osnovu strukture programa. Tako se mogu generisati testovi koji izvršavaju sve naredbe u programu, pozivaju sve funkcije i prolaze kroz sve tokove kontrole unutar komponente. Na ovaj način se može proveriti skoro celokupan kôd. Specifičnim testovima može se

proveriti postojanje beskonačnih petlji ili izvršavanje kôda koji u regularnim uslovima ne bi trebalo nikada da se izvrši.

Glavni nedostatak ovog metoda je u tome što i ova vrsta pristupa može da bude praktično neizvodljiva. Na primer, ako komponenta sadrži mnogo grananja ili petlji, broj putanja koje treba proveriti može da bude ogroman (reda milijardi). Čak i u slučaju jednostavnijih logičkih struktura, teško je detaljno testirati komponentu sa velikim brojem iteracija ili rekurzija. U ovim slučajevima, treba usvojiti neku strategiju testiranja u kojoj bi se petlje ili rekurzije izvršavale manji broj puta nad reprezentativnim, pažljivo odabranim vrednostima. Strategija može da se zasniva na podacima, funkcijama, kontrolnim strukturama ili nekim drugim kriterijumima.

Slično metodu „crne kutije“, i metod „bele kutije“ je poslužio kao osnova za razvoj više tehnika strukturnog testiranja. To su:

- pokrivanje iskaza
- pokrivanje odluka
- pokrivanje uslova

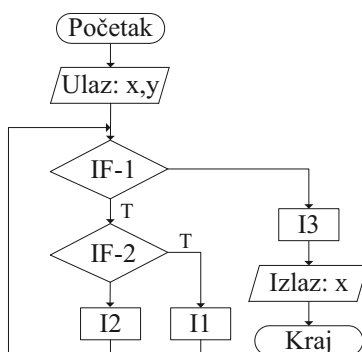
Pokrivanje iskaza

Pri testiranju programa, test primeri se prave tako da se svaki iskaz u programu bar jednom izvrši, jer se drugačije ne može znati da li u njemu postoji greška. Međutim, ako se iskaz ispravno izvrši za jednu ulaznu vrednost, to ne znači da će se dobro izvršiti i za neku drugu ulaznu vrednost. Ovo je glavni nedostatak razmatrane tehnike.

Pretpostavimo da treba proveriti ispravnost programa koji realizuje Euklidov algoritam za nalaženje najvećeg zajedničkog delioca brojeva x i y :

```
int euklid(int x, int y) {
    while (x != y) {           // uslov IF-1
        if (x > y) then        // uslov IF-2
            x = x - y;         // iskaz I1
        else
            y = y - x;         // iskaz I2
    }
    return x; }               // iskaz I3
```

Pre generisanja test primera, dati algoritam je pogodno predstaviti pomoću dijagrama toka, kao na slici 7.5.



Slika 7.5 Dijagram toka za primer Euklidovog algoritma

Na osnovu dijagrama toka, mogu se napisati test primeri koji pokrivaju sve iskaze u programu. Na primer, prvi test (T1) pokriva slučaj kada uslov IF-1 nije ispunjen, pa se izvršava izkaz I3. Na sličan način su formirani i ostali testovi koji su navedeni u tabeli 7.4:

Test primer	x	y	Pokriveni iskazi
T1	3	3	IF-1, I3
T2	4	3	IF-1, IF-2, I1
T3	3	4	IF-1, IF-2, I2

Tabela 7.4 Test primeri za pokrivanje iskaza

Pokrivanje odluka

Tehnika pokrivanja odluka podrazumeva projektovanje test primera tako da se svaka od različitih grana uslovnog iskaza izvrši bar jednom. Ovo je „jači“ metod od pokrivanja iskaza zato što pokrivanje odluka garantuje pokrivanje iskaza.

Odlučivanje se vrši na osnovu iskaza:

if A then B else C

Ako je uslov A ispunjen, pokrivaju se svi iskazi B koji se nalaze u odgovarajućoj grani, dok se posebno mora napisati test kada uslov A nije ispunjen, kako bi se pokrili i iskazi C.

Program koji realizuje Euklidov algoritam može se testirati i ovom metodom. Najpre se generiše dijagram toka (kao ranije), a zatim se pišu test primeri (isti su kao i kod tehnike pokrivanja iskaza).

Pokrivanje uslova

U tehnici pokrivanja uslova, test primeri se pišu tako da svaka elementarna komponenta nekog složenog uslova uzima vrednost iz dozvoljenog i nedozvoljenog skupa vrednosti. Elementarni uslovi se posmatraju potpuno nezavisno jedan od drugog. Ovaj metod ne garantuje pokrivanje svih odluka, pa samim tim ni pokrivanje svih iskaza u programu.

Na primer, neka je dat logički izraz:

((c1 and c2) or c3)

Test primeri se formiraju tako da u njima c1, c2 i c3 imaju obe logičke vrednosti: tačno (T) i netačno (F):

T1: c1 = T c2 = T c3 = F

T2: c1 = F c2 = F c3 = T

Postoji i tehnika pokrivanja višestrukih uslova koja podrazumeva pokrivanje svih mogućih kombinacija elementarnih uslova u složenim izrazima. Na primer, neka je uslov:

if (c == 'A' || c == 'B') c = 'X'

Elementarni uslovi su: c == 'A' i c == 'B'. Test primeri sa svim mogućim kombinacijama na ulazu (c) dati su u tabeli 7.5:

Elementarna komponenta	Test primeri			
	c = 'A'	c = 'B'	c = 'X'	c = ?
c == 'A'	T	F	F	T
c == 'B'	F	T	F	T

Tabela 7.5 Test primeri za pokrivanje uslova

Kao što se vidi, ne postoji ulaz za koji bi oba elementarna uslova bila tačna.

Ako se složeni uslov sastoji od n elementarnih komponenta, potrebno je napisati 2^n test primera. Ova tehnika je praktično izvodljiva samo za male vrednosti n .

7.2.2 Integraciono testiranje

Po završetku testiranja pojedinačnih modula, pristupa se njihovom povezivanju u jednu celinu koja predstavlja sistem namenjen korisniku. Integracija sistema se radi postepeno, uključivanjem komponentata za koje je utvrđeno da ispravno rade. To znači da u datom trenutku, posebno kod složenijih sistema, neke komponente su u fazi kodiranja, druge u fazi jediničnog testiranja, a treće se povezuju sa drugim komponentama i testiraju kao celina. Pri integraciji sistema treba poštovati usvojenu strategiju. Strategija pokazuje kako i zašto se komponente povezuju i kako se sistem testira. Ona utiče na redosled kodiranja i vremenski raspored uključivanja komponentata, a samim tim i na troškove na projektu.

Iako su pojedinačne komponente istestirane i potvrđeno je da ispravno rade, to nije garancija da će ispravno raditi i u sprezi sa drugim komponentama. Razloga za to ima mnogo. Najpre, može se desiti da jedan modul nepovoljno utiče na drugi modul, pa dolazi do gubitka nekih podataka. Osim toga, korišćenje globalnih struktura podataka od strane više modula može da bude neadekvatno. Nepreciznost u vrednostima podataka koja je prihvatljiva u jednom modulu, može da naraste na neprihvatljivo velike vrednosti u drugom modulu.

Integraciono testiranje se obavlja po planu u okviru koga su definisani testovi kroz koje prolaze moduli koji su prošli jedinično testiranje. Rezultat integracionog testiranja je integrisani sistem koji je spreman za sistemsko testiranje.

Postoje različiti pristupi integracionom testiranju:

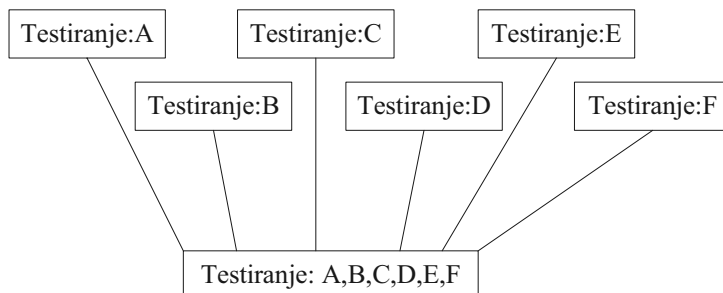
- integracija po principu „velikog praska“
- integracija od dna ka vrhu
- integracija od vrha ka dnu
- „sendvič“ integracija

Prvi pristup predstavlja metod nepostupne integracije, dok ostala tri metoda integrišu sistem postupno.

7.2.2.1 Integracija po principu „velikog praska“

Testiranje po principu „velikog praska“ može se izvršiti u slučaju kada su sve komponente od kojih se sastoji sistem uspešno prošle jedinično testiranje. Ovo testiranje je pokušaj da se ceo sistem poveže u celinu uključivanjem svih

komponenata odjednom, a zatim da se isproba da li će sve raditi kako treba. Primer ovakvog testiranja dat je na slici 7.6.



Slika 7.6 Testiranje po principu „velikog praska“

Na slici su pojedinačne komponente označene velikim slovima abecede, a sistem je dobijen povezivanjem ovih komponenata u jednu celinu.

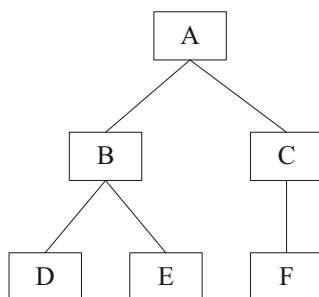
Primena ovog pristupa testiranju se ne preporučuje iz više razloga. Najpre, verovatnoća je izuzetno mala da sistem odmah proradi. Drugo, da bi sve komponente prošle jedinično testiranje, potrebno je napraviti mnogo „lažnih“ procedura. Treće, kada se otkrije greška, pošto su sve komponente integrisane odjednom, vrlo je teško pronaći njen uzrok. Poseban problem je što se greške u vezama između komponenata teško razlikuju od ostalih grešaka u sistemu.

Ipak, mnogi programeri koriste ovaj metod kod malih sistema, dok se kod velikih sistema on praktično ne primenjuje.

7.2.2.2 Integracija od dna ka vrhu

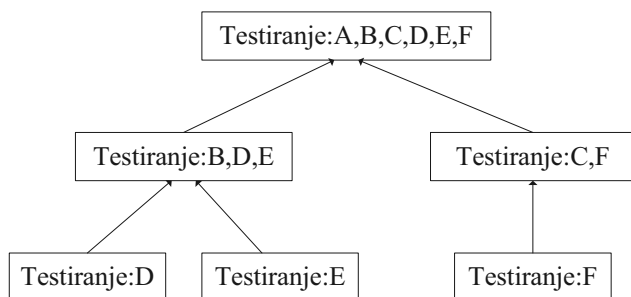
Integracija od dna ka vrhu je često korišćen pristup integracionog testiranja. Ovaj metod polazi od toga da su komponente sistema organizovane u hijerarhiju na čijem vrhu se nalazi glavni program. Testiranje otpočinje jediničnim testiranjem svih komponenata koje se nalaze na najnižem nivou u hijerarhiji. Zatim se testiraju komponente na sledećem nivou koje pozivaju prethodno istestirane komponente. Postupak se ponavlja sve dok kroz njega ne prođu sve komponente sistema.

Pretpostavimo da se integraciono testiranje treba izvršiti nad sistemom čije su komponente povezane u hijerarhiju datu na slici 7.7.



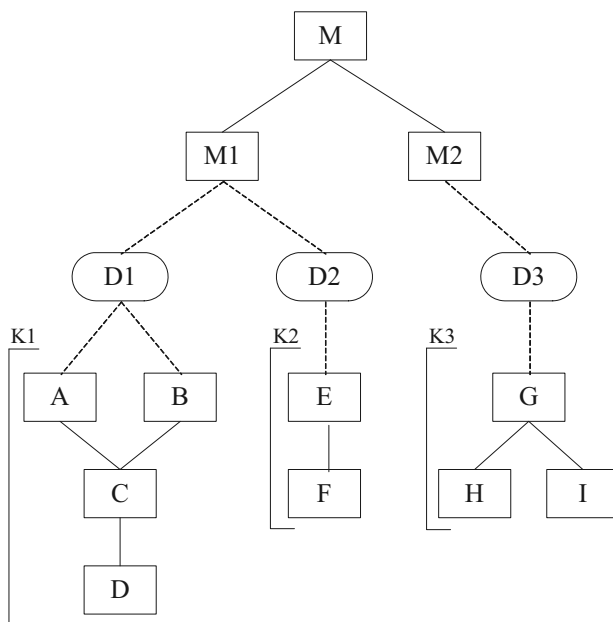
Slika 7.7 Primer hijerarhije komponenata

Testiranje od dna ka vrhu počinje testiranjem komponenata na najnižem nivou, tj. D, E i F. Pošto njih pozivaju komponente na višem nivou koje još nisu proverene, potrebno je napisati poseban programski kôd (drajver) kao pomoć pri integraciji. Ovaj kôd je obično vrlo jednostavan, ali treba paziti da se u njemu dobro definiše interfejs prema komponenti koja se testira. U datom primeru, kôd se piše za pozivanje komponenata D, E i F. Kada se utvrdi da navedene komponente ispravno rade, prelazi se na sledeći nivo u hijerarhiji. Za razliku od najnižeg nivoa, na ovom nivou, komponente se ne testiraju pojedinačno, već zajedno sa komponentama koje pozivaju (za koje je već utvrđeno da dobro rade). Tako se zajedno testiraju B, D i E, kao i C i F. Ako se, na primer, pri testiranju C i F pojavi greška, zna se da ona može da potiče samo od C ili je u interfejsu između C i F. Na slici 7.8 prikazan je redosled testiranja za dati primer.



Slika 7.8 Testiranje od dna ka vrhu

Čest je slučaj, posebno u složenijim sistemima, da se komponente na najnižem nivou ne testiraju pojedinačno, već se funkcionalno grupišu u klastere (grozdove) kao što je prikazano na slici 7.9.



Slika 7.9 Testiranje pomoću klastera

Da bi se obavilo testiranje klastera, potrebno je za svaki klaster napisati odgovarajući drajver, tj. kontrolni program za testiranje. U primeru sa slike, uvedena su tri klastera: K1, K2 i K3. Drajver D1 služi za testiranje klastera K1 koga čine komponente A, B, C i D. Na sličan način definisani su i drajveri D2 i D3. Po završetku testiranja klastera K1, drajver D1 se uklanja i klaster K1 direktno povezuje sa nadređenim modulom M1. Slično se postupa i sa drajverima D2 i D3. Na kraju se M1 i M2 integrišu sa M.

Testiranje od dna ka vrhu ima svoje nedostatke. Najveći nedostatak je taj što se komponente najvišeg nivoa, koje su obično najvažnije, testiraju poslednje. Naime, komponente na najnižem nivou uglavnom realizuju ulazne i izlazne funkcije ili neka izračunavanja koja se ponavljaju. S druge strane, komponente na višim nivoima upravljaju glavnim aktivnostima u sistemu. Stoga, mnogi projektanti smatraju da se ovakvim postupkom testiranja nalaženje značajnih grešaka odlaže do završetka testiranja, što nije dobro, jer greške mogu da nastanu kao posledica lošeg dizajna, pa bi ih trebalo što pre ispraviti.

Integracija od dna ka vrhu se koristi kada većina komponenata na najnižem nivou predstavlja funkcije opšte namene koje se pozivaju iz drugih modula, zatim kod objektno-orijentisanog pristupa, ili kada se u sistemu nalazi veći broj nezavisnih komponenata sa višekratnom upotrebom.

7.2.2.3 Integracija od vrha ka dnu

Integracija od vrha ka dnu je postupna integracija pri kojoj se polazi od testiranja glavnog programa koji se nalazi na vrhu hijerarhije komponenti. Zatim se komponente koje on poziva kombinuju i testiraju kao celina, sve dok se ne istestiraju sve komponente uključujući i one na najnižem nivou.

U ovom pristupu, komponenta koja se testira može da poziva komponentu na nižem nivou koja još nije proverena. Stoga je potrebno napisati stab, tj. kontrolni program za testiranje, koji simulira neproverenu komponentu. Stab odgovara na pozivanje i daje izlazne podatke tako da obezbeđuje kontinuitet testiranja.

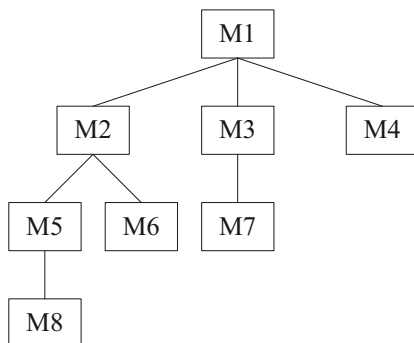
Komponente se u proces testiranja mogu uključivati na dva načina:

- strategijom „po dubini“
- strategijom „po širini“

Strategija „po dubini“ podrazumeva da redosled uključivanja komponenata u proces testiranja prati glavnu kontrolnu putanju u strukturi programa. Izbor glavne putanje je donekle proizvoljan i zavisi od specifikacije softvera.

Strategija „po širini“ u proces testiranja uključuje komponente redom po nivoima, sukcesivno.

Na primer, neka se sistem sastoji od hijerarhije modula kao što je dato na slici 7.10.



Slika 7.10 Primer modularnog sistema

Ovaj sistem se testira strategijom „po dubini“ tako što se najpre proveravaju moduli u putanji levo, tj. M1, M2 i M5. Zatim se integriše modul M8, ili M6 (radi ispravnog funkcionisanja M2). Na kraju se integrišu moduli na srednjoj (M1, M3 i M7) i krajnje desno putanji (M1 i M4).

Isti sistem se primenom strategije „po širini“ testira tako što se najpre ispituje modul M1, a onda M2, M3 i M4. Zatim se integrišu moduli na trećem nivou, tj. M5, M6 i M7, i na kraju modul M8.

Integraciono testiranje od vrha ka dnu izvodi se na sledeći način:

- najpre se testira glavni modul sa vrha hijerarhije, pri čemu se svi moduli koje on poziva zamenjuju stabovima
- u zavisnosti od korišćene strategije, podređeni stabovi se (nakon uspešnog prolaska kroz testove) u odgovarajućem redosledu zamenjuju stvarnim komponentama
- postupak se ponavlja dok se u sistem ne integrišu sve komponente

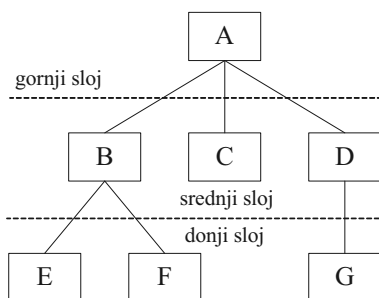
Prednost integracije od vrha ka dnu je u tome što se u ranoj fazi testiranja mogu ispitati ključne funkcije sistema. Svaka funkcija se testira počevši od najvišeg nivoa kontrole, po odgovarajućoj putanji do dna hijerarhije. Problem kod ovog načina testiranja može da bude u generisanju velikog broja stabova koji moraju da podrže sve moguće uslove u kojima simulirana komponenta treba da radi. Zato stabovi mogu u velikoj meri da utiču na ispravnost celog sistema.

7.2.2.4 „Sendvič“ integracija

„Sendvič“ integracija kombinuje dva ranije opisana pristupa integraciji: od dna ka vrhu i od vrha ka dnu. Po ovom postupku, sistem se organizuje u tri sloja (slično sendviču). U sredini se nalazi ciljani sloj koji obuhvata komponente izabrane na osnovu karakteristika i hijerarhije sistema. Gornji sloj sadrži komponente koje se testiraju od vrha ka dnu, a donji sloj komponente koje se testiraju od dna ka vrhu. To znači da, s obzirom na ovakvu organizaciju, testiranje uvek konvergira ka srednjem sloju. U donjem sloju se obično nalazi veliki broj pomoćnih komponenta opšte namene.

Neka je sistem predstavljen hijerarhijom kao na slici 7.11. „Sendvič“ integracija počinje testiranjem najpre komponente A, a zatim proverom modula E, F i G. Testiranje gornjeg i donjeg nivoa može da se radi paralelno. Nakon toga, testiraju se zajedno A, B, C i D, pa B, E i F, onda D i G i na kraju sve komponente u sistemu.

„Sendvič“ integracija objedinjuje dobre osobine integracije od vrha ka dnu i od dna ka vrhu, tako što u ranoj fazi testira i kontrolne komponente i pomoćne programe.



Slika 7.11 Struktura sistema po slojevima

7.2.3 Sistemsko testiranje

Sistemsko testiranje je najviši, završni nivo testiranja. Njime se proverava da li se sistem, kao celina, ponaša u skladu sa specifikacijom zahteva koje je postavio kupac. Pošto je većina funkcionalnih zahteva već proverena na nižim nivoima testiranja, sada je naglasak na nefunkcionalnim zahtevima, kao što su brzina, pouzdanost, efikasnost, veze prema drugim aplikacijama i okruženju u kome će se sistem koristiti.

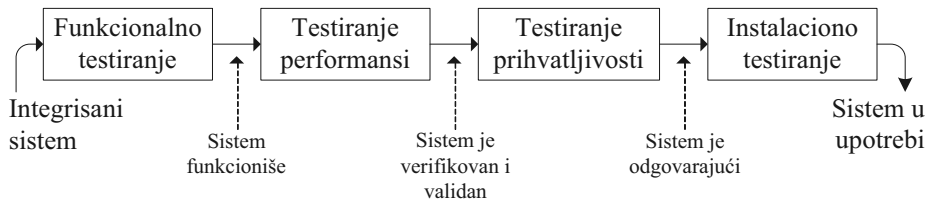
Testiranje sistema se obavlja u drugačijim uslovima nego kod jediničnog ili integracionog testiranja. Kod jediničnog testiranja, pojedinac koji testira imao je potpunu kontrolu nad procesom. Sâm je generisao test primere, pisao test procedure i vršio testiranje. Prilikom integracije komponenta, postojala je saradnja sa drugim članovima tima za testiranje ili tima za razvoj. Kada se testira sistem, u proces se mora uključiti ceo razvojni tim pod kontrolom rukovodioca projekta.

Za razliku od ranije opisanih nivoa testiranja čiji je cilj bio da se ispita da li napisani kôd radi ono što od njega očekuju projektanti, kod sistemskog testiranja se proverava da li sistem radi ono što želi kupac.

Testiranje sistema obuhvata nekoliko koraka:

- funkcionalno testiranje
- testiranje performansi
- testiranje prihvatljivosti
- instalaciono testiranje

7.12. Svaki od navedenih koraka ima drugačiju svrhu, kao što je prikazano na slici



Slika 7.12 Postupak sistemskog testiranja

Ulaz u proces sistemskog testiranja je integrisani sistem koji je prošao jedinično i integraciono testiranje. U procesu sistemskog testiranja, najpre se proverava funkcionalnost integrisanog sistema prema zahtevima koje je postavio kupac. Na primer, ako je softver namenjen analizi investicija, proverava se da li pravilno obračunava troškove osnovnih i obrtnih sredstava, kredite, troškove amortizacije i dr. Kada se završi funkcionalna provera, prelazi se na proveru nefunkcionalnih zahteva po pitanju performansi. U datom primeru, to može da bude ispitivanje brzine izračunavanja, preciznosti dobijenih rešenja, preduzetih mera bezbednosti, vremena odziva, itd. Po završetku ovog koraka dobija se verifikovan i validan sistem. Međutim, do sada su u procesu testiranja učestvovali samo oni koji su razvijali sistem, tako da sistem predstavlja rezultat njihovog shvatanja problema. Kupci, takođe, testiraju sistem i proveravaju da li on odgovara njihovom shvatanju zahteva. Ovaj test se naziva testom prihvatljivosti. Test prihvatljivosti može da se radi na mestu gde je softver razvijan, ili na krajnjoj lokaciji gde će raditi. Ukoliko se testira tamo gde je rađen razvoj, potrebno je naknadno uraditi instalacioni test na krajnjoj lokaciji. U svakom slučaju, testiranje se mora obaviti i u operativnom okruženju (na krajnjoj lokaciji), kako bi korisnici još jednom ispitali funkcionisanje sistema i prijavili dodatne probleme, ako se pojave.

7.2.3.1 Funkcionalno testiranje

Funkcionalno testiranje se izvodi po principu sličnom „crnoj kutiji“. Njime se proverava da li sistem obavlja neku funkciju, ne vodeći računa o tome koja je komponenta u sistemu za nju odgovorna. Mnoge funkcije su već proverene u ranijim fazama testiranja.

Sistemi obavljaju neki skup funkcija. Kod složenijih sistema, ovaj skup može da sadrži veoma veliki broj funkcija. Efikasnost funkcionalnog testiranja u znatnoj meri zavisi od redosleda u kome se funkcije testiraju. Pri određivanju redosleda treba voditi računa o prirodnoj ugnježđenosti funkcija.

Funkcionalno testiranje obavlja tim nezavisan od projekatanta i programera koji su realizovali sistem. Timu su poznati ulazi i očekivani izlazi ili akcije i on testira sistem kako za ispravne, tako i za neispravne ulazne podatke. Verovatnoća otkrivanja grešaka kod ove vrste testiranja je velika. Uglavnom se detektuju greške koje bi korisnik sâm otkrio prilikom upotrebe softvera.

Tokom funkcionalnog testiranja nije dozvoljeno da se sistem menja kako bi se testiranje olakšalo. Pošto se testira jedna po jedna funkcija, funkcionalno testiranje može da počne pre nego što je ceo sistem realizovan.

Test primeri kod funkcionalnog testiranja se prave na osnovu specifikacije zahteva. Na primer, softver za obradu teksta može da se testira tako što se proverava kako obavlja sledeće funkcije:

- kreiranje dokumenata
- modifikovanje dokumenata
- brisanje dokumenata

U okviru svake od navedenih funkcija, testira se skup odgovarajućih podfunkcija. Na primer, modifikovanje dokumenata može se testirati sledećim skupom podfunkcija:

- dodavanje/brisanje karaktera
- dodavanje/brisanje reči
- dodavanje/brisanje pasusa
- izmena tipa/veličine fonta, itd.

7.2.3.2 Testiranje performansi

Nakon što se ustanovi da sistem radi prema specificiranim zahtevima korisnika, prelazi se na testiranje nefunkcionalnih zahteva. Testiranje performansi se bavi proverom načina na koji se funkcije izvršavaju. Performanse sistema se mere o odnosu na ciljeve koje je postavio kupac. Na primer, ako je jedna funkcionalnost sistema da izračuna neki statistički parametar za veliki uzorak podataka, ispravnost realizacije statističkog metoda se proverava tokom funkcionalnog testiranja, dok se brzina odgovora, preciznost izlaznog parametra i slično, porede sa performansama koje je dao kupac.

Testiranje performansi izvršava tim za testiranje koji zatim dobijene rezultate prezentira kupcu. Ova vrsta testiranja obuhvata sledeće vrste testova:

- *testovi konfiguracije.* Ovim testovima se ispituje ponašanje softvera u različitim hardversko/softverskim okruženjima nadevenim u zahtevima. Postoje sistemi koji imaju ceo spektar konfiguracija namenjenih različitim korisnicima. Na primer, može se definisati minimalna konfiguracija sistema koja opslužuje pojedinačnog korisnika. Ona se dalje može nadograđivati generisanjem novih konfiguracija za druge vrste korisnika. Testovi ispituju sve konfiguracije i proveravaju da li one zadovoljavaju systemske zahteve.
- *testovi opterećenja.* Ovo su testovi kojima se ocenjuje rad sistema kada se on opatereti do svojih operativnih granica u kratkom vremenskom periodu. Na primer, ako se zahteva da sistem treba da opsluži određen broj korisnika ili uređaja, testovi opterećenja analiziraju performanse sistema (najčešće protok i vreme odziva) kada su istovremeno aktivni svi ti korisnici ili uređaji. Testovi su posebno važni kod sistema koji uglavnom rade ispod maksimalnog opterećenja, ali u datim trenucima trpe vrlo veliko opterećenje. Ovim testovima se mogu ispitivati i porediti i sistemi sa različitim konfiguracijama pri istom opterećenju.
- *testovi kapaciteta.* Ovi testovi proveravaju kako sistem obrađuje velike količine podataka. To podrazumeva ispitivanje da li su strukture podataka (nizovi, tabele, liste i sl.) definisane tako da imaju dovoljan kapacitet da mogu da prihvate podatke u svim mogućim situacijama. U istom smislu se proveravaju i dužine polja, zapisa, datoteka, itd. Ovim testovima se, takođe, ispituje ispravnost rada sistema u slučaju kada skupovi podataka dostignu svoje maksimalne vrednosti.
- *testovi kompatibilnosti.* Ovi testovi se koriste kada sistem ostvaruje spregu sa drugim sistemima iz okruženja. Testovima se proverava da li je realizacija interfejsa u skladu sa zahtevima. Na primer, ako sistem treba da pristupa drugom sistemu radi pribavljanja nekog podatka, testovi ispituju brzinu i preciznost pronalaženja podatka.
- *testovi bezbednosti.* U nekim sistemima, bezbednost je od velikog značaja. Ovi testovi ispituju da li su određene funkcije dostupne isključivo onim korisnicima kojima su namenjene. Takođe se testiraju i dostupnost, integritet i poverljivost različitih vrsta podataka.
- *regresivni testovi.* Ova vrsta testiranja podrazumeva da se jednom razvijen test primer primeni više puta za testiranje istog softvera. To se obično radi posle neke izmene u softveru, kako bi se proverilo da nije došlo do lošeg rada nekih funkcija koje nisu bile obuhvaćene izmenom. Regresivni testovi se koriste i kada testirani sistem treba da zameni postojeći sistem, kao i prilikom faznog razvoja softvera.

- *vremenski testovi*. Ovi testovi proveravaju zahteve koji se odnose na vremena izvršenja pojedinih funkcija i vremena odziva. Obično se rade u kombinaciji sa testovima opterećenja, kako bi se ispitalo da li su vremenska ograničenja zadovoljena i u ekstremnim uslovima rada, tj. pri maksimalnom opterećenju.
- *testovi okruženja*. Svrha ovih testova je da analiziraju sposobnost sistema da radi na lokaciji na kojoj je instaliran. Ako su u zahtevima postavljene granice tolerancije na temperaturu, vlagu, električna i magnetna polja, prisustvo hemikalija, radioaktivnost i sl., ovim testovima se proverava da li su te granice ispoštovane.
- *testovi oporavka*. Ovi testovi ispituju kako sistem reaguje na pojavu grešaka u smislu gubitka napajanja, gubitka podataka, uređaja ili usluga. Ispitivanja se rade tako što se sistem izlaže gubitku određenog resursa, a zatim se posmatra kako se od toga oporavlja.
- *testovi upotrebljivosti*. Testovi analiziraju zahteve vezane za interakciju korisnika sa sistemom. Ocenjuju estetski aspekt aplikacije kroz izgled ekrana, konzistentnost korisničkog interfejsa, informativnost poruka koje se daju korisniku, formate izveštaja, itd. Osim ovoga, ispituje se da li korisničke procedure zadovoljavaju zahteve po pitanju lakoće korišćenja.
- *testiranje dokumentacije*. Ovom vrstom testiranja se proverava da li su svi predviđeni dokumenti zaista napisani. To su uglavnom uputstvo za korisnika, uputstvo za održavanje i tehnička dokumentacija. Osim što se proverava da li postoje, analizira se i sadržaj dokumenata kako bi se utvrdilo da li su konzistentno i precizno napisani, tj. da li su upotrebljivi. Ukoliko se u zahtevima precizira format dokumenata, potrebno je proveriti i da li je on ispoštovan.
- *testovi održavanja*. Ako postoji zahtev da se obezbede logičke šeme, dijagnostički programi, snimci memorije, praćenje transakcija i druga pomagala koja olakšavaju nalaženje grešaka, potrebno je sprovesti testove održavanja. Njima se proverava da li ova pomagala postoje i da li ispravno rade.

7.2.3.3 Testiranje prihvatljivosti

Funkcionalno testiranje i testiranje performansi garantuju da sistem zadovoljava sve funkcionalne i nefunkcionalne zahteve koji su navedeni u početnoj fazi razvoja softvera. Ostaje da se kupac konsultuje za mišljenje.

Testiranje prihvatljivosti podrazumeva da se kupcima i korisnicima omogućiti da se sami uvere da li napravljeni softver zaista zadovoljava njihove potrebe i očekivanja. Testove prihvatljivosti pišu, izvode i procenjuju korisnici, a učesnici u razvoju softvera im pružaju pomoć oko tehničkih pitanja, ukoliko se to od njih zahteva.

Kupac može da proceni sistem na tri načina:

- referentnim testiranjem
- pilot testiranjem
- paralelnim testiranjem

Kod *referentnog testiranja*, kupac generiše tzv. referentne test slučajeve koji predstavljaju uobičajene uslove u kojima sistem treba da radi kada bude instaliran. Ove testove izvode obično stvarni korisnici (ili poseban tim) koji su dobro upoznati sa zahtevima i mogu da procene performanse sistema.

Referentno testiranje se koristi kada kupac ima posebne zahteve. Na primer, kupac može da angažuje dva razvojna tima za realizaciju softvera prema njegovoj specifikaciji. Kada sistemi budu gotovi, kupac može nad njima da sprovede referentno testiranje. Iako oba sistema možda zadovoljavaju zahteve, referentno testiranje može da pokaže da je jedan sistem brži ili lakši za upotrebu. Rezultati testiranja pomažu kupcu da odabere sistem koji će koristiti.

Pilot testiranje podrazumeva instalaciju sistema na probnoj lokaciji. Zatim korisnici rade na sistemu kao da je on već u upotrebi. Kod ove vrste testiranja, ne prave se posebni test slučajevi, već se testiranje sprovodi simulacijom svakodnevnog rada na sistemu. Iako kupac može da pripremi spisak funkcija koje bi korisnik trebalo da uključi u tipičan svakodnevni posao, testiranje je ipak manje zvanično.

Kod sistema koji se isporučuju velikom broju korisnika, pilot testiranje se obično sprovodi u dve faze. Sistem najpre testiraju korisnici iz organizacije koja je razvila sistem (alfa test), a zatim pilot testiranje radi stvarni korisnik (beta test). Na primer, ako je reč o novoj verziji nekog operativnog sistema, alfa test se radi u prostorijama onoga ko je razvio softver, a beta test kod posebno izabrane grupe kupaca.

Paralelno testiranje se koristi u faznom razvoju, kada jedna verzija softvera zamenjuje drugu, ili kada novi sistem treba da zameni stari. Ovaj način testiranja podrazumeva da paralelno (istovremeno) rade dva sistema (stari i novi). Korisnici se postepeno privikavaju na novi sistem, ali i dalje koriste i stari. Tako korisnici

mogu da uporede ova dva sistema i provere da li je novi sistem efikasniji i delotvorniji od starog.

Izbor načina na koji će se obaviti test prihvatljivosti zavisi od konkretnog sistema i od želja kupca.

7.2.3.4 Instalaciono testiranje

Završnu fazu procesa testiranja predstavlja instalaciono testiranje. Ono se sprovodi instaliranjem softvera na lokaciji na kojoj će se stvarno koristiti.

Instalaciono testiranje otpočinje konfigurisanjem sistema u skladu sa okruženjem. Zatim se sistem povezuje da potrebnim brojem uređaja iz okruženja i uspostavlja se komunikacija između njih. Sledi alokacija potrebnih datoteka i postavljanje kontrole pristupa pojedinim funkcijama sistema.

Instalacioni testovi se rade u saradnji sa korisnicima da bi se utvrdilo šta je potrebno posebno proveriti na datoj lokaciji. Ispituje se da li uslovi koji postoje na lokaciji utiču na neke funkcionalne ili nefunkcionalne osobine sistema. Može se izvršiti i regresiono testiranje kako bi se proverilo da li je sistem ispravno instaliran i da li i na licu mesta radi ono što je radio prilikom ranijeg testiranja.

Test slučajevi potvrđuju kupcu da je sistem ispravan i kompletan. Kada kupac postane zadovoljan rezultatima, testiranje se završava i sistem se formalno isporučuje.

7.3 Proces testiranja

Proces testiranja softvera može da bude vrlo zahtevan i složen. Teškoće mogu da nastanu iz različitih razloga. One mogu da budu posledica korišćene hardverske ili softverske platforme, ili da potiču iz postupaka implementiranih u samom sistemu. Posebno teško može da bude testiranje složenijih sistema, kao što su distribuirani sistemi ili sistemi koji rade u realnom vremenu. Kod projekata na kojima radi veliki broj ljudi, dodatan problem predstavlja koordinacija razvojnog tima i tima za testiranje. Da bi se teškoće uspešno prevazišle, testiranju se mora pristupiti na sistematičan i dobro organizovan način. Potrebno je definisati standardizovane propise i procedure načina rada tima za testiranje, kao i ostalih učesnika u razvoju. Sve aktivnosti tima za testiranje moraju da budu dokumentovane.

Tim za testiranje može da ima različit broj članova u zavisnosti od složenosti projekta. Veći projekti zahtevaju više ljudi sa dobro definisanim zaduženjima. U brojnijim timovima za testiranje mogu se izdvojiti sledeće uloge:

- menadžer kvaliteta
- član tima koji razvija testove
- član tima koji testira softver
- menadžer zahteva za izmenama

Menadžer kvaliteta je odgovoran za planiranje testiranja. On kontroliše proces testiranja, proverava rokove realizacije i predlaže potrebne korekcije. Obezbeđuje neophodan materijal timu za testiranje, formira izveštaje i ostalu neophodnu dokumentaciju. Takođe informiše rukovodstvo projekta o rezultatima rada tima za testiranje.

Član tima koji razvija testove generiše test primere i serije testova. Član tima koji testira softver izvršava ove testove (kako na nivou pojedinačnih modula, tako i na nivou celog sistema) i šalje izveštaje sa prijavljenim greškama i otkazima.

Menadžer zahteva za izmenama analizira rezultate testova i na osnovu njih formira zahteve za potrebnim izmenama, dodeljuje im prioritete i planira redosled njihovog sprovođenja po iteracijama.

Proces testiranja softverskog proizvoda sprovodi se kroz sledeće aktivnosti:

- izrada plana testiranja
- specifikacija testova
- realizacija testiranja
- evaluacija rezultata testiranja

Navedene aktivnosti uključuju eksplicitno zadavanje odgovornosti i zaduženja po pitanju planiranja testova, njihovog sprovođenja i evaluacije dobijenih rezultata testiranja. One ukazuju na vrste testova koji će biti primenjeni, njihove ciljeve i raspored izvršenja testova.

7.3.1 Plan testiranja

Plan testiranja je dokument u kome je opisana organizacija procesa testiranja. Zahvaljujući njemu, tim za testiranje može da bude siguran da je sistem istestiran

sveobuhvatno i na odgovarajući način. Svaki korak u procesu testiranja mora da bude dat u planu.

Plan testiranja se pravi na osnovu poznavanja projektnih zahteva, dizajna sistema i projekta kôda. Testiranje prolazi kroz različite faze počevši od jediničnog, preko integracionog, do sistemskog testiranja.

Za svaku fazu u planu, najpre se navodi cilj. U zavisnosti od ciljeva, identifikuju se zahtevi za testiranjem koji ukazuju na to šta treba da bude istestirano. Zahtevi se odnose na funkcije sistema, performanse sistema i pouzdanost sistema. Zahtevi za testiranje funkcija sistema izvede se iz ranije utvrđenih funkcionalnih zahteva. Dobra praksa je da se za svaki slučaj korišćenja generiše bar po jedan funkcionalni zahtev. Zahtevi za testiranjem performansi obično uključuju proveru vremena odziva i potrošnje resursa u različitim režimima rada. Zahtevi za pouzdanošću se izvede na osnovu nefunkcionalnih zahteva projektne i implementacione dokumentacije. Na osnovu zahteva, projektuju se test slučajevi, o čemu će biti reči u narednom poglavlju.

Sledeća stavka u planu testiranja jeste utvrđivanje redosleda testiranja. Da bi se formirao redosled, potrebno je odrediti prioritete komponenata koje će biti testirane. U tome može da pomogne procena rizika. Najrizičnije komponente u pogledu otkaza treba da budu testirane među prvima. Dalje, što se komponenta češće koristi, posebno od strane različitih korisnika, trebalo bi da ima veći prioritet. Tako se mogu definisati prioriteti komponenata, kao što su H (*high*) za najveći prioritet, M (*medium*) za srednji i L (*low*) za najniži prioritet. Sada se redosled testiranja pravi tako što se komponente sa H prioritetom prve testiraju i to obavezno, zatim bi trebalo testirati komponente sa M prioritetom, dok se komponente sa L prioritetom mogu testirati, ali tek pošto se istestiraju prethodne dve kategorije.

U planu testiranja, za svaki test treba navesti šta će se njime ispitivati, koji je kriterijum uspešnosti testa, koja osoba će raditi testiranje, kog dana i u koje vreme. Takođe je potrebno navesti i alate koji će biti korišćeni za testiranje (na primer, računarska konfiguracija Intel Core i5 Quad Core 2.5 GHz, 4 GB RAM, 1333MHz DDR3, Mac OS X Snow Leopard), kao i standarde i procedure.

Na kraju, treba definisati kriterijume za završetak testiranja. Oni određuju u kom trenutku se neki program može proglasiti dovoljno dobrim da nisu potrebna dalja testiranja.

7.3.1.1 Završetak testiranja

Pitanje završetka testiranja je vrlo važno, posebno kada se ima u vidu da je praktično nemoguće napraviti softver bez grešaka. Odgovori na pitanje kada treba prestati sa testiranjem mogu da budu različiti. Trivijalni odgovori, kao što su kada

istekne vreme planirano za testiranje, ili kada se uspešno izvrše svi planirani test primeri, pokazali su se lošim. Bolje je kao kriterijum postaviti izabrane metode testiranja kroz koje sistem mora da prođe. Ipak, ni ovo nije dovoljno dobro, jer su neke metode pogodnije za jedne, a druge metode za druge sisteme.

Efikasniji kriterijum je da se sistem testira dok se ne pronađe zadati broj grešaka. Naravno, ovaj broj grešaka nije proizvoljan, već se dobija na osnovu iskustva, ili iz matematičkih modela. Jednu od tehnika za određivanje broja grešaka u programu razvio je Mills 1972.godine. To je sejanje grešaka.

Sejanje grešaka je metod koji se izvodi tako što jedan član tima za testiranje namerno ubaci (poseje) poznati broj grešaka S u program koji se testira. Zatim ostali članovi tima za testiranje pokušavaju da pronađu što je moguće više grešaka. Neka su oni otkrili s posejanih grešaka. Po ovom metodu, smatra se da je odnos otkrivenih posejanih grešaka prema broju posejanih grešaka isti kao i odnos otkrivenih neposejanih grešaka n prema ukupnom broju neposejanih grešaka u programu N , tj. važi:

$$N = Sn/s$$

Na primer, ako je u programu posejano 100 grešaka, a u testiranju pronađeno samo 70, na osnovu jednačine sledi da u programu ima oko 30% neotkrivenih izvornih grešaka.

Iako je metod sejanja grešaka vrlo jasan i jednostavan, da bi bio koristan, trebalo bi da posejane greške budu iste vrste i iste složenosti kao stvarne greške u programu. Međutim, pošto je nemoguće znati kakve greške postoje u programu dok one ne budu pronađene, vrlo je teško posejati reprezentativne greške.

Da bi se povećala verovatnoća da greške budu dovoljno reprezentativne, mogu se koristiti istorijski podaci o greškama prikupljeni u ranijim sličnim projektima. Ako ovi podaci ne postoje, onda se metod sejanja grešaka može modifikovati uvođenjem dve nezavisne grupe koje testiraju isti program. Neka je prva grupa otkrila x , a druga y grešaka od mogućih n i neka su obe grupe pronašle q zajedničkih grešaka ($q \leq x$ i $q \leq y$). Efikasnost grupe se može shvatiti kao njena sposobnost da pronađe greške iz datog skupa postojećih grešaka. U skladu sa ovim, efikasnosti (E) grupe su

$$E1 = x/n \quad \text{ i } \quad E2 = y/n$$

Ako se uzme da grupe podjednako efikasno pronalaze greške u svakom delu programa, onda važi i

$$E1 = x/n = q/y$$

$$E2 = y/n = q/x$$

pa se dobija da je ukupan broj grešaka

$$n = q/(E1E2)$$

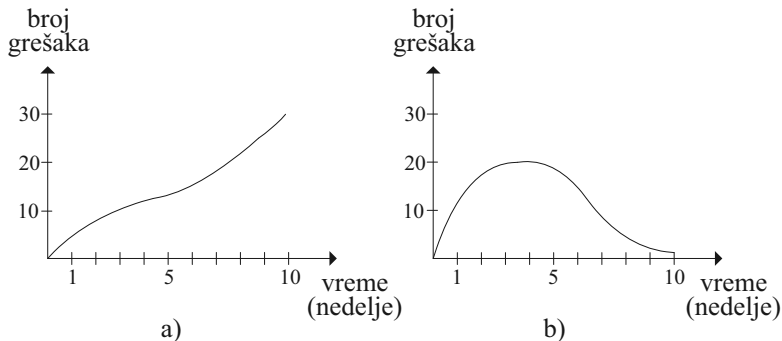
Na primer, ako je prva grupa pronašla 25, a druga 30 grešaka, s tim što su obe grupe pronašle 15 zajedničkih grešaka, onda su efikasnosti grupa:

$$E1 = q/y = 15/30 = 0,5$$

$$E2 = q/x = 15/25 = 0,6$$

pa je ukupan broj grešaka $n = 15/(0,5*0,6) = 50$.

U slučaju da broj pronađenih grešaka ne može da dostigne zadati broj, može se primeniti još jedan efikasan kriterijum, a to je vođenje evidencije o pronađenim greškama u određenom vremenskom intervalu. Tada se prave dijagrami zavisnosti između broja pronađenih grešaka i broja nedelja. Na slici 7.13 data su dva primera ovih dijagrama.



Slika 7.13 Praćenje broja grešaka u vremenu

Kao što se vidi, u slučaju pod a) broj grešaka se povećava iz nedelje u nedelju, tako da se testiranje mora nastaviti. Kriva pod b) pokazuje da se nakon 9 nedelja broj grešaka dovoljno smanjio da testiranje može da bude okončano.

Najbolji rezultati se postižu kombinovanjem svih navedenih kriterijuma, izuzimajući trivijalne.

7.3.2 Specifikacija testova

Specifikacija testova podrazumeva identifikaciju i opis skupa test primera za verifikaciju sistema, kao i test procedura koje opisuju način izvršenja test primera. Specifikacija se pravi na osnovu projektnih zahteva, modela sistema i projektne dokumentacije.

Skup testova se formira na osnovu projektnih zahteva. Jedan od načina da se to uradi jeste da se generiše tabela povezanosti testova i zahteva (tabela 7.6). U vrstama ove tabele nabrajaju se funkcije sistema koje treba da budu predmet testiranja. U kolonama se navode pojedinačni zahtevi.

Test	Zahtev 6.1: Rad sa bazom podataka	Zahtev 6.2: Dohvatanje podataka	Zahtev 6.3: Izrada izveštaja
1. Izrada indeksa		X	
2. Menjanje polja	X		
3. Brisanje polja	X		

Tabela 7.6 Tabela povezanosti testova i zahteva

Kao što se vidi, uz svaki zahtev navodi se broj zahteva u dokumentu. Svaka navedena funkcija povezana je sa onim zahtevom u čijoj koloni se nalazi oznaka X za tu funkciju.

Data tabela opisuje funkcije sistema koje će biti proverene testovima. Na sličan način se mogu opisati i testovi performansi, s tim što se tada u tabeli, umesto funkcionalnih zahteva, navode zahtevi u pogledu brzine pristupa, protoka, bezbednosti i dr.

Često se jedan test sastoji od nekoliko manjih testova koji svi zajedno treba da ispune neki zahtev. U ovom slučaju, specifikacija ukazuje na veze manjih testova sa zahtevom.

Za svaki pojedinačni test, pravi se posebna specifikacija. Na početku specifikacije navode se zahtevi čiju realizaciju test treba da proveri, tj. daje se svrha testa. Zatim se utvrđuje skup metoda pogodnih za izvršavanje konkretnog testa. Ukoliko postoje neka ograničenja za primenu ovih metoda (usled realne situacije u kojoj se vrši testiranje), navode se i stvarni uslovi testiranja. Na primer, uslovi mogu da budu:

- da li se pri testiranju koriste ulazni podaci prispeli iz stvarnog sistema, ili se oni generišu samo za testiranje

- da li postoje ograničenja za testiranje u smislu vremena, opreme, osoblja i dr.

Za svaki test se, takođe, daje na koji način će se proceniti da li je on kompletan. Navode se kriterijumi na osnovu kojih se zna kad je test završen i da li su ispunjeni relevantni zahtevi. Ako je pre procene potrebno na neki način obraditi podatke (na primer, svesti veću količinu podataka na manju, kako bi procena bila jednostavnija), u specifikaciji se navodi i način obrade. Na kraju se predlaže metod za procenu. To može da bude, na primer, da se rezultati testiranja sakupe i ručno urede, a zatim daju na pregled timu za testiranje. Tim može da koristi i automatizovane alate za procenu nekih podataka i pregled zbirnih izveštaja, ili da proverava stavku po stavku i poredi je sa očekivanim rezultatima.

Centralni deo specifikacije je opis testa. Dokument opisa testa treba da bude napisan na jasan i razumljiv način, i da sadrži dovoljno detalja, tako da bude koristan pri testiranju. On sadrži:

- sredstva kontrole koja će se koristiti pri testiranju
- podatke bitne za testiranje
- procedure koje će se primenjivati pri testiranju

Dokument započinje opštim opisom testa. Zatim se navode sredstva kontrole testiranja, u smislu da li će se testiranje pokretati i kontrolisati ručno ili automatski (na primer, da li se ulazni podaci unose preko tastature, ili će se preuzimati iz nekog programa).

Podaci bitni za testiranje obuhvataju: ulazne podatke, ulazne komande, ulazna stanja, izlazne podatke, izlazna stanja i poruke koje sistem šalje. Svaku vrstu podataka potrebno je detaljno opisati. Na primer, pri opisu ulaznih komandi treba objasniti kako se test pokreće, zaustavlja, prekida, ponavlja, nastavlja i sl.

Na kraju se opisuju procedure koje treba primeniti u testu. Navode se vrednosti koje treba uneti, radnje koje treba obaviti, ispravni, tj. očekivani rezultati ili odzivi (u tekstualnom ili grafičkom obliku), kao i opis aktivnosti koje treba obaviti nakon završetka testa (na primer, štampanje rezultata, brisanje nekih podataka, isključivanje delova opreme i sl.).

Sledi primer opisa testa za funkciju AVG koja računa srednju vrednost elemenata datog niza.

INPUT DATA:

Ulazni podaci se preuzimaju iz programa GEN_NIZ.

Program generiše niz od N prirodnih brojeva iz opsega 1 do 100.

Program se poziva sa GEN_NIZ(N) u pokretaču testiranja.

Izlaz se smešta u globalnu promenljivu A.

Skup podataka za testiranje:

slučaj 1: Upotrebiti GEN_NIZ sa parametrom N = 100

slučaj 2: Upotrebiti GEN_NIZ sa parametrom N = 500

slučaj 3: Upotrebiti GEN_NIZ sa parametrom N = 1000

INPUT COMMANDS:

Program AVG računa srednju vrednost elemenata niza A.

Program AVG se poziva sa AVG().

OUTPUT DATA:

Rezultat se štampa na ekranu računara.

SYSTEM MESSAGES:

Dok traje računanje, na ekranu se ispisuje poruka:

„Calculating...“.

Po završetku računanja, na ekranu se ispisuje poruka:

„Calculation is completed.“

Testiranje se može prekinuti pre završne poruke pritiskom na Ctrl-C na tastaturi.

7.3.3 Realizacija testiranja

Realizacija testiranja obično polazi od razvoja test skriptova koji odgovaraju test primerima. Test skriptovi moraju redovno da se ažuriraju, kako bi uvek bili konzistentni sa test primerima.

Test skriptovi predstavljaju uputsvo iz koga se jasno vidi kako se korak po korak izvršava testiranje. Oni obebeđuju potpunu kontrolu nad testiranjem, tako da se, ukoliko dođe do nekog otkaza, uslovi testiranja mogu ponoviti i sistem ponovo dovesti do tog otkaza. To je neophodno da bi se ustanovio uzrok nastalog problema.

U tabeli 7.7 dat je primer test skripta sa aktivnostima koje treba preduzeti da bi se tekst napisan u programu *Microsoft Word* boldovao.

Testiranje rade zaduženi pojedinci iz tima za testiranje polazeći od specifikacije testa. Testovi se mogu izvršavati ručno ili automatski. Tokom testiranja, dobijeni rezultati se unose u izveštaj koji će kasnije biti analiziran kako bi se procenila uspešnost testa. Izveštaj sa rezultatima testiranja je koristan iz više razloga:

- izveštaj dokumentuje rezultate testova
- ukoliko dođe do otkaza, izveštaj pruža informacije potrebne za ponavljanje testa, što je važno zbog utvrđivanja uzroka otkaza

- izveštaj pruža osnovu za stvaranje poverenja u sistem u pogledu performansi
- izveštaj sadrži informacije na osnovu kojih se može utvrditi da li je razvoj završen

Korak	Instrukcija	Očekivani rezultat	Stvarni rezultat	Komentar
1	Aktivirati aplikaciju MS Word.	MS Word je aktivan.	U redu.	Za test se koristi MS Word 2003.
2	Kreirati novi dokument.	Prazan dokument je na ekranu.	U redu.	
3	Ispisati tekst „This is what bold text looks like.“	Rečenica je uneta u dokument.	U redu.	
4	Selektovati reč „bold“.	Reč „bold“ je označena.	U redu.	
5	Pritisnuti Ctrl-B na tastaturi.	Reč „bold“ je boldovana.	U redu.	

Tabela 7.7 Primer test skripta

U okviru izveštaja postoje različiti obrasci koji sadrže podatke relevantne za pojedine aspekte testiranja. Jedan od njih je obrazac za izveštaj o greškama. U njemu se navode sve pronađene greške. Svaka greška ima svoj identifikacioni broj i detaljan opis koji sadrži:

- datum identifikovanja greške
- identitet člana tima za testiranje koji je pronašao grešku
- datum kada je pronađen i otklonjen uzrok greške
- broj časova potreban za ispravljanje greške (ako postoje drugi problemi višeg prioriteta, greška ne mora da se ispravlja odmah po otkrivanju)
- cenu otklanjanja greške
- vrstu greške
- identifikator dela sistema, tj. naziv modula ili dokumenta u kome je pronađena greška
- fazu u razvoju u kojoj je načinjena greška (analiza zahteva, projektovanje, implementacija, itd.)
- vrsta poruke ili aktivnost koja je dovela do otkrivanja greške
- opis otkaza koji je greška prouzrokovala

- mehanizam koji ukazuje na to kako je izvor greške nastao, kako je otkriven i ispravljen
- vrstu ljudskog propusta koja je dovela do greške
- ozbiljnost otkaza do koga je došlo ili je moglo da dođe

U tabeli 7.8 dat je primer dela izveštaja o greškama koji sadrži neke od navedenih podataka.

Identifikator greške	Nedelja prijave	Područje greške	Vrsta greške	Nedelja rešenja	Trajanje ispravljanja
...	...	...	...	...	...
F256	92/14	C2	P	92/17	4,5
F257	92/15	C5	I	92/16	3
...	...	...	...	...	...

Tabela 7.8 Obrazac za izveštaj o greškama

7.3.4 Evaluacija rezultata testiranja

Nakon svake faze testiranja radi se evaluacija dobijenih rezultata. Sprovedena testiranja mogu se analizirati i procenjivati na različite načine.

Jedan od načina je utvrđivanje delotvornosti ili učinka testa. Graham je 1996.godine predložio da se učinak testa meri odnosom broja grešaka koje su pronađene u datom testu i ukupnim brojem otkrivenih grešaka (uključujući i greške pronađene po završetku datog testa). Na primer, ako je tokom integracionog testiranja otkriveno 36 grešaka, a ukupan broj nađenih grešaka je 90, onda je efikasnost integracionog testiranja 40%. Međutim, ukoliko je do isporuke sistema nađeno 90 grešaka, a zatim (nakon isporuke) još 30 grešaka za prva tri meseca korišćenja softvera, onda se efikasnost integracionog testiranja smanjuje na 30%, jer je od mogućih 120 grešaka u ovoj fazi pronađeno 36. Sa protokom vremena, učinak datog testa može samo dalje da se smanjuje.

Opisani način ocenjivanja testa može da se poboljša uzimanjem u obzir činjenice da nisu sve greške podjednako ozbiljne. U tom smislu, svakom otkazu se može dodeliti stepen ozbiljnosti, koji bi zatim imao udela u računanju učinka testa. Učinak testa bi se posebno računao za različite stepene ozbiljnosti, pa bi se, na primer, moglo dobiti da je učinak integracionog testiranja u otkrivanju grešaka koje

dovode do kritičnih otkaza 30%, a u otkrivanju grešaka koje dovode do manjih otkaza 70%.

Osim učinka, korisno je proceniti i efikasnost neke faze testiranja. Efikasnost testiranja se određuje kao odnos broja grešaka pronađenih pri testiranju i rada koji je uložan u to testiranje. Stoga se efikasnost izražava u broju grešaka po čovek/satu. Efikasnost testiranja se koristi za izračunavanje troškova nalaženja grešaka.

U okviru evaluacije, može se uraditi i analiza pronađenih grešaka sa ciljem da se ukaže na česte i zajedničke greške tima koji razvija softver. Na osnovu ove analize, mogu se predložiti aktivnosti koje bi poboljšale proces razvoja softvera, kao na primer, pohađanje određenih kurseva, sticanje naprednijih znanja, promena alata korišćenih u razvoju, itd.

8 Isporuka i održavanje softvera

Isporuka softvera i njegovo održavanje predstavljaju završne faze u procesu razvoja softvera. One slede nakon faza u kojima je problem identifikovan, rešenje isprojektovano, implementirano i provereno. Pošto je softver kompletan, može se isporučiti kupcu i pustiti u rad, nakon čega sledi njegovo održavanje.

Isporuka softvera se često shvata kao formalnost u smislu instaliranja softvera na radnoj lokaciji, što nije dobra praksa. Ova faza je mnogo složenija. Od nje zavisi kako će korisnici prihvatiti sistem i u kojoj meri će njime biti zadovoljni. Cilj isporuke je, osim instaliranja softvera, da se korisnicima pomogne da u potpunosti razumeju kako softverski proizvod funkcioniše da bi se osećali udobno pri radu u novom okruženju. Ukoliko isporuci nije posvećena dovoljna pažnja, može se desiti da korisnici ne primenjuju softver na najbolji način, što može da umani njihovu produktivnost i efikasnost.

Nakon isporuke, životni vek softvera se nastavlja sve dok je on u upotrebi. U tom periodu, softver neprestano evoluirao kako kroz otklanjanje zaostalih grešaka, tako i kroz razna poboljšanja i modifikacije koji su predmet održavanja.

8.1 Isporuka sistema

Tokom razvoja sistema, mora se imati u vidu činjenica da je svaki softverski proizvod pre svega namenjen krajnjem korisniku. Da bi krajnji korisnik mogao lako i efikasno da upotrebljava sistem, potrebno je ranije isplanirati i razviti sredstva koja bi mu u tome pomogla. Sredstva koja najviše mogu da doprinesu boljem i lakšem radu korisnika su: obuka za korišćenje softvera i prateća dokumentacija.

8.1.1 Obuka korisnika

Softverski sistemi obično imaju dve vrste korisnika: krajnje korisnike i operatere (administratore). Krajnji korisnici izvršavaju osnovne funkcije sistema u cilju rešavanja problema opisanih u projektnim zahtevima. Operatori izvršavaju pomoćne funkcije sistema koje služe za podršku osnovnim funkcijama. Na primer, krajnji korisnici mogu da izvršavaju aktivnosti predviđene rešenjem nekog problema (uključujući razna izračunavanja i algoritme), da unose i analiziraju podatke, generišu izveštaje, dijagrame i sl. Operateri pružaju administrativnu podršku u smislu odobravanja prava pristupa pojedinim delovima sistema, periodičnog pravljenja kopija datoteka bitnih za sistem, instaliranja novih uređaja i softvera, izvršavanja aktivnosti za oporavak sistema nakon nekih neregularnih stanja (nestanka napajanja, nehotičnog brisanja delova sistema, itd.). U nekim situacijama krajnji korisnik i operator mogu da budu ista osoba.

Pošto krajnji korisnik i operator imaju različite zadatke, obuci se pristupa sa dva aspekta u zavisnosti od toga kome je namenjena.

Obuka krajnjeg korisnika podrazumeva njegovo upoznavanje sa načinom pokretanja i izvođenja osnovnih funkcija sistema. U tom smislu, korisnik se najpre upoznaje sa korisničkim interfejsom, rasporedom opcija, izgledima ekrana i značenjem pojedinih oznaka na ekranu (na primer, boja, simbola i sl.). Iako se smatra da krajnji korisnici u dovoljnoj meri poznaju domen i funkcionalnost sistema, dešava se da sistem implementira i neke nove funkcije sa kojima treba upoznati korisnika. Za krajnjeg korisnika je važno da razume funkcije sistema i da nauči kako da ih koristi. On ne mora da poznaje unutrašnji rad sistema. Na primer, korisnik mora da zna da može da sortira neke podatke po željenim kriterijumima, ali ne mora da poznaje algoritme koji su korišćeni za sortiranje. Takođe, krajnji korisnik ne mora da bude upoznat ni sa funkcijama podrške koje obezbeđuju bolje performanse sistema. Na primer, za korisnika nije bitno ko, osim njega, trenutno pristupa sistemu, ili na kom serveru se nalazi deo aplikacije.

U praksi se često stari sistem zamenjuje novim. U ovim slučajevima, tokom obuke krajnjeg korisnika, može da dođe do problema u vezi sa prihvatanjem novog sistema. Naime, korisnici su često prisiljeni da odbace ono što znaju i na šta su se već navikli, pa da prihvate nešto novo. Tada se kod njih može stvoriti odbojnost prema novom sistemu, koja može da utiče na normalan rad. U ovim situacijama, mora se korisnicima dati dovoljno vremena da postupno prihvate novi način rada, uz davanje objašnjenja zašto je novi pristup bolji od starog.

Obuka operatera ima za cilj da operater ovlada funkcijama za podršku radu sistema. Tokom obuke, operater se upoznaje sa organizacijom sistema, a ne sa funkcijama za rešavanje problema iz domena za koji je softver napravljen. To znači

da operater ne mora da zna šta sistem radi (domen problema), već kako radi (način funkcionisanja).

Obuka operatera obuhvata dva aspekta: uspostavljanje uslova za normalan rad sistema i podršku krajnjim korisnicima. Operateri najpre moraju da nauče:

- da aktiviraju novi sistem i puste ga u rad
- da konfiguriraju sistem, tj. podese zadati skup parametara od kojih zavisi efikasnost rada sistema
- da administriraju prava pristupa sistemu
- da raspodeljuju resurse korisnicima (zadatke, prostor na disku i sl.)
- da nadgledaju i podešavaju performanse sistema

Operateri, takođe, treba da nauče kako da pomognu krajnjim korisnicima u slučaju pojave nekih neregularnih situacija, kao što su neželjeni gubitak podataka ili datoteka, otkazi, problemi u komunikaciji, problemi u obraćanju drugim sistemima, itd.

Obuka korisnika se obavlja putem kurseva na kojima se postupno, ali detaljno polaznici upoznaju sa sistemom. Obuka se može izvoditi na različite načine u zavisnosti od prirode sistema, sposobnosti predavača ili afiniteta polaznika. U izvođenju obuke mogu biti od koristi sledeća sredstva:

- dokumentacija
- lekcije i demonstracije
- stručni korisnici

Da bi bio kompletan, svaki sistem mora da ima prateću formalnu dokumentaciju. U dokumentima su date sve informacije potrebne za ispravan i efikasan rad sistema. Iako je dokumentacija prvenstveno namenjena za korišćenje u fazi eksploatacije sistema, može da bude korisna i u fazi obuke. Međutim, istraživanja su pokazala da samo 10-15% polaznika obuke čita dokumentaciju koja im se stavi na raspolaganje. Nakon šest meseci od obuke, ustanovljeno je da više niko ne čita dokumentaciju.

Polaznici obuke znatno češće koriste pomoć u elektronskoj formi (*Help*), ukoliko im je dostupna na računaru. Na taj način štede vreme koje bi potrošili na pretraživanje dokumenata u papirnoj formi.

Istraživanja su pokazala da se obuka znatno pojednostavljuje ukoliko se korisnički interfejs softvera projektuje tako da funkcije budu razumljive i da im se lako pristupa. Najpogodniji način za pristup funkcijama su ikone. One se pamte

lakše od opcija u menijima (slika se povezuje sa temom), a jednostavnije se i pokreću (dvostrukim pristiskom na dugme miša).

Bez obzira na eventualnu elektronsku pomoć, polaznici obuke rado slušaju lekcije i demonstracije koje služe za predstavljanje pojedinih aspekata sistema putem niza kratkih prezentacija ili predavanja. Lekcije i demonstracije su vrlo dinamične i fleksibilne (živa reč), pa su zato zanimljivije od čitanja dokumentacije. Funkcije sistema se najpre objašnjavaju u okviru lekcija, a zatim se isprobavaju putem demonstracija. Efikasnost demonstracija se povećava ako se one izvode na računaru ili na *web*-u. Neki programi obuke uključuju i multimedijalne elemente (audio i video snimke). Obuka se obično izvodi u računarskim učionicama, pri čemu svaki polaznik raspolaže sopstvenim računarom. Postoje specijalizovani softveri koji omogućavaju predavačima da nadgledaju šta svaki od polaznika radi na svom računaru, kao i da preuzmu kontrolu nad korisničkim računarom kako bi ukazali na niz komandi koje polaznik treba da izvrši.

Lekcije i demonstracije su obično vrlo uspešne, jer predavač odmah dobija povratne informacije od polaznika, pa može da im pomogne u prevazilaženju nastalih problema.

Značajnu pomoć pri obuci mogu da pruže i stručni korisnici. To su pojedinci koji su se pre obuke obučili da rade na sistemu, pa na obuci imaju ulogu demonstratora ili pomoćnika predavača. Stručni korisnici su obično dobro prihvaćeni od strane korisnika, jer korisnici u njima nalaze slične sebi. Korisnici su znatno opušteniji u radu sa stručnim korisnicima, postavljaju im pitanja, diskutuju sa njima o problemima, i što je takođe važno, shvataju da je neko drugi već savladao rad na sistemu, što im uliva samopouzdanje. Osim ovoga, stručni korisnici mogu da ukažu na mesta gde su oni imali najviše problema i kako su te probleme prevazišli.

Stručni korisnici najbolje razumeju potrebe korisnika, tako da mogu da daju najbolje izveštaje o tome koliko su korisnici zadovoljni sistemom, da li ima potrebe za dodatnom obukom, šta je bilo najteže, a šta najlakše u savladavanju načina funkcionisanja sistema i sl.

Obuka se smatra uspešnom ukoliko zadovolji potrebe polaznika. Međutim, i sami polaznici mogu znatno da se razlikuju. Na primer, mogu da imaju različita predznanja, da im odgovaraju različite metode učenja (čitanje, slušanje, ponavljanje i dr.), da imaju različite veštine (kucanje na tastaturi, poznavanje principa rada na računaru). Stoga bi bilo dobro (ako je moguće) da se obuka u određenoj meri prilagodi individualnim polaznicima. To se može postići podelom prezentacija u manje celine, tako da korisnici ne bi morali da uče ono što već znaju.

Iako se tokom obuke polaznici detaljno upoznaju sa celim sistemom, dešava se da nešto propuste ili ne shvate. Takođe, ako nakon obuke pri radu ne koriste pojedine funkcije, vremenom ih zaborave. Na primer, ako operater radi arhiviranje nekih podataka jednom godišnje, može se desiti da sledeće godine zaboravi proceduru arhiviranja. Ovi problemi se prevazilaze ili ponovnom obukom (može i ponavljanjem dela obuke) ili čitanjem dokumentacije.

8.1.2 Dokumentacija

Softverski proizvod je u potpunosti završen tek ako ima prateću dokumentaciju namenjenu korisnicima. Iz ove dokumentacije, korisnici se mogu informisati o svim karakteristikama sistema. Dobra dokumentacija, takođe, pruža pomoć korisnicima u prevazilaženju neregularnih situacija do kojih može da dođe u radu sistema.

Za dokumentaciju je važno da bude napisana na jasan, jednostavan i pregledan način, da bude konzistentna (usaglašena) i aktuelna (mora u potpunosti da odgovara sistemu koga dokumentuje). Dobra preglednost se postiže uvođenjem indeksa u vidu ključnih reči uparenih sa brojevima stranica na kojima se ključne reči nalaze u dokumentima.

Dokumentaciju može da čini veliki broj dokumenata u zavisnosti od njene namene i realnih potreba. Pre izrade dokumentacije, treba utvrditi skup potrebnih dokumenata i kome su oni namenjeni: korisnicima, operaterima, osoblju iz systemske podrške, itd. U nastavku je opisano nekoliko vrsta dokumenata koji se u praksi najčešće koriste.

Uputstvo za korisnika je dokument namenjen onima koji treba da koriste sistem, tj. izvršavaju njegove funkcije.

Na početku uputstva za korisnika je naslovna strana (*cover page*) sa naznakom o kom dokumentu se radi i strana sa informacijama o autorskim pravima (*copyright page*). Nakon toga, sledi predgovor u kome se opisuje svrha uputstva. Ovo je vrlo bitna informacija, jer se na osnovu nje korisnik uverava da će u dokumentu zaista naći ono što mu je potrebno. U predgovoru se, takođe, daju i detalji o dokumentima i informacijama bitnim za korišćenje uputstva. Radi lakše upotrebe, navode se skraćenice, akronimi i specijalni termini koji se koriste u priručniku. Zatim sledi sadržaj dokumenta. Posle sadržaja, pristupa se opisivanju sistema. Najpre se u nekoliko pasusa daje opšti pregled sistema navođenjem sledećih stavki:

- namena i ciljevi sistema
- osobine i prednosti sistema

- funkcionalne mogućnosti sistema (uključujući veze koje postoje između funkcija)

Radi lakšeg razumevanja rada sistema, osim teksta, u opisima se često koriste i slike, tabele, dijagrami i druga sredstva. Na primer, struktura sistema na opštem nivou se može efikasno predstaviti dijagramom u kome su jasno naznačeni ulazni podaci i njihovi izvori, izlazni podaci i njihova odredišta, kao i osnovne funkcije sistema. Takođe, funkcije sistema se mogu razvrstati po nivoima i prikazati tabelarno.

Nakon preglednog upoznavanja sa sistemom, sledi predstavljanje pojedinačnih funkcija. Funkcije se opisuju sa stanovišta korisnika, tj. tako da korisnik može da shvati šta se njima postiže i na koji način. Detalji o tome kako sistem izvršava funkcije ne navode se u uputstvu za korisnika. To znači, da korisnik treba da nauči šta, ali ne i kako sistem radi.

Opis pojedinačne funkcije treba da sadrži sledeće stavke:

- naziv funkcije
- opis svih ulaznih podataka koje funkcija očekuje
- opis svih izlaznih podataka koje funkcija generiše
- redosled opcija koje treba aktivirati tokom izvršavanja funkcije
- prikaz ekrana koje korisnik dobija nakon svake aktivirane opcije
- detaljno objašnjenje svakog ekrana koje uključuje njegovu namenu, svrhu svakog podatka na njemu, moguće izbore na ekranu i njihove uloge

Funkcije se u uputstvu obično tematski grupišu. Na primer, najpre se opisuju sve statističke, a zatim sve grafičke funkcije.

Važan aspekt koji treba da bude uključen u uputstvo za korisnika je deo koji se odnosi na ukazivanje na greške do kojih može da dođe u radu sistema i metode za njihovo prevazilaženje.

Uputstvo za korisnika je veoma važan dokument, jer od njegove razumljivosti i sistematičnosti u velikoj meri zavisi kako će se korisnik osećati dok upotrebljava sistem. Ako korisnik teško pronalazi potrebne informacije u priručniku, počće sâm da traži rešenja za svoje probleme, što će sigurno imati za posledicu smanjenje efikasnosti njegovog rada. Zato se uputstvo mora pisati vrlo pažljivo, uz korišćenje raznih tehnika za povećanje čitljivosti i lakši pristup informacijama. Korisne tehnike su: odgovarajuća numeracija, primena različitih boja po temama, korišćenje rečnika, tabela, dijagrama, unakrsno referenciranje i sl. Na primer, ako treba objasniti funkcije nekih tastera, bolje je tastere prikazati slikom sa pridodatim

tekstualnim objašnjenjima i strelicama koje uparuju objašnjenje i taster, nego samo tekstualno opisati raspored tastera i njihove funkcije.

Uputstvo za operatera je tehnički dokument namenjen onima koji treba da pruže podršku radu sistema. Format ovog dokumenta je sličan uputstvu za korisnika, ali mu je sadržaj drugačiji. U uputstvu za operatera se ne opisuju funkcije sistema, već se navode:

- konfiguracija sistema (kako hardverska, tako i softverska)
- postupci dodeljivanja prava pristupa pojedinim grupama korisnika
- performanse sistema i načini njihovog podešavanja
- postupci arhiviranja podataka od interesa
- procedure priključivanja i uklanjanja perifernih uređaja iz sistema i dr.

Uputstvo za operatera počinje isto kao i uputstvo za korisnika. I ono sadrži naslovnu i stranu sa informacijama o autorskim pravima, kao i predgovor i sadržaj. Nakon sadržaja, u uputstvu za operatera daje se opšti pregled sistema sa stanovišta podrške, a zatim i detaljan opis funkcija za podršku. Praksa pokazuje da je dobro da donekle postoji preklapanje uputstava za operatera i korisnika, jer to može da pomogne operateru da razume da li greška koju je prijavio korisnik može da bude otklonjena primenom neke funkcije za podršku, ili je neophodno da o njoj obavesti osoblje zaduženo za održavanje sistema.

Uputstvo za instalaciju je tehnički dokument u kome je detaljno, korak po korak, opisan postupak instalacije softvera. U njemu se navode i uslovi neophodni za regularan rad softvera, uključujući hardversku i softversku platformu, parametre sredine, osobine okruženja, itd.

Osim navedenih dokumenata, korisniku se može isporučiti i *opšti vodič kroz sistem* u kome se opisuje šta radi sistem, bez detaljnog opisa pojedinačnih funkcija. Na osnovu ovog vodiča, kupac treba da bude u stanju da proceni da li sistem u potpunosti odgovara njegovim potrebama. U vodiču se može koristiti unakrsno referenciranje navođenjem referenci na strane korisničkog uputstva na kojima se može pronaći više informacija o nekoj funkciji.

Uputstvo za programere je dokument u kome su detaljno opisane programske komponente i njihov odnos prema funkcijama sistema. Ono pomaže programeru da lakše pronađe deo programskog kôda koji implementira neku funkciju. Pronalaženje funkcije je česta potreba, bilo zbog pojave nekog otkaza ili izmene i unapređenja postojeće funkcije. Uputstvo za programere, osim funkcija sistema,

sadrži i opise funkcija za podršku, što olakšava onima koji održavaju sistem da pronađu izvor problema.

Mnogi korisnici, umesto čitanja dokumenata u papirnoj formi, više vole da sistem upoznaju kroz praktično izvršavanje njegovih funkcija. Za njih se mogu razviti automatizovana uputstva za učenje i pregled sistema. U ovim uputstvima se kombinovanjem teksta i akcija prikazuju funkcije sistema na ilustrativan način, korak po korak.

8.2 Održavanje sistema

Održavanje sistema je aktivnost koja počinje nakon isporuke sistema i traje sve dok je sistem u upotrebi. Potreba za održavanjem postoji zato što sistem stalno evoluira. Pod održavanjem se podrazumevaju sve aktivnosti vezane za izmene sistema dok je on u fazi eksploatacije.

Održavanje softverskog sistema se razlikuje od održavanja uređaja ili hardverskih rešenja. Za razliku od njih, softver nije podložan habanju tokom vremena, pa stoga ne zahteva periodično održavanje.

Ciljevi održavanja softvera su sledeći:

- otklanjanje grešaka koje se pojavljuju nakon isporuke softvera
- izvođenje izmena koje su neophodne zbog promena u hardveru, softveru ili u vezama sa okruženjem
- predlaganje izmena u slučaju kada korisnici imaju problema u korišćenju softvera
- praćenje rada i predviđanje potencijalnih problema u funkcionisanju sistema
- evidentiranje izmena u poslovanju koje zahtevaju promene u softveru

Održavanje sistema izvodi tim za održavanje koji obično nije isti kao tim koji je razvijao sistem. Ako razvojni tim zna da će neko drugi održavati njihov sistem, onda članovi razvojnog tima mnogo više pažnje posvećuju standardima programiranja i izradi dokumentacije. Ponekad se u tim za održavanje uključi i neko iz razvojnog tima, što je dobro, jer smanjuje moguće nesuglasice oko razumevanja načina rada sistema. Tim za održavanje je objektivniji od razvojnog tima u sagledavanju onoga šta sistem zaista radi u odnosu na ono što bi trebalo da radi.

Održavanje sistema zahteva intenzivnu komunikaciju tima za održavanje sa korisnicima, kupcima, operaterima i dr. Tim obično dobija od korisnika informaciju o nastalom problemu. Članovi tima najpre pokušavaju da shvate problem iskazan jezikom korisnika, a zatim da ga transformišu u zahtev za izmenom. Zahtev obuhvata opis sadašnjeg načina rada, opis načina na koji korisnik želi da sistem radi i opis izmene koju treba izvršiti da bi se sa sadašnjeg prešlo na željeni načina rada. Zatim se pristupa izvršavanju izmene kroz modifikovanje odgovarajućeg programskog kôda. Ukoliko je to potrebno, na kraju se može sprovesti i kraća obuka korisnika koja se odnosi na uvedene izmene.

Aktivnosti koje spadaju u delokrug tima za održavanje su vrlo raznovrsne. One obuhvataju:

- detaljno upoznavanje sa sistemom kako bi se u potpunosti razumeo način njegovog rada
- davanje informacija o radu sistema
- upoznavanje sa dokumentacijom da bi se u njoj brzo i lako pronašla potrebna informacija
- ažuriranje dokumentacije prema potrebama
- pronalaženje uzroka otkaza i njihovo otklanjanje
- prepoznavanje potencijalnih problema u sistemu i njihovo predupređivanje
- izvođenje potrebnih izmena u sistemu (u dizajnu, kôdu, testovima)
- oslobađanje sistema od komponenata koje više nisu u upotrebi

8.2.1 Vrste održavanja

Proces održavanja se bavi različitim aspektima evolucije sistema. On obuhvata održavanje svakodnevnih funkcija sistema, održavanje uvedenih izmena, rad na poboljšanju postojećih funkcija i spečavanje degradacije sistema po pitanju performansi na neprihvatljiv nivo. U tom smislu su se izdvojile različite vrste održavanja koje će biti opisane u nastavku.

Korektivno održavanje se bavi kontrolom svakodnevnog rada sistema i otklanjanjem problema nastalih zbog pojave grešaka u softveru. Tim za korektivno održavanje registruje izvor problema i koriguje grešku predlažući potrebne izmene u zahtevima, dizajnu, implementaciji ili testovima, zavisno od mesta nastanka greške. Sprovedena ispravka često ne predstavlja najbolje rešenje i privremenog je karaktera. Njen cilj je da sistem nastavi sa normalnim radom. Trajno rešenje

problema može da zahteva znatno više vremena, posebno ako se u okviru njega rešavaju i opštiji problemi u dizajnu ili programskom kôdu. Na primer, ako se neki trošak računa po algoritmu koji nije implementiran u kôdu, a potreban je za dalji rad sistema, može se korisniku dati mogućnost da ga ručno unese. Zatim, tim za održavanje implementira pomenuti algoritam u okviru kritičnog dela kôda kako bi sistem ubuduće ispravno radio bez intervencije korisnika.

Adaptivno održavanje se primenjuje u situaciji kada izmena u jednom delu sistema zahteva izmene u drugim delovima sistema (sekundarne izmene) da bi sistem ispravno radio. Implementacija sekundarnih izmena predstavlja adaptivno održavanje. Na primer, pretpostavimo da dati softver komunicira sa drugim softverom koji treba da bude zamenjen svojom novom verzijom. Ukoliko nova verzija ima promenjen broj ili značenje parametara u interfejsu, ova promena se mora reflektovati i na dati sistem. Adaptivne izmene koje tada treba izvršiti ne odnose se na ispravljanje grešaka, već na dalju evoluciju sistema i njegov napredak.

Adaptivno održavanje je uglavnom potrebno nakon izmena u hardveru, softveru ili radnom okruženju. Na primer, ako se sistem planiran da radi u stabilnom okruženju premesti u novi prostor sa izraženim vibracijama ili bukom, potrebno ga je prilagoditi novim uslovima.

Održavanje u cilju unapređenja sistema podrazumeva sprovođenje izmena radi poboljšanja nekog dela sistema. Izmene ne moraju da budu posledica grešaka, već se uvode na predlog tima za održavanje koji stalno preispituje dizajn sistema, programski kôd, primenjene testove i dokumentaciju tražeći moguća poboljšanja. U ove izmene spadaju, na primer, promene u testovima kako bi oni bili sveobuhvatniji, ili promene u dokumentaciji radi pojašnjenja onih delova teksta koji nisu dovoljno dobro objašnjeni. Izmene mogu da budu i u kôdu ili dizajnu. Na primer, tim za održavanje može da uoči da zbog priključivanja nekog uređaja i promene načina na koji on šalje podatke sistemu (paralelno ili serijski), treba promeniti strukturu u kojoj se čuvaju ti podaci.

Preventivno održavanje predstavlja modifikovanje softvera u cilju detekcije i korekcije prikrivenih grešaka pre nego što se one ispolje. To znači da je cilj preventivnog održavanja predviđanje mogućih problema i uvođenje izmena u softver kako bi se otkazi preduhitрили. U okviru preventivnog održavanja, može se, na primer, poboljšati mehanizam upravljanja greškama kako bi što veći broj grešaka bio identifikovan i opslužen na odgovarajući način. To se postiže dodavanjem novih kôdova grešaka u iskaz koji analizira greške (*case* ili *catch*).

Svaka vrsta održavanja zahteva određeno vreme. Rezultati istraživanja sprovedenih u cilju utvrđivanja koja vrsta održavanja najduže traje, pokazali su da

se najveći rad ulaže u održavanje u cilju unapređenja sistema (oko 50% ukupnog vremena posvećenog održavanju), a zatim u adaptivno održavanje (oko 25%).

8.2.2 Problemi održavanja

Održavanje softverskog sistema je težak i složen proces iz sledećih razloga:

- sistem je već u eksploataciji
- mogu se javiti personalni i organizacioni problemi
- mogu se javiti tehnički problemi

Sistem koji se održava je već u upotrebi. To znači da svako njegovo modifikovanje mora da se uskladi sa potrebama korisnika. Neki sistemi dopuštaju da izvesno vreme ne budu aktivni, jer time ne ugrožavaju poslovanje (na primer, sistem za obračun plata). Međutim, ima sistema čija priroda zahteva permanentnu dostupnost (na primer, sistem za dovod kiseonika pacijentu). U tim slučajevima, tim za održavanje mora pronaći neko alternativno rešenje kojim ne bi ugrozio korisnika, a mogao bi da sprovede predviđene izmene u softveru. Tada se obično koristi rezervni sistem, a zatim se istestirani deo kôda prebacuje u sistem u upotrebi.

U procesu održavanja, često su prisutni problemi personalne i organizacione prirode. Problem razumevanja je jedan od najizraženijih. Jedna analiza je pokazala da se 47% rada na održavanju softvera troši na razumevanje sistema. Dobro shvatanje sistema je preduslov za efikasno sprovođenje izmena. I najmanja izmena (na primer, jednog reda kôda) može da zahteva izvođenje na desetine testova kako bi se proverio njen uticaj na druge delove sistema.

Poseban problem predstavlja nedovoljno razumevanje načina funkcionisanja sistema i nedostatak potrebnih veština od strane korisnika. Dešava se da zbog ovoga korisnici timu daju nepotpune ili čak pogrešne podatke u vezi sa nekom greškom ili otkazom. Ovi problemi se donekle mogu rešiti kvalitetnom obukom i dokumentacijom.

Problemi mogu da nastanu i ukoliko se javi nesklad između prioriteta rukovodstva i korisnika. Dešava se da rukovodstvo, koje je usredsređeno prvenstveno na poslovanje, zahteva od tima za održavanje česte izmene sistema radi dopune funkcionalnosti. Pri tome, rukovodstvo ne uviđa da je sistem već toliko izmenjen da bi bilo bolje zameniti ga novim sistemom koji bi korisnicima omogućio ugodniji i efikasniji rad.

Veliki broj proizvođača softvera i velika konkurencija na tržištu zahtevaju sve brži razvoj softvera. Za proizvođača je važno da njegov proizvod što pre izađe na

tržište, tako da prelazi preko činjenice da takav softver nije dovoljno istestiran. Ovo pravi mnogo problema timu za održavanje, jer se takav proizvod teško razume i modifikuje.

Veliki značaj u održavanju ima moral tima za održavanje. Studije pokazuju da 11% problema u održavanju potiče od niskog morala, a samim tim i niske produktivnosti članova tima. Glavni razlog niskog morala je taj što postoje predrasude da je posao održavanja drugorazredni u odnosu na posao razvoja. Pojedinci smatraju da je potrebno više znanja, kreativnosti i veštine da se sistem isprojektuje i implementira, nego da se održava. Međutim, to nije tako. Programeri iz održavanja se bave mnogim problemima kojih projektanti nisu ni svesni. Na primer, oni moraju da budu ne samo obučeni za pisanje kôda, već i za rad sa korisnicima, za predviđanje potencijalnih problema, preprojektovanje sistema, izvođenje izmena, itd. Za praćenje problema od njegovog izvora, preko predloga izmena, do njihovog sprovođenja u praksi potrebni su velika veština, istrajnost i sistematičnost. Problem morala se rešava rotiranjem programera iz razvojnog tima u tim za održavanje i obrnuto. Međutim, tada se dešava da jedan programer radi na više projekata i poslova, što ga sprečava da se u potpunosti posveti jednom problemu i da ga efikasno reši.

Posebnu grupu problema pri održavanju čine tehnički problemi. Oni nastaju zbog načina projektovanja i implementacije sistema, kao i zbog izabranih hardverskih i softverskih platformi korišćenih u realizaciji.

Ukoliko softver nije dobro i fleksibilno isprojektovan, to stvara velike probleme timu za održavanje. Tim ne može lako da shvati strukturu i funkcionisanje sistema, pa mu je teško da proceni da li predložena izmena može da se implementira i na šta bi sve to uticalo. Primer lošeg projektovanja je bio ograničavanje polja za predstavljanje godine na dve cifre, što je prouzrokovalo probleme u mnogim aplikacijama na prelasku između dva veka.

Problemi mogu da nastanu i kada je softver realizovan na hardverskoj/softverskoj platformi koja je nepouzdana i podložna otkazima. Osim toga, ukoliko softver dodeljuje ograničene resurse korisnicima, to može da pravi probleme prilikom nekih izmena. Na primer, ako je korisniku dodeljena neka količina memorijskog prostora, a izmena je takva da on u taj prostor treba da smesti veću količinu podataka, tim za održavanje je taj koji mora da reši problem.

Tim za održavanje može da ima problema i sa testiranjem izmena. Naime, može se desiti da tim nema na raspolaganju odgovarajuće ulazne podatke da bi izmena mogla da bude istestirana na pravi način. Na primer, ako se izmena odnosi na uvođenje analize neke snimljene slike, a kamere još nisu instalirane u okruženju, onda softver nema realne ulazne podatke, već ih mora generisati simulacijom.

8.2.3 Troškovi održavanja

Održavanje je poslednja faza u životnom veku softvera. Ono obezbeđuje da softver u svakom trenutku bude ažuran, tj. usklađen sa radnim okruženjem. Nakon isporuke proizvoda, sve promene u okruženju, kao i u korisničkim zahtevima, kroz održavanje se implementiraju u softveru, što vodi njegovom modifikovanju i stalnom unapređivanju.

Proces održavanja u velikoj meri zavisi od prethodnih faza u razvoju softvera, i zato se tokom celog razvoja softverskog proizvoda mora voditi računa i o mogućnostima održavanja. U fazi projektovanja, strukturu sistema treba planirati tako da može lako da se izmeni. U fazi implementacije, programski kôd se piše tako da bude razumljiv i čitljiv i da može lako da se modifikuje. Ako su ranije faze dobro urađene, održavanje će biti lakše i efikasnije.

Efikasnost održavanja je važan cilj kome se uvek teži, zato što je u direktnoj vezi sa troškovima održavanja. Troškovi održavanja uključuju potrošeno vreme, uloženi rad i novac. Nekad se dešava da troškovi održavanja budu tako veliki da se postavlja pitanje da li je bolje napraviti novi sistem ili nastaviti sa održavanjem starog. Van Vliet je 2000.godine uradio studiju koja je pokazala da najmanje 50% ukupnih troškova u životnom veku projekta odlazi na troškove održavanja. Procena je da će se troškovi održavanja u budućnosti povećati i do 80% ukupnih troškova. Troškovi se razlikuju po vrstama održavanja. Najveći troškovi su kod održavanja u cilju unapređenja sistema (oko 50% troškova održavanja), zatim adaptivnog (oko 25 %) i korektivnog održavanja (oko 20%), dok na preventivno održavanje odlazi samo oko 5%.

Karakteristike softverskog proizvoda koje utiču na troškove održavanja su:

- *veličina sistema*. Veći sistemi se teže održavaju od manjih, zato što je potrebno znatno više vremena da se upozna njihov rad u toj meri da bi se mogle izvesti potrebne modifikacije. Veći sistemi sadrže veliki broj raznovrsnih funkcija, pa je i verovatnoća pojave grešaka veća.
- *životni vek softvera*. Stariji sistemi zahtevaju više napora pri održavanju od novijih, zato što vremenom sistem raste, postaje lošije organizovan zbog brojnih promena i manje razumljiv (posebno ako je došlo do promena u timu za održavanje). Ovo je manje uočljivo kod softvera čiji je životni vek kraći nego kod dugoročnih projekata.
- *vrsta aplikacije*. Neke vrste aplikacija su, zbog svoje prirode, teške za održavanje. Modifikacija vremenski kritičnih aplikacija je složena, jer zahteva sinhronizovanje svih komponentata u sistemu koje su u vezi sa

izmenjenom komponentom, bilo direktno ili posredno. Distribuirani sistemi se, takođe, teško održavaju zbog svoje razuđene strukture.

- *programski kôd*. Kraći kôd se lakše održava. Na primer, manji su troškovi ako se menja 10% od 200 linija kôda u nekom modulu, nego ako se menja 20% od 100 linija kôda u drugom modulu, iako se u oba slučaja radi o 20 linija. Programski jezik na kome je sistem implementiran, takođe, utiče na troškove održavanja. Programi pisani na jezicima višeg nivoa se lakše održavaju od programa pisanim na jezicima nižeg nivoa.
- *strukturiranost sistema*. Troškovi održavanja su manji ako je sistem dobro strukturiran. To znači da su njegove komponente u velikoj meri nezavisne, ali kompaktne, i sa dobro definisanim vezama prema ostatku sistema. U tom slučaju, izmena jedne komponente se može lako kontrolisati, i uglavnom ne zahteva promene u drugim komponentama.

Martin i McClure su 1983.godine izdvojili faktore koji utiču na smanjenje troškova održavanja:

- *primena strukturiranog projektovanja*. Precizno definisana i osmišljena struktura projekta omogućava bolje razumevanje sistema i lako praćenje izmena i njihovih posledica.
- *primena savremenih softverskih tehnika*. Pogodnosti koje nude savremene softverske tehnike treba u što većoj meri uključivati u sistem kako bi njegov rad bio fleksibilniji i efikasniji.
- *korišćenje automatskih alata*. Automatski alati, s jedne strane, ubrzavaju rad, a sa druge stane, smanjuju mogućnost greške usled ljudskog faktora. Zato ih treba koristiti u svim fazama razvoja gde je to pogodno (pri implementaciji, testiranju, itd.)
- *iskusan tim za održavanje*. Troškovi održavanja su manji ako je tim za održavanje dobro organizovan i ako njegovi članovi već imaju iskustva u sličnim poslovima.

9 Kvalitet softvera

U razvoju softvera primenjuju se različiti postupci, tehnike, metode i alati da bi se postigao postavljeni cilj, tj. dobilo softversko rešenje zadatog problema. Donošenje brojnih odluka predstavlja sastavni deo procesa razvoja. Odlučivanje je prisutno pri izboru tehnika razvoja, izboru alata, resursa, itd. Pri donošenju odluke, uvek se može postaviti pitanje da li je jedna tehnika bolja od druge, da li je efikasniji jedan ili drugi metod, i sl.

Nakon završetka softverskog proizvoda, potrebno je proceniti njegov kvalitet. On se procenjuje sa različitih aspekata, koji uključuju ne samo proizvod, već i proces njegovog razvoja (ispravnost donetih odluka), njegovu primenu, korišćene resurse, i dr. Procenom kvaliteta se utvrđuje ispunjenost postavljenih funkcionalnih zahteva, stepen produktivnosti, performanse sistema i druge nefunkcionalne osobine.

U narednim poglavljima, opisani su mogući pogledi na kvalitet softvera i odabrani modeli kvaliteta.

9.1 Pogledi na kvalitet softvera

Procena kvaliteta softverskog proizvoda zavisi od konteksta u kome se proizvod posmatra. Isti nedostatak se ne može tretirati na isti način u svakom softveru. Na primer, nedostatak u softveru za obradu teksta nema isti značaj, a ni posledice, kao kada se pojavi u softveru namenjenom zaštiti podataka, gde može da ugrozi ceo sistem.

U zavisnosti od gledišta, kvalitet softvera se može posmatrati na tri načina: kroz kvalitet proizvoda, kvalitet postupka izrade proizvoda i kvalitet sa stanovišta poslovnog okruženja u kome se softver koristi.

9.1.1 Kvalitet proizvoda

Procena kvaliteta proizvoda podrazumeva utvrđivanje da li je softver kvalitetan ili nije na osnovu njegovih karakteristika. Međutim, ako bi se isti softver dao većem broju osoba da procene njegov kvalitet, one bi pri tome verovatno razmatrale različite karakteristike. To je tako zato što svako posmatra softver sa svog stanovišta.

Za korisnika, softver je kvalitetan ako ima potrebnu funkcionalnost, jednostavnu obuku i lako se koristi. Može se reći da korisnik posmatra softver „spolja“, tj. sa aspekta upotrebe. Kada meri kvalitet softvera, korisnik procenjuje broj otkaza, kao i vrste otkaza. Često otkaze klasifikuje u minorne, glavne i otkaze sa katastrofalnim posledicama. Što je veći broj katastrofalnih i glavnih otkaza, softver je nekvalitetniji.

Softver procenjuju i projektanti i drugi učesnici u razvoju, posebno tim za održavanje koji treba da nastavi sa radom na softveru. Oni obično analiziraju interne karakteristike softverskog proizvoda, tj. posmatraju softver „iznutra“. To podrazumeva procenu nedostataka u zahtevima, u projektu, u programskom kôdu, itd. Učesnici u razvoju mere kvalitet softvera, takođe, kroz njegove nedostatke, ali su ti nedostaci drugačije prirode od nedostataka koje meri korisnik.

Svaki od navedena dva pogleda na softver ima svoj značaj i ne može se zanemariti. Iako i korisnici i projektanti imaju svoje argumente, njihovi zaključci po pitanju kvaliteta softvera mogu se razlikovati. Na primer, ako se neki softver teško uči i koristi, korisnik može da ga proceni kao loš, iako on ima funkcionalnost koja je nova i vredna, što mu svakako predstavlja kvalitet i daje mu prednost u odnosu na konkurentske softvere.

Da bi se postigla jedinstvena i objektivna ocena kvaliteta softverskog proizvoda, neophodno je objediniti spoljašnji pogled korisnika sa unutrašnjim pogledom učesnika u razvoju softvera. Ovaj cilj je ostvaren uvođenjem modela kvaliteta. U poglavlju 9.2 opisani su najčešće primenjivani modeli kvaliteta.

9.1.2 Kvalitet procesa razvoja

Kvalitet procesa razvoja softvera je podjednako važan kao i kvalitet proizvoda. Tokom razvoja, izvršavaju se mnoge aktivnosti koje imaju uticaja na kvalitet finalnog proizvoda. Ako neka od aktivnosti krene u pogrešnom smeru, smanjuje se kvalitet proizvedenog softvera.

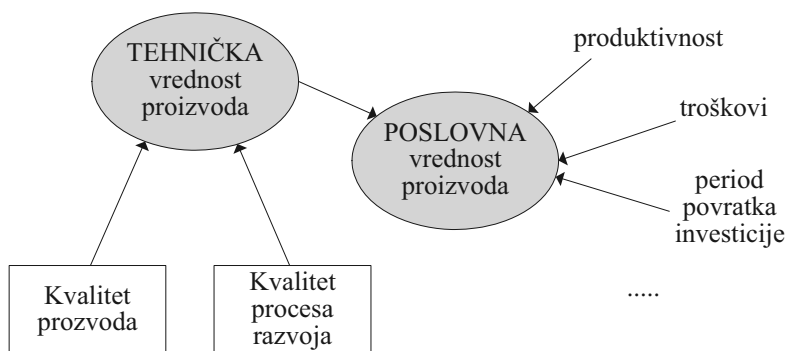
Proces modelovanja razvoja softvera znatno doprinosi boljem kvalitetu softvera. Modelovanjem je omogućena pravovremena analiza sistema, uz

uočavanje i otklanjanje postojećih nedostataka i nalaženje načina za dalja unapređenja sistema. Tokom modelovanja, značajno vreme se troši na analizu nedostataka, što direktno utiče na kvalitet. Ova analiza daje odgovore na pitanja gde i kada će se verovatno pojaviti nedostaci, kako ih pronaći što pre, da li se može ugraditi tolerancija na greške kako one ne bi prešle u otkaz, itd. Analizi mogu biti podvrgnute sve faze u procesu razvoja. Metodi modelovanja opisani su u poglavlju 2.

9.1.3 Kvalitet sa stanovišta okruženja

Svaki softver namenjen je korišćenju u datom poslovnom okruženju. Ukoliko to nije slučaj, smatra se da softver nema smisla. Nakon isporuke, softver ulazi u fazu upotrebe, uz povremeno održavanje. To uvodi još jedan pogled na kvalitet softvera, a to je pogled sa aspekta poslovanja. Za razliku od ocenjivanja kvaliteta proizvoda i procesa razvoja, koji se mere brojem otkaza, vremenom potrebnim za njihovo otklanjanje i dr., kvalitet sa aspekta poslovanja se posmatra u zavisnosti od proizvoda i usluga koje pruža okruženje u kome softver radi.

Imajući ovo u vidu, može se smatrati da softverski proizvod poseduje svoju tehničku i poslovnu vrednost. *Tehničku vrednost* čine karakteristike proizvoda i procesa razvoja. One zavise od načina izrade softvera, tj. od tehničkog aspekta. S druge strane, *poslovna vrednost* predstavlja poslovne efekte koje softver proizvodi u radnom okruženju. To su ekonomske kategorije, kao na primer produktivnost, troškovi, period povratka investicije, itd. Ove dve vrednosti su međusobno zavisne. Povećanje tehničke vrednosti proizvoda svakako utiče na poslovnu vrednost. Odnos tehničke i poslovne vrednosti proizvoda prikazan je na slici 9.1.



Slika 9.1 Tehnička i poslovna vrednost proizvoda

Tehnička vrednost proizvoda je važna za direktne korisnike softvera, jer utiče na njihovo prihvatanje softvera. Za rukovodstvo kompanije, bitnija je poslovna vrednost. Oni kvalitet mere kroz ostvarenu dobit (ostvarenu zbog uvođenja softvera), troškove održavanja, troškove izmena, odnos vremena operativnog rada i zastoja, i dr.

9.2 Modeli kvaliteta

Modeli kvaliteta su nastali zbog potrebe uključivanja različitih aspekata posmatranja u procenu kvaliteta proizvoda. Bez obzira na veliki broj modela kvaliteta opisanih u literaturi, svi modeli podržavaju isti pristup. Svaki model definiše skup karakteristika proizvoda koje smatra važnim. Ove karakteristike potiču kako od spoljašnjeg, tako i od unutrašnjeg pogleda na softver. One su međusobno povezane na odgovarajući način i imaju udela u postupku ocene kvaliteta.

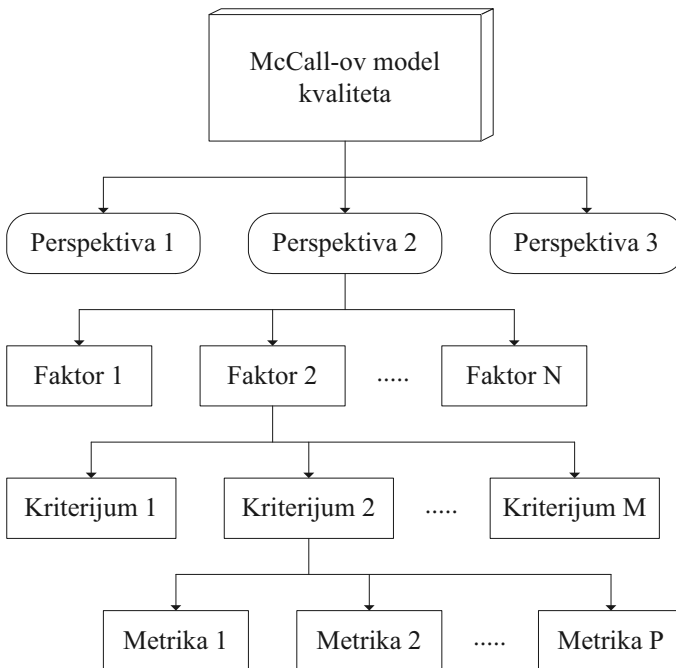
Mnogi modeli kvaliteta su nedorečeni, tj. ne preciziraju u potpunosti postupak procenjivanja kvaliteta proizvoda, već uglavnom daju okvire i smernice koji se mogu koristiti u proceni. Takođe, ne navode se razlozi zašto su pojedine karakteristike uključene u model, a druge ne. Nema objašnjenja ni zašto se neka karakteristika nalazi baš na konkretnom mestu u modelu, a ne na nekom drugom.

U narednim poglavljima opisani su najpoznatiji modeli kvaliteta: Mc Call-ov model, Boehm-ov model, Dromey-ov model i ISO 9126.

9.2.1 McCall-ov model

McCall-ov model je jedan od najpoznatijih i najstarijih modela. Predložio ga je McCall 1977.godine i izvorno je bio namenjen američkoj vojsci. Model predstavlja pokušaj prevazilaženja razlika u shvatanjima korisnika i razvojnog tima po pitanju kvaliteta softvera.

McCall-ov model čini hijerarhija faktora, kriterijuma i metrike kojima se proizvod opisuje, kako sa spoljašnjeg, tako i sa unutrašnjeg aspekta. Faktori predstavljaju različite vrste karakteristika koje opisuju ponašanje sistema. Svakom faktoru pridružen je skup kriterijuma, koji odgovaraju atributima faktora. Svaki kriterijum se može uključiti u jednu ili više metrika. Metrika omogućava merenje kvaliteta i upravljanje njime. Model obuhvata tri perspektive, kao što je prikazano na slici 9.2.



Slika 9.2 Struktura McCall-ovog modela kvaliteta

U modelu postoji 11 faktora koji definišu kvalitet proizvoda sa stanovišta korisnika i 23 kriterijuma koji opisuju kvalitet sa stanovišta onih koji razvijaju sistem. Svaki faktor je povezan sa dva do pet kriterijuma. U zavisnosti od značenja, faktori su raspoređeni u tri perspektive:

- *operativnost proizvoda* (Perspektiva 1 na slici), sadrži faktore koji se odnose na upotrebu proizvoda
- *revizija proizvoda* (Perspektiva 2), sadrži faktore koji opisuju mogućnosti menjanja proizvoda
- *tranzicija proizvoda* (Perspektiva 3), sadrži faktore o prilagodljivosti proizvoda radu u novom okruženju

Perspektiva *operativnost proizvoda* sadrži sledeće faktore:

- *ispravnost*, pokazuje u kojoj meri funkcionalnost sistema odgovara specifikaciji
- *pouzdanost*, ukazuje na učestalost pojave grešaka

- *efikasnost*, odnosi se na način korišćenja resursa i uključuje efikasnost izvršavanja, kao i efikasnost pamćenja podataka
- *integritet*, ukazuje na sposobnost zaštite od neovlašćenog pristupa softveru
- *upotrebljivost*, opisuje lakoću korišćenja softvera

Perspektiva *revizija proizvoda* obuhvata:

- *moгуćnost održavanja*, ukazuje na trud potreban za lociranje i otklanjanje grešaka u programu unutar radnog okruženja
- *fleksibilnost*, pokazuje lakoću izvođenja izmena potrebnih zbog promena u radnom okruženju
- *moгуćnost testiranja*, odnosi se na lakoću testiranja programa

Perspektivu *tranzicija proizvoda* čine faktori:

- *prenosivost*, ukazuje na napor potreban za prenos softvera iz jednog u drugo okruženje
- *moгуćnost ponovnog korišćenja*, opisuje mogućnost ponovnog korišćenja softvera u drugom kontekstu
- *interoperabilnost*, odnosi se na mogućnosti spezanja softvera sa drugim sistemima

U tabeli 9.1 navedeni su kriterijumi za faktore perspektive *operativnost proizvoda* (pod a)) i perspektive *revizija proizvoda* (pod b)).

Perspektiva	Faktor	Kriterijum
operativnost proizvoda	ispravnost	praćenje i razumljivost
		potpunost
		konzistentnost
	pouzdanost	konzistentnost
		preciznost
		tolerancija na greške
	efikasnost	efiksanost izvršavanja
		efiksanost pamćenja
	integritet	kontrola pristupa
		revizija pristupa
	upotrebljivost	operativnost
		obuka
		kommunikacija

a)

Perspektiva	Faktor	Kriterijum
revizija proizvoda	mogućnost održavanja	jednostavnost
		konciznost
		samoopisivost
		modularnost
	fleksibilnost	samoopisivost
		proširivost
		opštost
	mogućnost testiranja	jednostavnost
		instrumentacija
		samoopisivost
		modularnost

b)

Tabela 9.1 Sadržaj McCall-ovog modela kvaliteta

Osnovni cilj McCall-ovog modela je formiranje strukture faktora i kriterijuma koja daje potpunu sliku o kvalitetu softvera i primena metrika na tu strukturu radi procene kvaliteta softvera.

McCall definiše skup metrika u vidu izraza za računanje uticaja pojedinačnih elemenata na kvalitet. Opšti oblik izraza za faktore kvaliteta je:

$$F_q = c_1m_1 + c_2m_2 + \dots + c_nm_n$$

gde je F_q faktor kvaliteta, m_i kriterijumi od kojih faktor zavisi, a c_i koeficijenti uticaja (težinski koeficijenti). Vrednost koeficijenata c_i zavisi od konkretnog softverskog proizvoda. Priroda mnogih kriterijuma je takva da se njihova ispunjenost može proceniti samo subjektivno. Skup metrika se može definisati i u vidu tabele sa označenim zavisnostima između faktora i kriterijuma bitnih za taj faktor. Deo skupa dat je u tabeli 9.2.

Pri merenju kvaliteta softvera koristi se metod odgovaranja sa „da“ ili „ne“ na postavljena pitanja. Na primer, da bi se utvrdio uticaj nekog kriterijuma, postavlja se veći broj pitanja u vezi sa njim. Ako je isti broj odgovora „da“ i „ne“, smatra se da se kriterijum dostiže sa 50%. Metrike se mogu formirati po kriterijumima, faktorima, ili po proizvodu.

K R I T E R I J U M I	F A K T O R I	ispravnost	pouzdanost	efikasnost	integritet	upotrebljivost	održavanje	fleksibilnost	testiranje	prenosivost	pon. korišćenje	interoperabilno
jednostavnost			X				X	X	X			
konzistentnost		X	X				X	X				
opštost								X		X	X	X
modularnost			X				X	X	X	X	X	X
tolerancija na greške			X									

Tabela 9.2 Veza faktora i kriterijuma u metrici

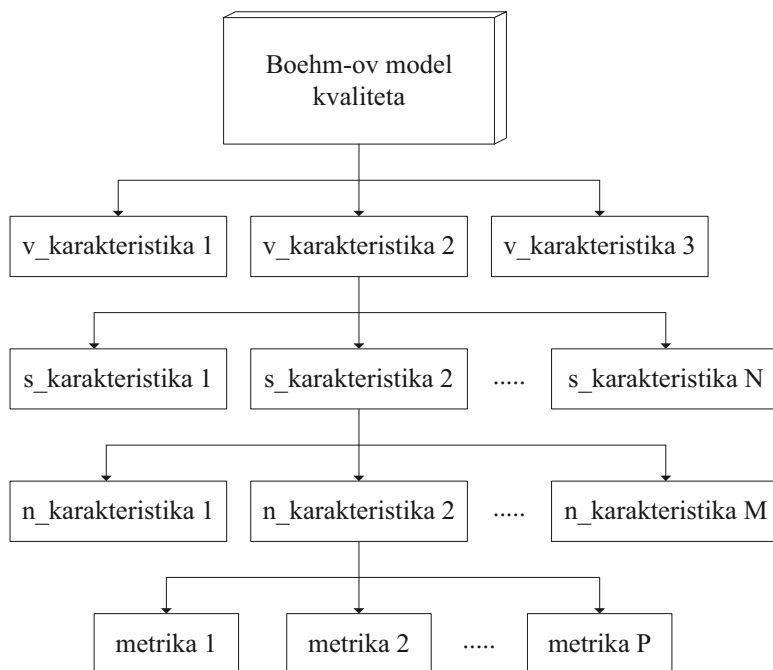
9.2.2 Boehm-ov model

Boehm-ov model je nastao neposredno posle McCall-ovog modela, 1998. godine. I ovaj model opisuje kvalitet proizvoda u vidu hijerarhije karakteristika koje uključuju očekivanja i korisnika i učesnika u razvoju. Međutim, Boehm-ov model ima širi pristup od McCall-ovog, zato što uključuje i troškove i efikasnost održavanja.

Za Boehm-a i ostale autore modela, primarna karakteristika kvaliteta je opšta korisnost. Softver, pre svega, mora da bude koristan. Ako to nije slučaj, onda je njegov razvoj bio gubitak vremena, novca i rada. Sâm model se sastoji od tri nivoa: visokog, srednjeg i niskog nivoa karakteristika. Na dnu hijerarhije se nalaze metrike pridružene karakteristikama najnižeg nivoa, kao što je prikazano na slici 9.3.

Visok nivo karakteristika opisuje osnovne zahteve po pitanju kvaliteta koji moraju da budu zadovoljeni. Ovaj nivo sadrži tri karakteristike:

- *upotrebljivost* (v_karakteristika 1 na slici). Pokazuje koliko efiksano, lako i pouzdano se softver može koristiti (onakav kakav je trenutno).
- *moгуćnost održavanja* (v_karakteristika 2). Ukazuje na mogućnost lakog razumevanja, modifikovanja i ponovnog testiranja softvera.
- *prenosivost* (v_karakteristika 3). Odnosi se na mogućnost korišćenja softvera u slučaju izmenjenog okruženja.



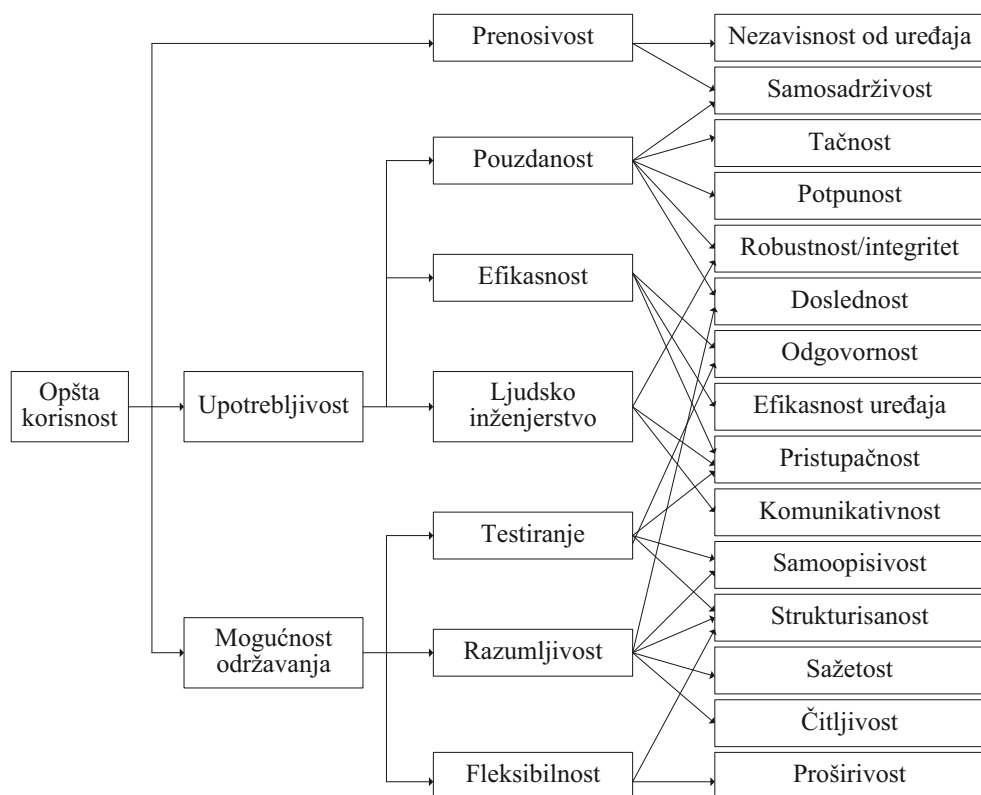
Slika 9.3 Struktura Boehm-ovog modela kvaliteta

Na srednjem nivou se nalazi sedam karakteristika koje opisuju šta se po pitanju kvaliteta očekuje od softverskog sistema. To su:

- *prenosivost*. Softver može dobro da radi u računarskom okruženju drugačijem od postojećeg.
- *pouzdanost*. Softver na zadovoljavajući način izvršava funkcionalnost koja se od njega očekuje.
- *efikasnost*. Sistem efikasno koristi resurse.
- *upotrebljivost*. Softver je pouzdan, efikasan i pogodan za korišćenje.
- *moгуćnost testiranja*. Uspostavljeni su kriterijumi validacije i evaluacija performansi.
- *razumljivost*. Svrha softvera je jasno naznačena.
- *fleksibilnost*. Softver omogućava lako uključivanje izmena u sistem.

Niski nivo obuhvata petnaest karakteristika koje su povezane sa karakteristikama srednjeg i visokog nivoa na način prikazan na slici 9.4.

Kao što se vidi, za razliku od McCall-ovog modela, Boehm-ov model predstavlja hijerarhiju koja na najvišem nivou ima karakteristike bitne za krajnjeg korisnika, a na dnu karakteristike tehničkog tipa. Karakteristike na najnižem nivou predstavljaju osnovu za definisanje metrika kvaliteta. Pod metrikom, Boehm podrazumeva meru obima ili stepena do koga softverski proizvod poseduje neku karakteristiku kvaliteta. Merljive karakteristike se odnose na visoko tehničke detalje kvaliteta koje teško mogu da razumeju ne-tehnički učesnici u projektu.



Slika 9.4 Sadržaj Boehm-ovog modela kvaliteta

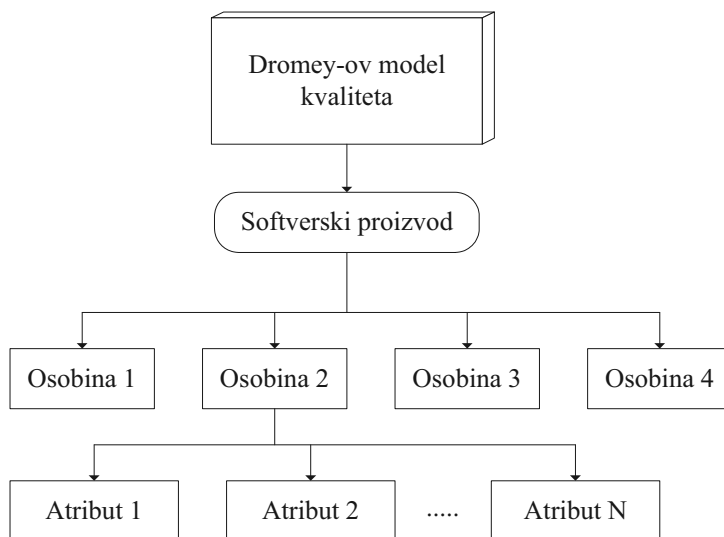
9.2.3 Dromey-ov model

Dromey-ov model kvaliteta potiče iz 1995.godine. On se zasniva na proizvodu. Uočeno je da postupak procene kvaliteta treba da se razlikuje od

proizvoda do proizvoda. Stoga je potrebno razviti više ideja, kako bi se one mogle primeniti na različite sisteme.

Model kvaliteta proizvoda se sastoji od četiri njegove osobine. Za svaku osobinu, definisan je određen broj atributa kvaliteta. Opšta struktura Dromey-ovog modela data na je slici 9.5.

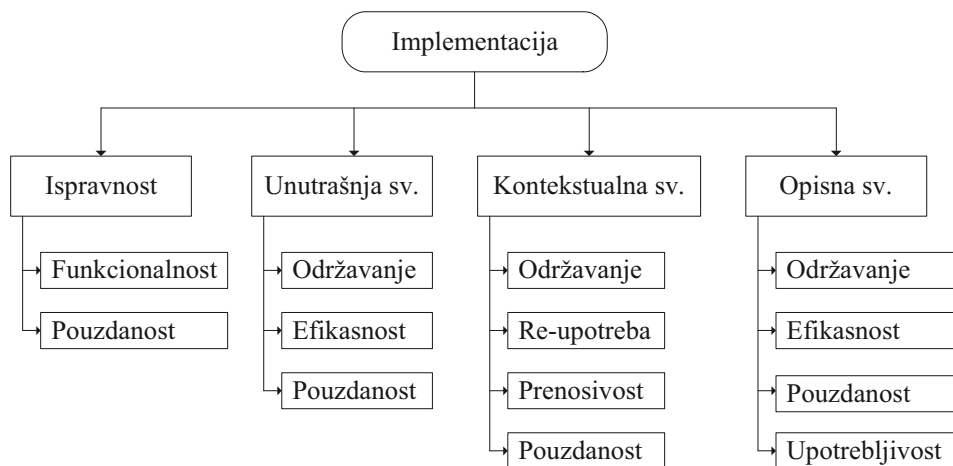
Dromey je predložio generičku tehniku za izgradnju modela kvaliteta. Po njemu, kvalitet proizvoda zavisi od izbora komponenata od kojih je proizvod načinjen, osobina tih komponenata i njihove kompozicije. Po pitanju komponenata, Dromey je razvio različite modele koji se mogu koristiti za evaluaciju kvaliteta zahteva, dizajna i sâmog softvera. Ideja je bila da se tokom celog životnog ciklusa proizvode elementi kvaliteta, tako da se na kraju dobije finalni proizvod koji ispoljava attribute visokog kvaliteta.



Slika 9.5 Struktura Dromey-ovog modela kvaliteta

Na slici 9.6 prikazan je sadržaj Dromey-ovog modela.

Kao što se vidi, izdvojene su četiri osobine proizvoda: ispravnost i unutrašnja, kontekstualna i opisna svojstva. Osobine su povezane sa atributima kvaliteta visokog nivoa, tj. atributima sa najvećim prioritetom (prioriteti atributa se razlikuju u različitim projektima).



Slika 9.6 Sadržaj Dromey-ovog modela kvaliteta

Primena modela obuhvata sledećih pet koraka:

1. Izabrati skup atributa kvaliteta visokog nivoa.
2. Identifikovati komponente sistema.
3. Identifikovati najznačajnija svojstva koja doprinose kvalitetu svake komponente.
4. Odrediti kako svako svojstvo utiče na attribute kvaliteta.
5. Uraditi evaluaciju modela i identifikovati njegove nedostatke.

9.2.4 Model ISO 9126

Model ISO (International Standardization Organization) 9126 iz 1991.godine nastao je kao rezultat pokušaja da se u softverskom inženjerstvu uspostavi konsenzus po pitanju terminologije vezane za procenu kvaliteta proizvoda. Ideja je bila da ovaj model postane svetski standard za procenu kvaliteta softvera. Kasnije su se pojavile brojne proširene verzije ovog standarda.

ISO 9126 model, osim modela kvaliteta, definiše i niz uputstava za merenje karakteristika kvaliteta koje se pojavljuju u modelu.

ISO model ukazuje na tri aspekta kvaliteta softvera:

- *kvalitet sa aspekta upotrebe*. Predstavlja korisnički pogled na kvalitet softvera kada je softver u upotrebi u datom okruženju i datom kontekstu.

Njime se meri obim u kome korisnici mogu da ostvare svoje ciljeve u datom okruženju, ne vodeći računa o osobinama samog softvera.

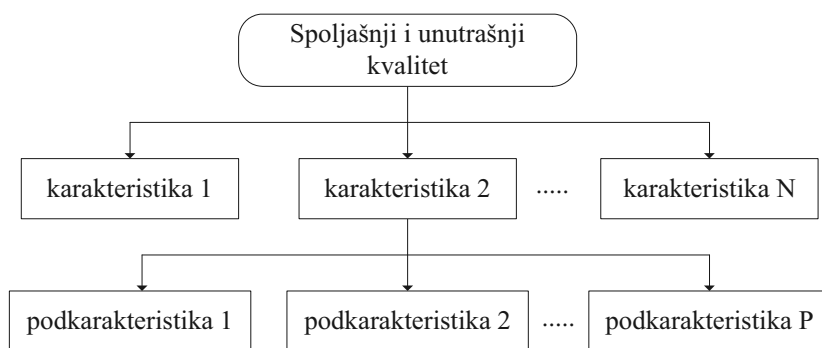
- *spoljašnji kvalitet*. Određen je ukupnim spoljašnjim karakteristikama softvera koji se izvršava. Obično se meri tokom testiranja u simuliranom okruženju sa simuliranim podacima primenom spoljašnjih metrika.
- *unutrašnji kvalitet*. Određen je ukupnim unutrašnjim karakteristikama softvera. Kvalitet softvera se može poboljšati tokom kodiranja, pregledanja i testiranja.

Navedeni aspekti su međusobno povezani. Atributi bitni za unutrašnji kvalitet utiču na attribute spoljašnjeg kvaliteta, a oni, dalje, utiču na attribute kvaliteta sa aspekta upotrebe.

ISO 9126 se može smatrati dvodelnim modelom kvaliteta, koga čine sledeći delovi:

- *spoljašnji i unutrašnji model kvaliteta*
- *model kvaliteta upotrebe*

Spoljašnji i unutrašnji model kvaliteta se zasniva na McCall-ovom i Boehm-ovom modelu. To je model koga čine tri nivoa. Prvi nivo sadrži 6 karakteristika kvaliteta, drugi 27 podkarakteristika kvaliteta, a treći mere kvaliteta. Model je prikazan na slici 9.7.



Slika 9.7 ISO 9126 model za spoljašnji i unutrašnji kvalitet

U okviru standarda, predloženo je više od 100 mera spoljašnjih i unutrašnjih karakteristika, mada taj skup nije potpun, već se mogu koristiti i druge mere. Ove mere su opisane u više dokumenata.

U nastavku su opisan sadržaj modela, tj. značenja svih njegovih karakteristika i podkarakteristika:

- *funkcionalnost* – sposobnost softverskog proizvoda da obezbedi potrebne funkcije pri radu u zadatim uslovima. Sadrži sledeće podkarakteristike:
 - *pogodnost* – sposobnost softvera da obezbedi odgovarajući skup funkcija za specificirane zadatke i korisničke ciljeve
 - *preciznost* – sposobnost softverskog proizvoda da obezbedi prave i dogovorene rezultate ili efekte sa potrebnim stepenom preciznosti
 - *sigurnost* – sposobnost softvera da zaštiti informacije i podatke tako da neautorizovane osobe ili sistemi ne mogu da im pristupaju, a autorizovani mogu
 - *interoperabilnost* – sposobnost softvera da ostvari interakciju sa drugim sistemima
 - *funkcionalna usaglašenost* – sposobnost softverskog proizvoda da poštuje standarde, dogovore ili zakonske regulative povezane sa funkcionalnošću
- *pouzdanost* – sposobnost softverskog proizvoda da postigne zadati nivo performansi kada se koristi u zadatim uslovima. Sadrži sledeće podkarakteristike:
 - *zrelost* – sposobnost softvera da prevaziđe nedostatak nastao kao rezultat greške u softveru
 - *tolerancija na greške* – sposobnost softvera da održi potreban nivo performansi u slučaju pojave softverskih grešaka
 - *moгуćnost oporavka* – sposobnost softvera da ponovo uspostavi potreban nivo performansi i oporavi podatke direktno uključene u nastanak nekog problema
 - *usaglašenost pouzdanosti* – sposobnost softverskog proizvoda da poštuje standarde, dogovore ili zakonske regulative povezane sa pouzdanošću
- *upotrebljivost* – sposobnost softverskog proizvoda da se lako usvaja, razume, uči i koristi kada se primenjuje pod zadatim uslovima. Sadrži sledeće podkarakteristike:
 - *razumljivost* – sposobnost softvera da korisniku omogući da lako shvati da li je sistem odgovarajući, kako se koristi i pod kojim uslovima
 - *moгуćnost učenja* – sposobnost softvera da korisniku pruži mogućnost učenja aplikacije

-
- *operativnost* – sposobnost softvera da omogući korisniku da radi i kontroliše sistem
 - *atraktivnost* – sposobnost softvera da bude privlačan za korisnika
 - *upotrebna usaglašenost* – sposobnost softverskog proizvoda da poštuje standarde, dogovore, uputstva ili zakonske regulative povezane sa upotrebljivošću
 - *efikasnost* – sposobnost softverskog proizvoda da pruži odgovarajuće performanse, u skladu sa korišćenim resursima, u zadatim uslovima rada. Sadrži sledeće podkarakteristike:
 - *ponašanje u vremenu* – sposobnost softvera da pri izvršavanju funkcija proizvede željeni odziv u zadatom vremenu i uz predviđene protoke (pod specificiranim radnim uslovima)
 - *ponašanje u vezi sa resursima* – sposobnost softvera da koristi odgovarajuće količine i vrste resursa pri izvršavanju svojih funkcija
 - *usaglašenost efikasnosti* – sposobnost softverskog proizvoda da poštuje standarde i konvencije povezane sa efikasnošću
 - *mogućnost održavanja* – sposobnost softverskog proizvoda da bude modifikovan. Modifikacije uključuju moguće korekcije, poboljšanja ili prilagođenja u skladu sa promenama u okruženju, promenama u zahtevima ili funkcionalnim specifikacijama. Sadrži sledeće podkarakteristike:
 - *mogućnost analize* – sposobnost softvera da dijagnostikuje nedostatke i njihove uzroke, kao i da identifikuje delove koje treba modifikovati
 - *promenljivost* – sposobnost softvera da se u njemu implementira potrebna modifikacija
 - *stabilnost* – sposobnost softvera da izbegne neočekivane efekte zbog izmena u softveru
 - *mogućnost testiranja* – sposobnost softvera da omogući validaciju nakon modifikacije softvera
 - *usaglašenost održavanja* – sposobnost softverskog proizvoda da poštuje standarde i konvencije povezane sa održavanjem
 - *prenosivnost* – sposobnost softverskog proizvoda da se prenese iz jednog okruženja u drugo. Sadrži sledeće podkarakteristike:
 - *prilagodljivost* – sposobnost softvera da se prilagodi različitim okruženjima bez dodatnih akcija (ili sredstava) u odnosu na one koje su već predviđene u softveru za tu svrhu

- *mogućnost instaliranja* – sposobnost softvera da bude instaliran u specificiranom okruženju
- *koegzistencija* – sposobnost softvera da radi u zajedničkom okruženju sa drugim nezavisnim softverima, deleći sa njima iste resurse
- *zamenljivost mesta* – sposobnost softvera da bude upotrebljen umesto drugog softvera sa istom svrhom i u istom okruženju
- *usaglašenost prenosivosti* – sposobnost softverskog proizvoda da poštuje standarde i konvencije povezane sa prenosivošću

Iz opisanog sadržaja modela može se zapaziti da se svaka podkarakteristika odnosi na tačno jednu karakteristiku (što nije slučaj kod McCall-ovog i Boehm-ovog modela). Osim toga, podkarakteristikama je opisan korisnikov pogled na softver, a ne interni pogled projektanta.

Model kvaliteta upotrebe sadrži četiri karakteristike kvaliteta vezane za upotrebu softvera, kao što je prikazano na slici 9.8.



Slika 9.8 ISO 9126 model kvaliteta upotrebe

Dodatak

Dodatak je namenjen samo studentima Univerziteta Singidunum koji polažu predmet Razvoj aplikativnog softvera. U okviru ovog predmeta, kao sastavni deo ispita, predviđena je izrada projekta. Cilj projekta je da studenti u praksi, tj. na primeru generisanja konkretnog softvera, prođu kroz sve faze životnog ciklusa softvera u onoj meri u kojoj fakultetsko okruženje to dopušta. Na ovaj način, studentima se pruža mogućnost da provere svoja teorijska znanja usvojena na predavanjima, kao i da steknu prva praktična iskustva u izradi softvera.

Projekat se radi timski na zadatu temu primenom poznatih tehnologija koje su raspoložive na fakultetu. Podrazumeva se da studenti vladaju razvojnim okruženjem, programskim jezikom i tehnikama programiranja koji će biti korišćeni u realizaciji softvera. Osim softvera, kao rezultat projekta očekuje se i dokument u pisanoj formi u kome će detaljno biti opisano izvođenje projekta po razvojnim fazama. Ovaj dokument sadrži samo delove koji su izvodljivi u datom okruženju. U zavisnosti od vrste projekta, dokument sadrži samo stavke primerene datom projektu.

U opštem slučaju, dokument sa opisom projekta treba da prati sledeću strukturu:

Naslovna strana

Na naslovnoj strani navesti naziv projekta, autore projekta i vreme izrade projekta.

1. Kratak opis projekta

Ukratko predstaviti temu i cilj projekta.

2. Postupak razvoja softvera

Izabrati jednu od tradicionalnih metoda modelovanja koja će biti primenjena u procesu razvoja konkretnog softvera (poglavlje 2.1). Preporučuju se kaskadni ili V model.

U skladu sa izabranom metodom, predložiti detaljan plan projekta u kome treba:

- definisati kritične tačke (*milestones*) koje pokazuju šta treba da bude urađeno na projektu i do kada
- navesti uloge svih članova projektnog tima uz precizan opis šta ko od njih treba da uradi i do kada
- navesti planirane rezultate za svaku kritičnu tačku; rezultati mogu biti softverski moduli, modeli, dokumentacija, kratka korisnička uputstva i sl.

3. Analiza zahteva

Izraditi specifikaciju softverskih zahteva. Posebnu pažnju posvetiti:

- funkcionalnim zahtevima sistema
- zahtevima kojima se definišu veze sistema sa okruženjem
- zahtevima po pitanju performansi koje sistem treba da ispuni

Specifikacija može, prema potrebi, da sadrži tekstualne opise zahteva, UML dijagrame, snimke ekrana, predloge izveštaja, tabela i dr.

4. Projektovanje sistema

Isprojektovati sistem sa aspekta njegove arhitekture i programskog kôda.

Predstaviti arhitekturu sistema u vidu modularne hijerarhije iz koje se jasno vidi koje komponente postoje u sistemu i kako su one međusobno povezane. Pri tome koristiti odgovarajući stil projektovanja (poglavlje 4.2).

Izabrati programski jezik koji će biti korišćen pri implementaciji i obrazložiti njegov izbor. Opisati strukture podataka i algoritme koji će biti primenjeni. Napraviti strukturu programskih modula, tj. datoteka pomoću kojih će sistem biti implementiran. Navesti potrebne procedure i funkcije u okviru svakog programskog modula.

Pri projektovanju intenzivno koristiti sve vrste UML dijagrama date u poglavlju 5.

5. Implementacija softvera

Na izabranom programskom jeziku napisati programski kôd koji realizuje sistem. Datoteke sa izvornim kôdom dopuniti odgovarajućom unutrašnjom dokumentacijom, tj. komentarima. Na jednoj strani, priložiti primer dobro dokumentovanog programskog kôda (početak datoteke sa zaglavljem).

6. Testiranje softvera

U okviru jediničnog testiranja po modulima, opisati sve testove koji su sprovedeni u cilju provere ispravnosti napisanog programskog kôda.

Navesti koji je princip integracionog testiranja korišćen (poglavlje 7.2.2) i dati hijerarhiju koja pokazuje redosled u kome su moduli testirani.

Nacrtati dijagram zavisnosti broja pronađenih grešaka u softveru po nedeljama.

7. Isporuka softvera

Napisati deo *Uputstva za korišćenje* koji se odnosi na jednu (izabranu) funkcionalnost sistema.

Literatura

1. S. L. Pfleeger, J. M. Atlee, *Softversko inženjerstvo: teorija i praksa*, prevod Računarski fakultet, Beograd, 2006.
2. A. Bijlsma, B. J. Heeren, E. E. Roubtsova, S. Stuurman, *Software Architecture*, Free Technology Academy, 2011.
3. I. Maršić, *Software Engineering*, Department of Electrical and Computer Engineering, Rutgers, State University of New Jersey, 2012.
4. Rational Software, *Rational Unified Process: Best Practices for Software Development Teams*, 2011.
5. M. Fowler, *UML kratko*, prevod Mikro knjiga, 2004.
6. D. Bojić, *Testiranje softvera*, skripta, Elektrotehnički fakultet Beograd, 2010.
7. K. Erdil, E. Finn, K. Keating, J. Meattle, S. Park, D. Yoon, *Software Maintenance As Part of the Software Life Cycle*, Department of Computer Science, Tufts University, USA, 2003.
8. R. E. Al-Quataish, *Quality Models in Software Engineering Literature: An Analytical and Comparative Study*, Journal of American Science 6(3), 2010.
9. R. Fitzpatrick, *Software Quality: Definitions and Strategic Issues*, Staffordshire University, 1996.
10. D. C. Rajapakse, *Practical Tips for Software-Intensive Student Projects*, 3rd edition, 2010.

CIP - Каталогизација у публикацији - Народна библиотека Србије, Београд
004.42:004.9(075.8)

ТОМАШЕВИЋ, Виолета, 1965-

Razvoj aplikativnog softvera / Violeta Tomašević. - 5. izd. - Beograd :
Univerzitet Singidunum, 2022 (Beograd : Caligraph). - 205 str. : ilustr. ;
24 cm

Tiraž 700. - Bibliografija: str. [206].

ISBN 978-86-7912-551-4

а) Апликативни софтвер - Пројектовање

COBISS.SR-ID 70517001

© 2022.

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovан u bilo kom vidu i
putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.



Violeta Tomašević

RAZVOJ APLIKATIVNOG SOFTVERA



Ova knjiga je nastala kao rezultat potrebe za odgovarajućim pisanim materijalom iz predmeta Razvoj aplikativnog softvera koji autor drži na četvrtoj godini Fakulteta za informatiku i računarstvo i Tehničkog fakulteta Univerziteta Singidunum u Beogradu. Pri pisanju je učinjen napor da knjiga bude prihvatljiva za čitaoce bez nekog većeg predznanja iz oblasti razvoja softvera. Namenjena je i prilagođena prosečnom studentu, jer je osnovni cilj autora bio da svi studenti koji slušaju predmet Razvoj aplikativnog softvera mogu na razumljiv i lak način da savladaju predviđeno gradivo.