1. **Pojam softvera**

Softverski produkt je skup raˇcunalnih programa i pripadne dokumentacije, stvoren zato da bi se prodao. Moˇze biti razvijen za sasvim odredenog korisnika (engleski bespoke product, customized product) ili op´cenito za trˇziˇste (engleski generic product).

Za današnji softver se podrazumijeva da on mora biti kvalitetan. Preciznije, od softverskog produkta se oˇcekuje da se on odlikuje sljede´cim atributima kvalitete.

• Mogu´cnost odrˇzavanja. Softver se mora mo´ci mijenjati u skladu s promijenjenim potrebama korisnika.

• Pouzdanost i sigurnost. Softver se mora ponaˇsati na predvidiv naˇcin, te ne smije izazivati fiziˇcke ili ekonomske ˇstete.

• Efikasnost. Softver mora imati zadovoljavaju´ce performanse, te on treba upravljati strojnim resursima na ˇstedljiv naˇcin.

• Upotrebljivost. Softver treba raditi ono ˇsto korisnici od njega oˇcekuju, suˇcelje mu treba biti zadovoljavaju´ce, te za njega mora postojati dokumentacija.

1. **Softverska tehnologija, softverski proces**

Softverska tehnologija obuhvata metode, alate, tehnike i procedure koje se koriste pri razvoju softvera.

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

Model za softverski proces je idealizirani prikaz softverskog procesa, kojim se odreduje poˇzeljni naˇcin odvijanja i medusobnog povezivanja osnovnih aktivnosti. Na primjer, model moˇze zahtijevati sekvencijalno odnosno simultano odvijanje aktivnosti.

1. **Elementi softverskog procesa**

U svim modelima viˇse ili manje su prisutne sljede´ce osnovne aktivnosti koje ˇcine softverski proces:

• Specifikacija. Analiziraju se zahtjevi korisnika. Utvrduje se ˇsto softver treba raditi.

• Oblikovanje (engleski design.) Oblikuje se grada sustava, naˇcin rada komponenti, te suˇcelje izmedu komponenti. Dakle projektira se rjeˇsenje koje odreduje kako ´ce softver raditi.

• Implementacija (programiranje). Oblikovano rjeˇsenje realizira se uz pomo´c raspoloˇzivih programskih jezika i alata.

• Verifikacija i validacija. Provjerava se da li softver radi prema specifikaciji, odnosno da li radi ono ˇsto korisnik ˇzeli. Obiˇcno se svodi na testiranje, mada postoje i druge tehnike.

• Odrˇzavanje odnosno evolucija. Nakon uvodenja u upotrebu, softver se dalje popravlja, mijenja i nadograduje, u skladu s promijenjenim potrebama korisnika.

1. **Analiza zahteva softvera**

Same zahtjeve dijelimo na dvije vrste:

• Funkcionalni zahtjevi. Opisuju funkcije sustava, dakle usluge koje bi on trebao obavljati, izlaze koje on daje za zadane ulaze, te njegovo ponaˇsanje u pojedinim situacijama.

• Nefunkcionalni zahtjevi. Izraˇzavaju ograniˇcenja na funkcije sustava, na primjer traˇzeno vrijeme odziva, traˇzenu razinu pouzdanosti, dozvoljeno zauze´ce memorije, obavezu koriˇstenja odredenog programskog jezika, poˇstivanje odredenih standarda.

Otkrivanje i analiza zahtjeva. Otkrivaju se zahtjevi, tako da se promatraju postoje´ci sustavi, analiziraju radni procesi, intervjuiraju budu´ci korisnici i njihovi manageri, itd. Na taj naˇcin stvaraju se modeli sustava (vidi Odjeljak 2.2), a ponekad i prototipovi (vidi Odjeljak 2.3), sve u cilju boljeg razumijevanja sustava kojeg treba stvoriti.

1. **Specifikacija softvera**

Specifikacija je poˇcetna faza softverskog procesa. Analiziraju se zahtjevi na budu´ci sustav. Rezultat je dokument o zahtjevima, koji opisuje ˇsto sustav treba raditi, po mogu´cnosti bez prejudiciranja kako da se to postigne. Specifikacijom se bavi tim sastavljen od razvijaˇca softvera i budu´cih korisnika.

Aktivnost specifikacije detaljnije se moˇze podijeliti u sljede´ce podaktivnosti.

• Studija izvodljivosti. Procjenjuje se da li se uoˇcene potrebe korisnika mogu zadovoljiti uz pomo´c dostupnih hardverskih i softverskih tehnologija, da li bi predloˇzeni sustav bio isplativ u poslovnom smislu, te da li sustav moˇze biti razvijen s raspoloˇzivim budˇzetom.

• Otkrivanje i analiza zahtjeva. Otkrivaju se zahtjevi, tako da se promatraju postoje´ci sustavi, analiziraju radni procesi, intervjuiraju budu´ci korisnici i njihovi manageri, itd. Na taj naˇcin stvaraju se modeli sustava (vidi Odjeljak 2.2), a ponekad i prototipovi (vidi Odjeljak 2.3), sve u cilju boljeg razumijevanja sustava kojeg treba stvoriti.

• Utvrdivanje zahtjeva. Informacije skupljene analizom pretvaraju se u tekstove koji definiraju zahtjeve. Postoje dvije razine u opisivanju zahtjeva: “korisniˇcki zahtjevi” i “zahtjevi na sustav”.

• Validacija zahtjeva. Provjerava se da li su zahtjevi realistiˇcni (mogu se ostvariti raspoloˇzivom tehnologijom i budˇzetom), konzistentni (nisu u medusobnom konfliktu), te potpuni (ukljuˇcene su sve potrebne funkcije i ograniˇcenja).

1. **Procesi planiranja zahteva**

Planiranje i pra´cenje projekta predstavlja redoviti, svakodnevni i najegzaktniji posao softverskog managera. Planiranje i pra´cenje ukljuˇcuje sljede´ce:

* definiranje projektnih aktivnosti, njihovog trajanja i meduzavisnosti,
* odredivanje kalendara poˇcetka i zavrˇsetka svake aktivnosti,
* rasporedivanje ljudi i drugih resursa na aktivnosti,
* pra´cenje izvrˇsenja aktivnosti,
* povremena revizija svih parametara.

1. **Planiranje**

Planiranje predstavlja, kao proces, menadžersku funkciju kojom se određuju ciljevi tekućeg poslovanja i razvoja organizacije, zatim predviđanje i definisanje budućih zadataka, kao i uslova u kojima ti zadaci treba da se izvršavaju a ciljevi ostvaruju. Ono predstavlja osnovu za obavljanje drugih menadžerskih aktivnosti: organizovanja, vođenja i kontrole.

Planovi, u zavisnosti od vremena koje obuhvataju, mogu da budu:

dugoročni — imaju strategijski karakter i prave se po pravilu za rok preko 10 godina

srednjoročni — prave se za rok do 5 godina

kratkoročni — prave se na period do oko godinu dana

1. **Implementacija**

Implementacija (programiranje). Oblikovano rjeˇsenje realizira se uz pomo´c raspoloˇzivih programskih jezika i alata.

Nakon oblikovanja slijedi implementacija, tj. svi dijelovi predvideni designom realiziraju se uz pomo´c pogodnog programskog jezika. Zadnje iteracije oblikovanja obiˇcno se preklapaju s implementacijom. Takoder, implementacija se donekle preklapa s verifikacijom (testiranjem), budu´ci da se programeri odmah bave debuggiranjem.

1. **Integracija, verifikacija i validacija**

Testiranje integracije provodi se nakon ˇsto se manji dijelovi softvera udruˇze u ve´cu cjelinu. Cilj testiranja je otkriti greˇske koje nastaju zato ˇsto dijelovi na pogreˇsan naˇcin koriste medusobna suˇcelja. U skladu sa Slikom 4.9, test primjeri se primjenjuju na cjelinu, a ne na pojedine dijelo  
  
Verifikacija (Are we building the product right?) je provjera da li softver odgovara specifikaciji, dakle dokumentu o zahtjevima.

Validacija (Are we building the right product?) je provjera da li softver zadovoljava “stvarnim” potrebama korisnika.

Metode za verifikaciju i validaciju ugrubo se mogu podijeliti u sljede´ce dvije skupine.

• Statiˇcka V&V (vidi Odjeljak 4.2). Svodi se na analizu i provjeru dokumenata, modela sustava, designa, odnosno programskog koda. Odvija se bez stvarnog izvodenja softvera.

• Testiranje (vidi Odjeljak 4.3). Svodi se na pokusno izvodenje softvera ili njegovih dijelova, na umjetno pripravljenim podacima, uz paˇzljivo analiziranje rezultata.

1. **Evolucija softvera**

Proces mijenjanja softvera nazivamo odrˇzavanje odnosno evolucija. Za neke ove dvije rijeˇci su sinonimi, s time da je prva tradicionalna i posudena iz tehnike, a druga je novija i toˇcnija. Mi ´cemo pod odrˇzavanjem podrazumijevati planirani rutinski dio procesa koji se sastoji od manje radikalnih promjena, dok ´ce evolucija oznaˇcavati ukupni proces s nepredvidivim elementima koji moˇze dovesti do bitnih promjena u arhitekturi softvera i koriˇstenoj tehnologiji.

Prema autoru Warren-u (1998), postoje sljede´ce tri strategije koje se medusobno dopunjuju.

• Odrˇzavanje softvera. Promjene se rade zato da bi se drˇzao korak s (promijenjenim) zahtjevima. Mijenja se funkcionalnost, no struktura softvera uglavnom ostaje ista.

• Arhitekturna transformacija. Znaˇcajna promjena arhitekture softverskog sustava. Na primjer, sustav se iz centralizirane mainframe arhitekture prebacuje u distribuiranu arhitekturu s klijentima i posluˇziteljima. Funkcionalnost se moˇze ili ne mora promijeniti.

• Softversko re-inˇzenjerstvo. Ne dodaje se nikakva nova funkcionalnost. Takoder, nema velike promjene u arhitekturi. Umjesto toga, sustav se transformira u oblik koji se lakˇse moˇze razumijeti i dokumentirati, te obradivati raspoloˇzivim alatima. Na primjer, programski kod se automatski prebacuje iz jednog (zastarjelog) programskog jezika u drugi (suvremeni).

1. **Softverski zahtevi**

Same zahtjeve dijelimo na dvije vrste:

• Funkcionalni zahtjevi. Opisuju funkcije sustava, dakle usluge koje bi on trebao obavljati, izlaze koje on daje za zadane ulaze, te njegovo ponaˇsanje u pojedinim situacijama.

• Nefunkcionalni zahtjevi. Izraˇzavaju ograniˇcenja na funkcije sustava, na primjer traˇzeno vrijeme odziva, traˇzenu razinu pouzdanosti, dozvoljeno zauze´ce memorije, obavezu koriˇstenja odredenog programskog jezika, poˇstivanje odredenih standarda.

1. **Faktori koji utiču na održavanje**

U nastavku nastojimo popisati osnovne faktore koji utjeˇcu na cijenu odrˇzavanja.

1. Cjelovitost polazne specifikacije. Ukoliko odmah ukljuˇcimo sve zahtjeve, kasnije ´ce biti manje perfekcijskog odrˇzavanja.

2. Dobrota oblikovanja. Dobar design je jeftiniji za odrˇzavanje. Smatra se da su sa stanoviˇsta odrˇzavanja najbolji objektno-oblikovani sustavi, koji se sastoje od malih modula s jakom unutraˇsnjom kohezijom i labavim vezama prema van. 5.2. UPRAVLJANJE KONFIGURACIJOM 55

3. Naˇcin implementacije. Kod u “stroˇzem” programskom jeziku poput Jave lakˇse se odrˇzava nego kod u jeziku poput C-a. Strukturirani kod (if, while) sa smisleno imenovanim varijablama razumljiviji je od kompaktnog koda s mnogo goto naredbi.

4. Stupanj verificiranosti. Dobro verificirani softver ima manje greˇsaka pa ´ce zahtijevati manje korekcijskog odrˇzavanja.

5. Stupanj dokumentiranosti. Uredna, dobro strukturirana i cjelovita dokumentacija olakˇsava razumijevanje softvera, te na taj naˇcin pojeftinjuje odrˇzavanje.

6. Naˇcin upravljanja konfiguracijom. Ukoliko se primjenjunju metode, alati i organizacijska pravila upravljanja konfiguracijom, tada je odrˇzavanje na dugi rok jeftinije.

7. Starost softvera. Sto je softver stariji, to je skuplji za odrˇzavanje, budu´ci da mu se grad ˇ a degradirala, ovisan je o zastarjelim razvojnim alatima, a dokumentacija mu je postala neaˇzurna.

8. Svojstva aplikacijske domene. Ako je rijeˇc o stabilnoj domeni gdje se poslovna pravila rijetko mijenjaju, tada ´ce se rijetko pojavljivati potreba za perfekcijskim odrˇzavanjem u svrhu uskladivanja s novim pravilima.

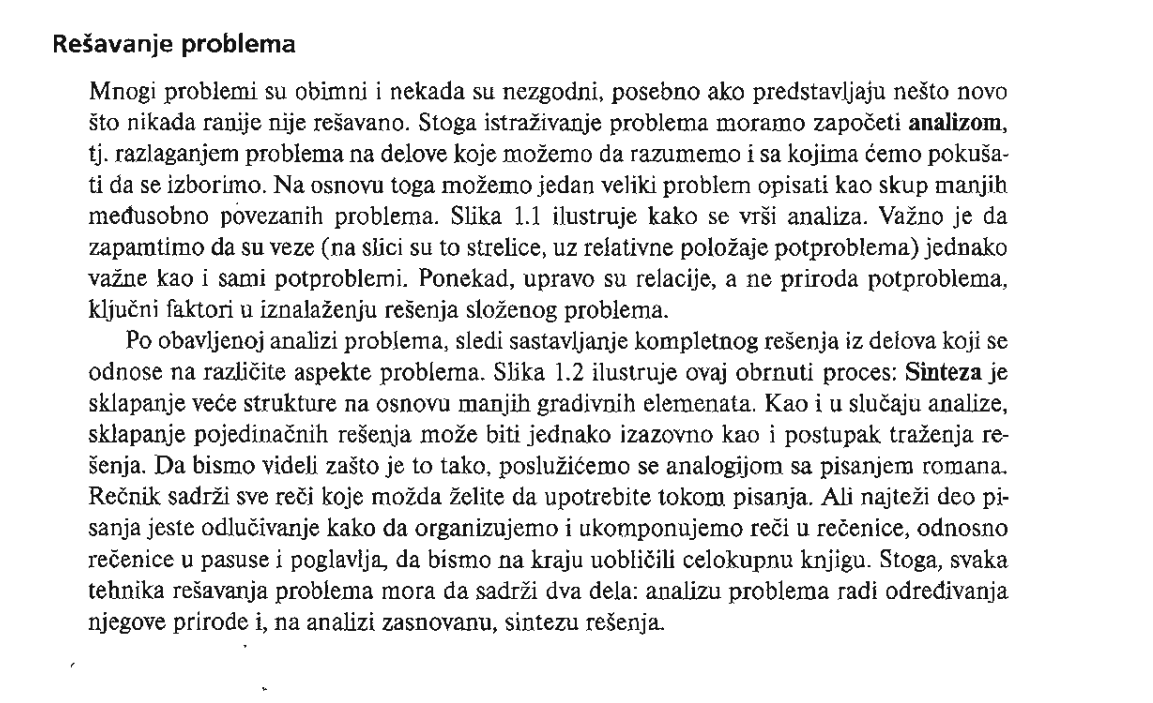
9. Stabilnost razvojnog tima. Odrˇzavanje je jeftinije ako se njime bave originalni razvijaˇci softvera, jer oni ne moraju troˇsiti vrijeme na upoznavanje sa softverom.

10. Stabilnost platforme. Ako smo softver implementirali na platformi koja ´ce joˇs dugo biti suvremena, tada ne´ce trebati adaptacijsko odrˇzavanje.

1. **Održavanje**

Odrˇzavanje softvera je kontinuirani rutinski proces koji se svodi na niz uzastopnih promjena. Kao rezultat odrˇzavanja nastaje velik broj razliˇcitih verzija softvera, od kojih svaka predstavlja neˇsto drukˇciju konfiguraciju sastavnih dijelova. Cijeli proces treba paˇzljivo planirati i kontrolirati. Takoder je neophodno imati toˇcnu evidenciju svih verzija i njihovih konfiguracija. Skup potrebnih metoda, alata i organizacijskih zahvata naziva se upravljanje konfiguracijom

1. **Rešavanje problema**



1. **Planirano održavanje**

Planirano održavanje (eng. scheduled maintenance) je vrsta održavanja softverskog sistema koja se svesno planira unapred, kako bi se unapredile performanse, poboljšala funkcionalnost ili sprečile moguće greške u budućnosti.

Ova vrsta održavanja se ne vrši zbog grešaka koje su se već pojavile, već radi:

* prilagođavanja promenama u okruženju (npr. novi operativni sistem, nova verzija baze podataka),
* poboljšanja performansi sistema,
* unapređenja bezbednosti,
* dodavanja funkcionalnosti koje su unapred planirane (ali ne kao hitna reakcija),
* refaktorisanja koda da bi buduće izmene bile lakše.

Cilj planiranog održavanja je očuvanje kvaliteta softvera tokom vremena i njegovo prilagođavanje budućim potrebama.

1. **Redizajn sistema**

Redizajn softvera, tj. reinženjering softvera se može definisati kao pristup kojim se analiziraju postojeći softver i njegova primena u praksi, uz nastojanje da se unapredi efikasnost korišćenja (eksploatacije) primenom fundamentalnih i radikalnih pristupa izmenama ili eliminaciji funkcija i opcija, njihovim unapređivanjem, te poboljšanjem strukture i performansi

Softverski reinženjering se definiše kao „ispitivanje i zamena postojećeg sistema novim sistemom koji ga rekonstituiše (drugačije je osmišljen i drugačje implementiran u odnosu na prvobitni sistem). Potreban je sistematski pristup analizi i redizajnu softverskih funkcija, tokova podataka i strukture organizacije, kako bi se unapredio kvalitet usluga i/ili proizvoda, uz smanjenje troškova i vremena rada.

Najčešće se softverski reinženjering realizuje u procesu od nekoliko koraka:

1. reverzni inženjering ,

2. restrukturiranje - modifikacija apstraktne forme nastala analizom u reverznom inženjeringu (npr. modela) u drugi oblik modela,

3. forward inženjering – implementacijom novog sistema koji će imati osobine u skladu sa modifikacijama i biće implementiran na drugačiji način,

4. (po potrebi) dodatne modifikacije unapređenjem sistema u skladu sa novim zahtevima, koji nisu podržani u prethodnom sistemu.

1. **Refaktoring**

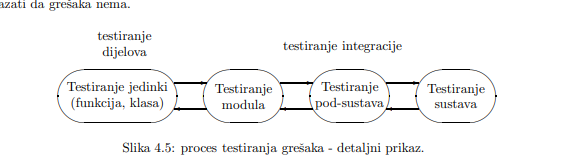
Refaktorisanje koda je proces restrukturiranja postojećeg računarskog koda - promjena kompozicije - bez promjene njegovog spoljašnjeg ponašanja. Refaktorisanje je namijenjeno poboljšanju nefunkcionalnih atributa softvera. Prednosti uključuju poboljšanu čitljivost koda i smanjenu složenost; ovo može olakšati održavanje izvornog koda i stvoriti ekspresivniju unutrašnju arhitekturu ili objektni model da bi se poboljšala proširivost.

1. **Procesi testiranja, tehnike**

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

• Testiranje greˇsaka. Sluˇzi za provjeru funkcionalnih zahtjeva. Namjera je otkriti odstupanja izmedu softvera i njegove specifikacije. Ta odstupanja su posljedica greˇsaka u softveru. Testpodaci su konstruirani tako da otkriju prisustvo greˇske, a ne tako da simuliraju normalnu upotrebu softvera.

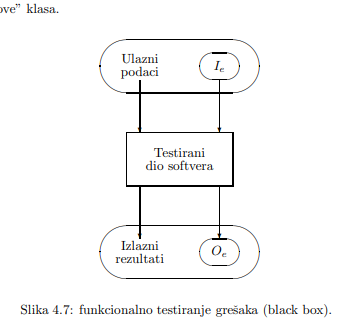
• Statistiˇcko testiranje. Sluˇzi za provjeru nekih oblika ne-funkcionalnih zahtjeva. Namjera je na primjer izmjeriti performanse softvera ili stupanj njegove pouzdanosti. Test-podaci su odabrani tako da liˇce na stvarne korisniˇcke podatke, te da odraˇzavaju njihovu stvarnu statistiˇcku razdiobu. Obiˇcno je rijeˇc o velikom broju sluˇcajno odabranih test primjera, ˇsto omogu´cuje statistiˇcku procjenu performansi ili pouzdanosti.



1. **Testiranje komponenti**

Da bi testiranje dijela softvera bilo uspjeˇsno, vaˇzno je znati odabrati test podatke. U nastavku ´cemo promatrati nekoliko pristupa koji se razlikuju po naˇcinu odabira test podataka.

**Funkcionalno testiranje** (engleski black-box testing). Test-primjeri izvode se iskljuˇcivo iz specifikacije dotiˇcnog dijela softvera. Ili, drukˇcije reˇceno, softver se promatra kao “crna kutija” o kojoj jedino znamo da bi ona (prema specifikaciji) za zadane ulaze trebala proizvesti zadane izlaze. Zbog postojanja greˇsaka, ulazi iz odredenog (nepoznatog) skupa Ie izazvat ´ce neispravno ponaˇsanje, koje ´cemo prepoznati po izlazima iz skupa Oe. To je sve prikazano na Slici 4.7. U postupku testiranja nastojimo izabrati ulaze za koje postoji velika vjerojatnost da pripadaju skupu Ie. Izbor takvih test primjera najˇceˇs´ce se zasniva na iskustvu softverskog inˇzenjera. Ipak, mogu´c je i sistematiˇcniji pristup zasnovan na podjeli skupa svih mogu´cih ulaznih podataka (domene) na klase. Ovdje rijeˇc “klasa” nije upotrebljena u smislu objektno-orijentiranog pristupa, nego u smislu matematiˇckog pojma “klasa ekvivalencije”. Oˇcekujemo da ´ce se program ponaˇsati sliˇcno za sve podatke iz iste klase. Biramo barem po jedan test primjer iz svake klase. Takoder je dobro isprobati “rubove” klasa.



**Strukturalno testiranje** (engleski white-box testing). Dodatni test-primjeri izvode se na osnovu poznavanja interne strukture testiranog dijela softvera i koriˇstenih algoritama. Dakle sluˇzimo se analizom programskog koda dotiˇcnog dijela softvera da bi odabrali dodatne test primjere, tj. da bi bolje podijelili domenu na klase. Kao primjer, pogledajmo u Prilogu 4.6 implementaciju u Javi naˇse procedure za traˇzenje elementa u polju. Koristi se algoritam binarnog traˇzenja. Specifikacija je sada malo stroˇza, naime polje mora biti uzlazno sortirano. Promatranjem programskog koda uoˇcavamo da svaki korak algoritma dijeli polje na tri dijela. Zato osim izbora elementa Key na rubovima te na sredini polja T uvodimo i dva nova izbora: neposredno ispred i iza srednjeg elementa u polju - vidi Prilog 4.7. Nakon revizije starih test primjera (polje mora biti sortirano), te nakon dodavanja dvaju novih test primjera, dobivamo ukupno 8 test primjera iz Priloga 4.8

**Testiranje po putovima** (engleski path testing). Rijeˇc je o posebnoj vrsti strukturalnog testiranja, gdje se test primjeri biraju tako da se isproba svaki “nezavisni put” kroz dijagram toka kontrole testiranog dijela softvera. Dakle najprije se mora nacrtati dijagram toka kontrole dotiˇcnog dijela softvera. Dijagram se crta tako da se naredbe while i if - then - else prikaˇzu obrascima sa Slike 4.8, a ostale naredbe se prikaˇzu svaka po jednim ˇcvorom. Uoˇcavaju se ˇcvorovi koji predstavljaju “poˇcetak” i “kraj” toka kontrole. Testiraje po putovima zahtijeva da se konstruiraju takvi test primjeri koji ´ce isprobati svaki od nezavisnih putova kroz dijagram. Nezavisni put je onaj koji ide od “poˇcetka” do “kraja” i prolazi bar jednim novim lukom.

1. **Testiranje sistema**

Testiranje integracije provodi se nakon ˇsto se manji dijelovi softvera udruˇze u ve´cu cjelinu. Cilj testiranja je otkriti greˇske koje nastaju zato ˇsto dijelovi na pogreˇsan naˇcin koriste medusobna suˇcelja. U skladu sa Slikom 4.9, test primjeri se primjenjuju na cjelinu, a ne na pojedine dijelov

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

Testiranje integracije posebno je teˇsko zato ˇsto se greˇske mogu pojaviti samo u iznimnim situacijama (veliko optere´cenje sustava) ili pak na neoˇcekivanim mjestima (u sasvim drugom dijelu softvera). Slijede op´ceniti naputci za testiranje integracije.

1. **Testiranje performansi**

Testiranje performansi je vrsta testiranja softvera kojom se proverava brzina, odziv, stabilnost i skalabilnost sistema pod određenim uslovima opterećenja. Cilj je da se osigura da softver zadovoljava zahteve u pogledu vremena odziva, iskorišćenosti resursa i ponašanja pod stresom.

* Ne fokusira se na tačnost funkcionalnosti, već na efikasnost i ponašanje softvera pod opterećenjem.
* Obuhvata testiranje brzine (odziva sistema), propusnosti (koliko zahteva može da obradi), stabilnosti i pouzdanosti u uslovima opterećenja.

#### **Vrste testiranja performansi:**

1. **Load testing (testiranje opterećenja)** – proverava kako sistem radi pri normalnom i povećanom broju korisnika.
2. **Stress testing (testiranje u uslovima stresa)** – testira granice sistema, kako se ponaša kada je preopterećen ili kada resursi nisu dovoljni.
3. **Soak testing (dugoročno testiranje)** – proverava stabilnost sistema kada je dugo vremena pod opterećenjem.
4. **Spike testing** – proverava ponašanje sistema kada naglo dođe do velikog povećanja broja korisnika.
5. **Kako obezbediti kvalitet softvera?**

Obezbeđivanje kvaliteta softvera znači primeniti postupke, metode i alate tokom celog softverskog procesa kako bi se osiguralo da konačni proizvod bude funkcionalan, pouzdan, bezbedan, efikasan i prilagođen korisnicima. Kvalitet se ne postiže samo testiranjem gotovog softvera, već kontinuiranim upravljanjem kvalitetom tokom celog životnog ciklusa.

Kvalitet softvera se ne dešava slučajno – on se planira, sprovodi, proverava i poboljšava. Obezbeđenje kvaliteta je timski zadatak koji obuhvata sve faze razvoja, a ne samo završno testiranje. Kvalitetan softver štedi novac, povećava zadovoljstvo korisnika i osigurava dugotrajnost sistema.

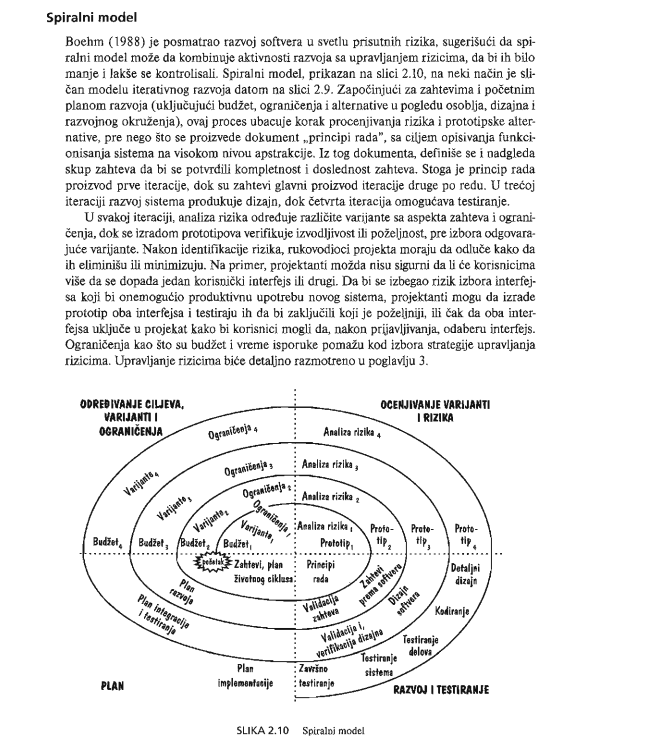
1. **Troškovi**

Ukupna cijena odrˇzavanja softvera dostiˇze i prelazi cijenu njegovog originalnog razvoja. Cijenu odrˇzavanja moˇzemo smanjiti opet tako da pove´camo kvalitetu originalnog razvojnog procesa. Uz ve´cu kvalitetu cijena razvoja ´ce biti ve´ca, ali ´ce se stvoriti softver koji je jeftiniji za kasnije odrˇzavanje. Ova ideja ilustrirana je Slikom 5.1. U nastavku nastojimo popisati osnovne faktore koji utjeˇcu na cijenu odrˇzavanja.

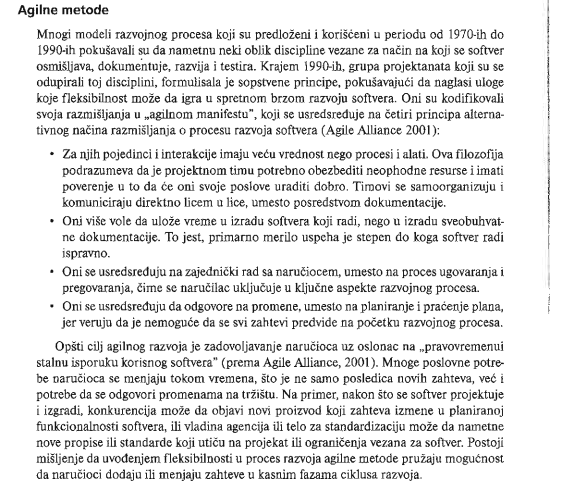
1. Cjelovitost polazne specifikacije. Ukoliko odmah ukljuˇcimo sve zahtjeve, kasnije ´ce biti manje perfekcijskog odrˇzavanja.

2. Dobrota oblikovanja. Dobar design je jeftiniji za odrˇzavanje. Smatra se da su sa stanoviˇsta odrˇzavanja najbolji objektno-oblikovani sustavi, koji se sastoje od malih modula s jakom unutraˇsnjom kohezijom i labavim vezama prema van.

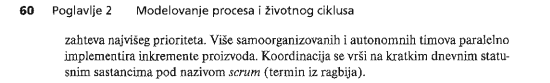
1. **Spiralni model**



1. **Agilan razvoj softvera**



1. **Scrum metodologija**



1. **Pouzdanost softvera**

Pouzdanost softvera predstavlja sposobnost softverskog sistema da kontinuirano funkcioniše bez grešaka tokom određenog vremenskog perioda, u definisanim uslovima rada. To je jedna od najvažnijih osobina kvaliteta softvera, posebno kod sistema koji su kritični za bezbednost, finansije ili zdravlje.

#### **Ključni elementi pouzdanosti:**

1. **Bezgrešno izvršavanje**

* Softver funkcioniše tačno onako kako je definisano u specifikaciji, bez neželjenih ponašanja.

1. **Otpornost na greške (robustnost)**

* Softver se stabilno ponaša čak i kada korisnik unese pogrešne podatke ili dođe do nepredviđenih situacija.

1. **Predvidivo ponašanje**

* Softver daje očekivane rezultate pri jednakim uslovima.

1. **Oporavak od grešaka**

* Ako dođe do greške, sistem je sposoban da se automatski oporavi ili bezbedno zaustavi rad.

#### **Kako povećati pouzdanost softvera:**

* Kvalitetna **specifikacija zahteva** i dizajn sistema.
* **Temeljno testiranje**: jedinica, integracija, sistemsko, stres testiranje.
* **Kod review** i statička analiza koda.
* **Upotreba proverenih biblioteka i alata.**
* Upravljanje izuzetnim situacijama (try-catch, fallback mehanizmi).
* **Praćenje i logovanje grešaka** u realnom vremenu.
* **Održavanje softvera** – redovno uklanjanje bagova, ažuriranje i unapređenje.

1. **Dizajn softvera u realnom vremenu**

**Da li dizajniramo u realnom vremenu ili dizajniramo softver koji radi u realnom vremenu?**

1. **Objektno orijentisan dizajn**

Rijeˇc je o takvom pristupu oblikovanju gdje projektant prvenstveno razmiˇslja o “objektima” (predmetima, osobama, pojavama, . . . ) a manje o “funkcijama” (operacijama, aktivnostima, procesima, . .. ).

Osnovni pojmovi unutar objektnog oblikovanja su objekt i klasa.

• Objekt je entitet uz kojeg je vezan skup atributa i skup operacija. Vrijednosti atributa vezanih uz

objekt odreduju stanje tog objekta. Operacije vezane uz objekt mijenjaju vrijednosti objektovih

atributa, te na taj naˇcin djeluju na stanje tog objekta.

• Klasa je skup istovrsnih objekata Definicija klase sluˇzi kao obrazac za stvaranje objekata. Definicija klase sadrˇzi deklaracije svih atributa i operacija vezanih uz objekt iz te klase.

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

Vaˇzno svojstvo objektno-oblikovanog sustava je da on omogu´cuje laganu evoluciju. Naime, sustav se sastoji od objekata, dakle dijelova s jakom unutarnjom kohezijom i labavim vezama prema van. Evolucija sustava zahtijevat ´ce promjenu unutarnje grade pojedinog objekta ili dodavanje novih objekata - takvi zahvati nemaju znaˇcajnog uˇcinka na ostatak sustava (vidi primjer na kraju ovog odjeljka).

1. **Dizajn softvera baziran na servisima**

Inženjering softvera zasnovan na servisima (Service-oriented Software Engineering – SOSE), takođe poznat kao inženjering servisa, predstavlja metodologiju softverskog inženjeringa koja je usmerena na razvoj softverskih sistema putem kompozicije (sastavljanja) ponovo upotrebljivih servisa (orijentacija na servise), koji su često obezbeđeni od strane drugih provajdera servisa.

Pošto uključuje kompoziciju, SOSE deli mnoge karakteristike sa inženjeringom softvera zasnovanim na komponentama (component-based software engineering) – pristupom u kojem se softverski sistemi grade od ponovo upotrebljivih komponenti – ali dodaje mogućnost dinamičkog pronalaženja potrebnih servisa u toku izvođenja (run-time). Ti servisi mogu biti obezbeđeni od strane drugih, na primer kao veb servisi, ali suštinski element SOSE-a je dinamička veza između korisnika servisa i provajdera servisa.