SOFTVERSKO INŽENJERSTVO

Skripta za usmeni ispit

By: **~**

1. Šta je softversko inženjerstvo?

   Softversko inženjerstvo je inženjerska disciplina koja se bavi teorijom, metodama i alatima za profesionalni razvoj softvera. I bavi se efikasnim (cost-effective) razvojem softvera. Predstavlja sistematski prilaz razvoju i evoluciji softvera.

2. Šta je softver?

   Softver je računarski program, ali ne samo program već i pridružena dokumentacija i konfiguracioni podaci neophodni da bi softver radio korektno.

3. Tipovi softverskih proizvoda.

   Postoje 2 osnovna tipa softverskih proizvoda:
   - Generički (generic) – samostalini sistemi namenjeni za prodaju na slobodnom tržištu, firma koja razvija softver definiše specifikaciju angažujući ljude iz domena za koji se razvija softver
   - Ugovorni (custom) – razvijeni za jednog korisnika prema njegovim zahtevima, specifikaciju definiše korisnik za koga se razvija softver

   Dakle, osnovna razlika između ova 2 tipa je ko definiše specifikaciju.

4. Razlike između softverskog inženjerstva i informatike.

   Informatika je nauka o računarstvu i ona se bavi teorijom i osnovama računarstva. Dok se softversko inženjerstvo bavi praktičnom stranom razvoja i isporuke korisnog softvera. Informatika, kao teoretska strana predstavlja solidnu osnovu za softversko inženjerstvo.

5. Koje su razlike između sistemskog i softverskog inženjerstva?

   Sistemsko inženjerstvo je inženjerska disciplina koja se bavi aspektima razvoja sistema zasnovanih na računarima ali se pored softvera bavi i hardverskim i procesnim inženjeringom. Dakle, sistemsko inženjerstvo predstavlja opštiji pojam od softverskog inženjerstva, odnosno softversko inženjerstvo predstavlja deo sistemskog inženjerstva. Može se reći da je sistemsko inženjerstvo šira i starija disciplina od softverskog inženjerstva jer obuhvata i hardverski i procesni inženjering, ali softversko inženjerstvo predstavlja njegov bitan deo.

6. Šta se softvarski proces?

   Softverski proces je struktuirani skup aktivnosti i pridruženih rezultata čiji je cilj proizvodnja softvera. Aktivnosti koje su zajedničke za sve softveske procese su:
   - Sprecifikacija – definisanje šta sistem treba da radi tj. jasno definisanje onoga što se očekuje od sistema
   - Projektovanje i implementacija – definisanje organizacije sistema i njegova implementacija
   - Validacija – provera da li sistem radi ono što naručioc želi; u suštini, ogleda se u podudarnosti zahteva naručioca i onoga što sistem zapravo radi; ne treba validaciju mešati sa verifikacijom; verifikacija je provera podudaranja onoga što sistem radi i onoga što je navedeno u specifikaciji; sistem može da prođe verifikaciju, ali da ne prođe validaciju, ukoliko se specifikacija nije urađena dobro, u skladu sa zahtevima naručioca
   - Evolucija – promena sistema u skladu sa promenom zahteva ili potreba naručioca; nije isto što i održavanje; održavanje podrazumeva da se stalno ulaže napor da postojeći sistem dobro radi, sve do trenutka dok se ne ukine; sistem može da se održava takav kakav jeste, ali da ne evoluira, ako se ne menja u skladu sa promenama potreba

7. Šta je model razvoja softverskog procesa?

Model softverskog procesa predstavlja uprošćenu reprezentaciju softverskog procesa koja predstavlja jedan pogled na ovaj proces. Definiše redosled u kome se izvršavaju aktivnosti vezane za softverski proces.

8. Opšti modeli za razvoj softvera.

Opšti modeli za razvoj softvera su:

- Model vodopada – aktivnosti vezane za softverski proces izvršavaju se u strogo definisanom redosledu i strogo sekvencijalno
- Iterativni razvoj – razvija se deo po deo finalnog proizvoda i tako gotovi delovi se odmah isporučuju klijentu
- Razvoj zasnovan na korišćenju gotovih komponenti – koriste se već gotove softvreske komponente, a sam razvoj se zasniva na projektovanju komunikacije među gotovim modulima

9. Šta su metode softverskog inženjerstva?

Metode softverskog inženjerstva predstavljaju struktuirani prilaz razvoju softvera. Jedna metoda obuhvata sledeće:

- Opise modela – opisi grafičkih modela koji nastaju u toku razvoja
- Pravila – ograničenja primenjena na model sistema
- Preporuke – saveti za dobro projektovanje
- Vodič kroz proces – uputstvo o tome kako teku aktivnosti

10. Šta su CASE alati?

CASE alati su softver koji služi za razvoj softvera. To su softverski sistemi namenjani pružanju automatske podrške aktivnostima softverskog procesa. Postoje CASE alati višeg i nižeg nivoa.

- U CASE alate višeg nivoa spadaju alati koji podržavaju aktivnosti procesa (inženjering zahteva i projektovanja).
- U CASE alate nižeg nivoa spadaju alati koji podržavaju kasnije aktivnosti (programiranje, debagiranje i testiranje)

11. Šta je sistem?

Sistem je kolekcija međusobno povezanih komponenti koje rade zajedno radi ostvarenja nekog zajedničkog cilja. Sistem obuhvata softver, hardver (mehaničke, električne i elektronske komponente) i ljude. Kod sistema sve ove komponente međusobno su povezane i svojstva i ponašanje svake od njih utiče na rad ostalih.

12. Kategorije sistema

Softver u opštem slučaju nije uvek deo sistema ali kada posmatramo sisteme koji uključuju softver oni se mogu razvrstati u dve kategorije:

- Tehnički sistemi zasnovani na računaru
- Socio-tehnički sistemi

Tehnički sistemi zasnovani na računaru su sistemi koji uključuju hardver i softver ali ne i procedure i procese. Pojedinci i organizacije koriste ove sisteme za neke namene ali znanje o tim namenama nije ugrađeno u tehnički sistem. Primeri ovih sistema su: televizor, mobilni telefon, PC softver.

Socio-tehnički sistemi su sistemi koji uključuju jedan ili više tehničkih sistema, ali takođe imaju i znanje o tome kako se koristi sistem da bi se postigao cilj. Ovi sistemi definišu radne procese koji su prilagođeni poslovnim i drugim procesima organizacije kojoj su namenjeni. Primeri ovakvih sistema su: sistem za publikovanje, satelitska stanica

13. Šta (Koja) su dodatna svojstva?

Dodatna svojstva su svojstva sistema kao celine i posledica su veza između komponenti sistema stoga se mogu utvrditi i meriti tek nakon integracije komponenti u sistem. Neki od primera dodatnih svojstava su:

- Ukupna težina sistema – primer svojstva koje se računa na osnovu svojstva svake od pojedinih komponenti. Određuje se tek kada imamo ceo sistem
- Pouzdanost sistema – najpre se računa pouzdanost svake od komponenti u zavisnosti od njenih karakteristika, od tih vrednosti se formulama računa pouzdanost sistema. Pouzdanost celog sistema zavisiće najviše od najmanje pouzdane komponente
- Upotrebljivost sistema – predstavlja podoblast HCI i ima veliku ulogu, zavisi, pored softvera i hardvera sistema, i od operatera i okruženja gde se koristi kao i od interfejsnog dela i toga koliko je lako nekom operateru da sa tim sistemom radi.

14. Proces sistemskog inženjerstva (Modeli razvoja sistema)

Obično se koristi model vodopada budući da treba paralelno razvijati različite delove sistema. Između faza se nalazi mali broj iteracija jer su promene hardverskih komponenti vrlo skupe pa softver mora kompenzovati hardverske probleme. U sistemskom inženjerstvu neizbežno je uključivanje velikog broja ljudi različitih profila i, specifično, inženjera iz različitih disciplina koji moraju da rade zajedno pa često dolazi do nesporazuma i nerazumevanja.

Tok procesa razvoja proizvoda u sistemskom inženjerstvu prolazi kroz sledeće faze:

- Definisanje zahteva
- Projektovanje sistema
- Razvoj podsistema
- Integracija sistema
- Instalacija sistema
- Evolucija sistema
- Rasturanje sistema

15. Model vodopada

Model vodopada nalaže razvoj softvera koji prolazi kroz faze strogo sekvencijalno, bez povratnih sprega i redosled tih faza tj. aktivnosti u razvoju softvera je strogo definisan. Kako bi se prešlo u sledeću fazu razvoja potrebno je da se prethodna faza završi i da svi njeni izlazni rezultati budu dostupni. Ovaj princip čini ovaj model jako nefeleksibilnim i nepogodnim u sličajevima kada je potrebno da postoji mogućnost izmene postojećih ili dodavanja novih zahteva u procesu razvoja jer faza analize i specifikacije predstavlja prvu fazu razvoja i zahteva se da svi njeni izlazni rezulatati, koji uključuju čitav spisak definisanih zahteva, budu potpuno definisani da bi uopšte moglo da se pređe na sledeću fazu razvoja. Ovaj model smatra se delimično zastarelim i koristi se samo u slučajevima kada se radi na razvoju izuzetno velikog sistema gde se razvoj odvija paralelno na više različitih lokacija pa je ovakav strogo planom vođen razvoj poželjan. Faze kroz koje prolazi razvoj po modelu vodopada su:

- Analiza i specifikacija zahteva
- Projektovanje sistema i softvera
- Implementacija i testiranje delova
- Integracija i testiranje sistema
- Eksploatacija i održavanje

16. Inkrementalni razvoj

Inkrementalni razvoj softvera podrazumeva razvoj koji kao ulazne zahteve ima samo okvirni opis krajnjeg proizvoda. Na osnovu njega se onda kreira neka inicijalna verzija softvera nakon čega se kružno vrši specifikacija, razvoj i validacija međuverzija softvera u inkrementima tako da se svaka sledeća međuverzija kompletnija, približnija finalnom proizvodu i u većoj meri zadovoljava realne potrebe zahtevaoca. Sa inkrementalnim razvojem se ciklično nastavlja sve dok se ne dođe do verzije softvera koja se smatra kompletnom i može se i finalno isporučiti. Prednosti ovakvog pristupa su što naručilac relativno brzo nakon početka razvija dobija funkcionalanu verziju softvera koju može da koristi dok ne stigne finalna verzija, takođe činjenica da naručilac može da dobije svu od međuverzija softvera na korišćenje omogućava veći stepen njegove uključenosti u sam razvoj i time omogućava kreiranje softvara koji je više „po njegovoj meri“ jer može da definiše nove ili izmeni postojeće zahteva zavisno od iskustva stečenog na interakciji sa nekom od međuverzija. Nedostaci su to što je proces razvoja dinamičan i promenljiv i samim tim nije praćen temeljnom dokumentacijom što čini proces razvoja manje vidljivim menadžerima i drugim osobama na upravljačkim pozicijama, takođe kako se struktura sistema relativno brzo definiše, a često se ne ulaže dovoljno novca u njeno refaktorisanje i preprakvu tokom samog razvoja to dovodi do toga da se struktura sistema svakim sledećim inkrementom usložnjava što je čini komplikovanom, neefikasnom i samim tim skupom za održavanje.

17. Razvoj zasnovan na korišćenju gotovih komponenti

Ovaj tip razvoja zasniva se na korišćenju već gotovih, ranije razvijenih komponenti ili komercijalno dostupnih komponenti. Tok razvoja je takav da se na osnovu navedenih zahteva analiziraju dostupne komponente, nakon što se odaberu komponente koje će biti korišćene vrši se prilagođavanje zahteva tako da odgovaraju sistemu koji se može kreirati na osnovu raspoloživih komponenti. Kako su komponente već razvijene i testirane razvoj po ovom modelu se zasniva na definisanju načina povezivanja i interakcije komponenata koje su uključene u sistem i testiranja tih veza. Faze procesa razvoja su: specifikacija zahteva, analiza gotovih komponenti, modifikacija zahteva, projektovanje sistema zasnovano na korišćenju gotovih komponenti, razvoj i integracija i validacija sistema.

18. Odnos faze i iteracije *(Odnosi se na RUP)*

Svaka faza može da sadrži više iteracija gde se rezultati onda razvijaju inkrementalno, a takođe možemo i ceo neki skup više faza da iteriramo po potrebi.

19. Navesti stavke agilnog manifesta

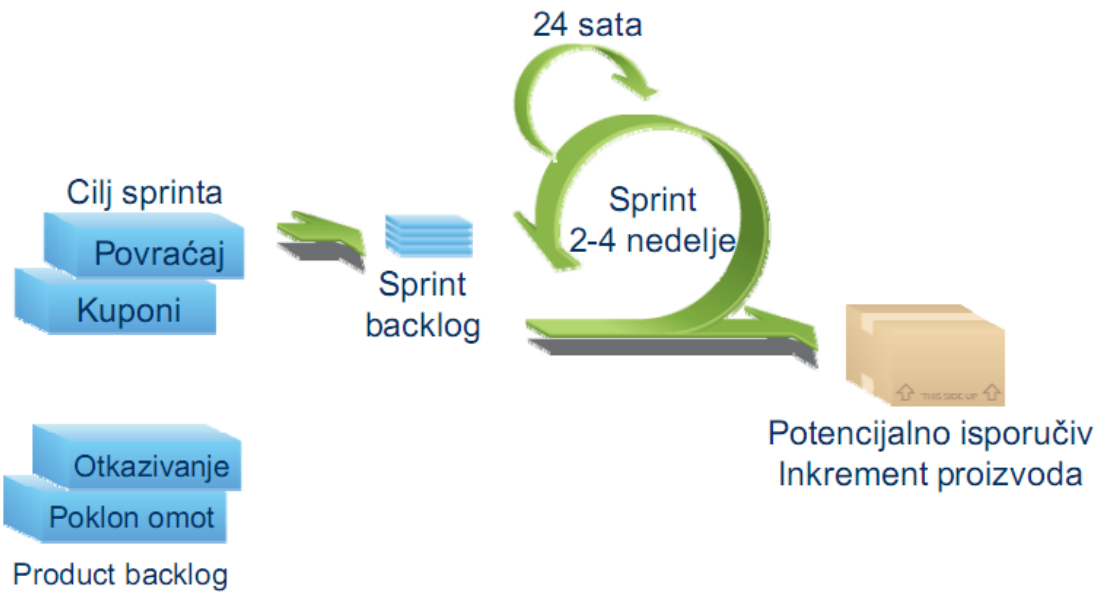
Agilni manifest je kratak dokument koji sadrži principe i ideje na koje se oslanja agilna metodologija za razvoj softvera. Taj manifest nalaže 4 glavne ideje:

- Više vrede pojedinci i interakcije nego procesni alati – u ovoj metodologiji znatno veći značaj se pridaje samim ljudima koji rade na razvoju i korisnicima, odnosno klijentima za koje se razvoj radi i njihovoh međusobnoj komunikaciji i interakciji radi što efikasnijeg postizanja cilja nego samim alatima i procesnim tehnikama koje se koriste u razvoju

- Više vredi programska podrška koja radi nego dokumentacija – U današnje vreme retko ko čita uputstvo koje dobije uz neki proizvod, tj. njegovu dokumentaciju, pogotovo kada su u pitanju softverski proizvodi, u tom slučaju korisnik se oslanja na intuitivnost i lakoću razumevanja samog korišćenja bez potrebe čitanja bilo kakve dokumentacije unapred. Ova stavka manifesta nalaže da je upravo iz tog razloga mnogo bitnije imati razvijenu i funkcionalnu programsku podršku kroz koju se korisnik može obratiti sa konkretnim problemom nego neki skup dokumentacije koju verovatno svakako niko nikada neće da pročita
- Više vredi saradnja sa klijentima nego ugovori – agilna metodologija razvoja promoviše razvoj koji je prilagođen promenljivosti zahteva tako da je u tom slučaju mnogo bitnije razviti dobar odnos sa klijentima i na taj način omogućiti da se kroz vreme i klijent i onaj ko proizvod razvija prilagode promenama u sistemu i dinamički odrede prioritete i funkcionalnosti u zavisnosti od trenutnog stanja sistema. Dok je sasvim suprotno tome, ugovor jedan statički dokument koji se sastavlja na samom početku saradnje i nije prilagođen dinamičkom karakteru sistema
- Više vredi odgovor na promene nego praćenje plana – povezano sa prethodnom stavkom, veći značaj ima podrška dinamici sistema nego strogo praćenje nekog plana koji je sastavljen u trenutku kada nisu bile dostupne sve informacije

20. Scrum proces

U Scrum metodologiji evidentira se tzv. Product backlog koji predstavlja spisak funkcionalnosti koje treba da se dodaju, odnosno opštije rečeno spisak nekih stvari koje treba da se naprave. Product backlog pravi se na osnovu potreba i zahteva naručioca tj. klijenta. Scrum proces sastoji se od sprintova i pre početka svakog sprinta određuje se koji je njegov cilj odnosno šta to želimo tim sprintom da postignemo. Na osnovu cilja sprinta se iz Product backlog-a bira skup stvari koji se prebacuje u sprint backlog. Sprint backlog predstavlja skup stvari koje treba napraviti u tom sprintu. Svaki sprint traje 2-4 nedelje i u okviru njega održavaju se sastanci na početku i na kraju. Potredno je u toku sprinta odraditi sve što je definisano sprint backlog-om. Po potrebi mogu da se dodaju kratki 24h sprintovi u okviru samog sprinta za rad na nekoj konkretnoj funkcionalnosti. Kao rezultat sprinta dobija se inkrement sistema koji je potencijalno isporučiv.



21. Sastanci u Scrum-u

U Scrum metodologiji postoji 4 tipa sastanaka. Pre početka sprinta organizuje se planiranje sprinta, kada sprint počne svakog dana postoji dnevni sastanak, nakon završetka sprinta organizuju se 2 tipa sastanka, pregled sprinta u kome se posmatra šta je urađeno u sprintu i retrospektiva sprinta koja služi za unapređenje procesa.

- **Planiranje sprinta** – Definiše se cilj sprinta i na osnovu njega biraju se stavke iz Product backlog-a na osnovu kojih se kreira sprint backlog. Mapiranje stavke iz Product backlog-a na stavku iz sprint backlog-a nije 1:1 jer kada smo već došli do trenutka da nešto moramo da implementiramo često se desi da nešto što smo smatrali za jednu stavku u Product backlogu predstavlja više različitih stavki u sprint backlogu. Kada se identifikuju zadaci sprinta svaki od njih se procenjuje po tome koliko će vremena da oduzme, ovo je potrebno zbog praćenja progresa sprinta. Ta procena se ne postiže tako što scrum master naloži neko procenu već svi učestvuju i daju sopstvene procene pa se na osnovu njih dogovaraju koncenzusom o nekoj finalnoj odluci o proceni svakog zadatka.
- **Dnevni sastanak** – Održava se svakog dana tokom trajanja sprinta, to je kratak sastanak koji traje otprilike 15 minuta i uslovno rečeno je „usputan“, njemu mogu da prisustvuju svi ali najviše razloga za pričanje imaju članovi razvojnog tima, scrum master i vlasnik proizvoda. Na ovom sastanku treba svako da kaže šta je radio juče, šta planira da radi danas i da li ima neke probleme koji mu stoje na putu. Ovo ne predstavlja neki raport scrum masteru već ima za cilj da se svi upoznaju sa time ko na čemu trenutno radi, šta je već urađeno i sa time da li ima nekih problema. Istovremeno, obzirom na to da se problem tako javno iznese, bilo ko ko ima ideju ili način da ga reši može da se javi i time se olakšava saradnja i pronalaženje adekvatne pomoći. Pomaže da se izbegnu ostali, nepotrebni sastanci.
- **Pregled sprinta** – Prezentuje se ono što je rađeno za vreme sprinta. Sadrži prezentaciju nove verzije proizvoda i funkcionalnosti koje su urađene, ali se ta prezentacija ne priprema mnogo. Postoji tzv. pravilo dvosatne pripreme, dakle kratko se pripremi i prezentuje ono što je urađeno bez nekih slajdova ili previše planiranja. U prezentaciji učestvuje ceo tim, a svi su pozvani da prisustvuju po želji.
- **Retrospektiva sprinta** – Predstavlja periodično razmatranje toga šta je dobro a šta ne. Posmatra se način na koji se rade određene stvari i to da li može nekako da se unapredi taj način. Ovaj sastanak ne traje dugo, obično 15-30 minuta i obično se organizuje preko svakog sprinta. Ceo tim učestvuje u ovom sastanku a mogu da se priključe i klijenti i ostali po želji. Cilj je unaprediti proces rada, možda nešto ukinuti, uvesti nešto novo ili izmeniti postojeće

22. Šta je RUP metodologija?

RUP je skup parcijalno uređenih koraka namenjenih osnovnom cilju - da se efikasno i u predviđenim okvirima korisniku isporuči sistem koji u potpunosti zadovoljava njegove potrebe. Proces proizvodnje softverskog proizvoda po RUP-a je:

- Inkrementalan i iterativan
- Planiran i kontrolisan
- Dokumentovan
- Baziran na UML-u

23. RUP karakteristike

RUP je inkrementalan i iterativan proces. To znači da se do konačne verzije sistema stiže kroz niz iteracija, a da se u svakoj iteraciji inkrementalno povećava funkcionalnost sistema sve dok se ne stigne do konačnog sistema. Na taj način, stalno se dobijaju izvršne verzije sistema sa sve većim brojem realizovanih funkcija, čime se tokom trajanja razvoja sistema, polako smanjuje rizik. Dobra osobina RUP metodologije je što je proces jako dobro dokumentovan (postoji i Web-tutor koji vodi korisnika kroz proces), dobro definisan - tačno je definisano šta se od proizvoda (modeli i dokumenti) u kojoj fazi dobija i u potpunosti podržan softverskim alatima (pre svega kompanija Rational (sada IBM) sa svojim softverskim alatima) i šablonima (templejtima) proizvoda. Sve ovo jako pomaže korisniku da dođe do konačnog cilja - što boljeg softverskog proizvoda.

24. 6 osnovnih principa za uspešan razvoj SW-a po RUP-u

- Iterativni razvoj SW-a – svaka od iteracija ima za rezultat izvršnu verziju SW-a
- Upravljanje zahtevima – Prevođenje zahteva korisnika u definisan skup potreba i funkcija sistema
- Komponentna arhitektura SW-a – poboljšava proširljivost i obezbeđuje višestruko korišćenje
- Vizuelno modeliranje - UML
- Kontinualna verifikacija kvaliteta
- Upravljanje promenama

25. RUP faze

1. Početna faza (Inception)
2. Faza razrade (Elaboration)
3. Faza izrade (Construction)
4. Faza isporuke (Transition)

Svaka faza može imati proizvoljan broj iteracija i svaka iteracija (osim, naravno, početne) treba da rezultira izvršnom verzijom koja se može testirati.

26. RUP proces

Napisati šta je RUP, koje faze sadrži i šta se u svakoj fazi radi.

27. Kontrolne tačke

Nakon svake od faza u RUP-u postoji po jedna kontrolna tačka. Nakon početne faze nalazi se predmetna kontrolna tačka, nakon faze razrade arhitekturna kontrolna tačka, nakon faze izrade kontrolna tačka početne funkcionalnosti i nakon faze iskoruke izvršna verzija proizvoda.

28. Početna faza

- Analiza problema
- Razumevanje potreba (potencijanih) korisnika
- Generalno definisanje sistema
- Upravljanje kod promena korisničkih zahteva

Rezultat ove faze je dokument Vizija sistema.

29. Vizija sistema

Piše se bez mnogo tehničkih detalja tako da bude razumljiva i korisnicima i razvojnom timu. Koriste se samo blok dijagrami za šematski prikaz sistema.

- Pozicioniranje proizvoda
- Opis korisnika
- Opis proizvoda
- Funkcionalni zahtevi
- Nefunkcionalni zahtevi
- Ograničenja
- Kvalitet

30. Faza elaboracije

- Izrada plana projekta
- Organizacija i ekipni rad
- Detaljna definicija zahteva
- Definisanje arhitekture sistem

Rezultati ove faze su:

- Plan projekta
- Use-case specifikacija
- Arhitekturni projekat sistema

31. Plan projekta

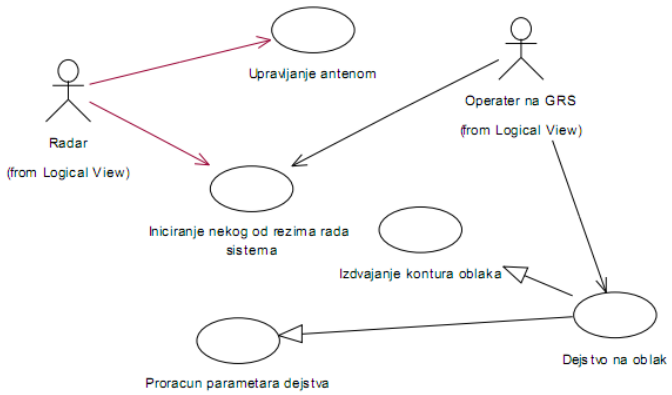
- Plan faza
- Plan izrade
- Rezultati projekta
- Kontrolne tačke (milestones)
- Resursi
- ...

32. Use case specifikacija

- Opis slučajeva korišćenja
- Definisanje aktera u sistemu
- Određivanje arhitekturno najznačajnijih slučajeva korišćenja

33. Opis slučajeva korišćenja

Opis slučajeva korišćenja može biti dat use-case dijagramom koji sadrži aktere, slučajeve korišćenja i relacije između njih:



Pored toga može biti dat tabelarno, u formatu koji sadrži naziv, aktere, svrhu algoritma, opis i redosled događaja pri čemu svaki od događaja može da sadrži akciju aktera ili reakciju sistema.

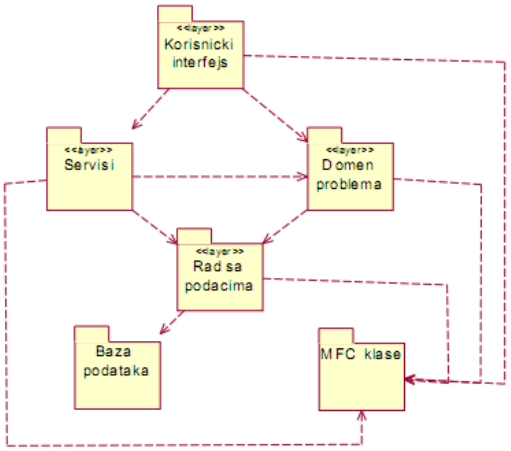
Naziv	Proračun elevacija za dejstvo
Akteri	Operater PRS1
Svrha algoritma	Proračun elevacija za lansiranje raketa
Opis	Nakon unosa novog sinoptičkog biltena u sistem potrebno je startovati izračunavanje novih elevacija za lansiranje raketa. Izračunavanje se vrši samo jednom nakon svake promene sinoptičkog biltena.

R. br. događaja	Akcija aktera	Reakcija sistema
1.	Ovaj slučaj korišćenja inicira operater PRS1 tako što izda komandu za izračunavanje elevacija.	
2.		Sistem daje informaciju o dužini trajanja proračuna a zatim vrši izračunavanje i na kraju izdaje poruku o završetku izračunavanja.

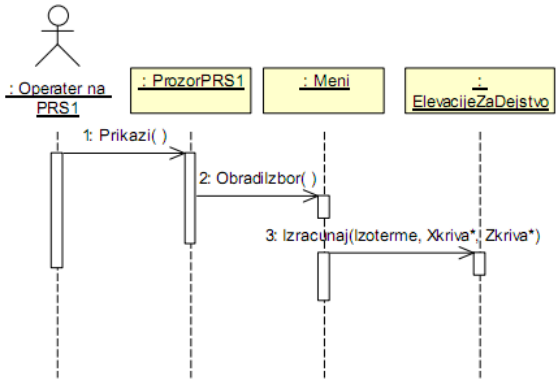
34. Arhitekturni projekat

- Definisanje arhitekture sistema
- Definisanje najbitnijih klasa
- Realizacija arhitekturno najznačajnijih slučajeva korišćenja
- UML dijagrami klasa

Arhitektura sistema opisuje se UML dijagramom:



Realizacija slučajeva korišćenja omogućuje se sekvencijalnim dijagramom:



35. Faza izrade

- Realizacija sistema
- Testiranje

Rezultati ove faze su:

- Plan testiranja
- Test specifikacija
- Detaljni projekat sistema
- Softverski proizvod

36. Plan testiranja

- Zahtevi testiranja
- Strategija testiranja
- Tehnike testiranja
- Alati za testiranje
- Resursi
- Proizvodi
- Kontrolne tačke
- ...

37. Test specifikacija

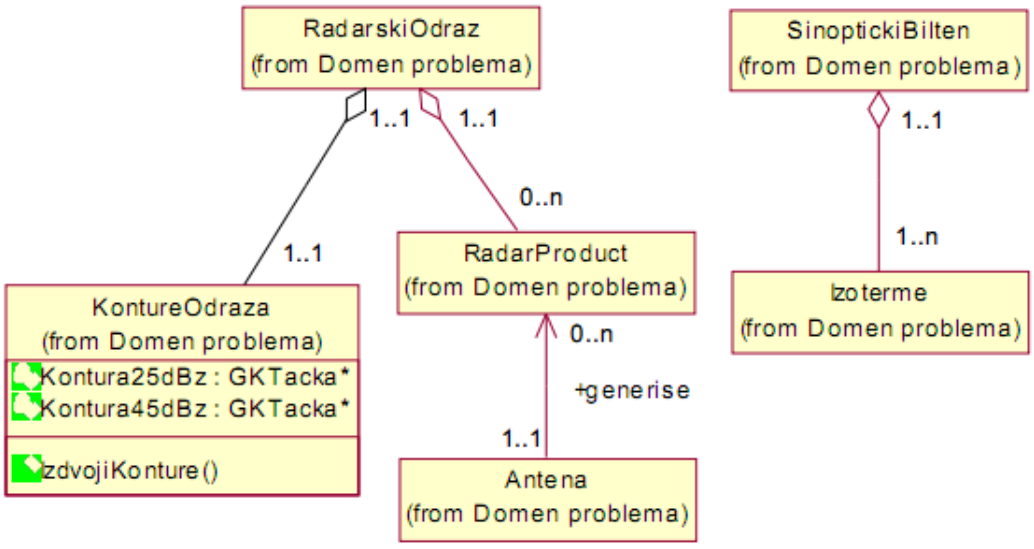
Sadrži test-case-ove. Jedan test case sadrži:

– Opis
– Radnje - ulaze
– Očekivane odzive - izlaze
– Završne radnje

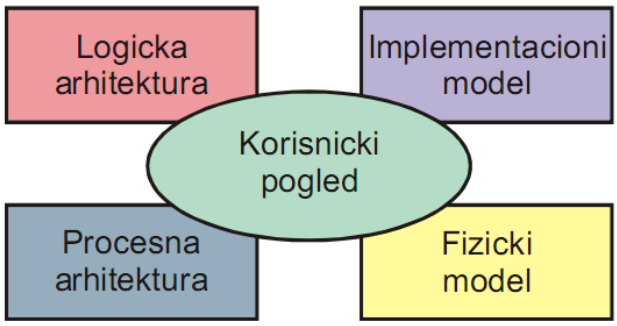
38. Detaljni projekat

- Arhitekturni projekat razvijen u detalje
- Dijagrami klasa
- “4+1” model sistema

Primer dijagrama klasa:



39. 4+1 model sistema (na slajdovima 36 do 42 su slike)



- **Logička arhitektura** sistema opisuje najvažnije klase u sistemu, njihovu organizaciju u pakete i podsisteme kao i organizaciju paketa i podsistema u nivoe (layers). Za predstavljanje logičke arhitekture se koriste dijagrami klasa i na osnovu tog dijagrama moguće je automatski generisati kod.
- **Procesna arhitektura** sistema opisuje najvažnije procese i niti (threads) u sistemu i njihovu organizaciju. Procesi se izvršavaju u nezavisnim adresnim prostorima računara, dok su niti procesi koji se izvršavaju paralelno sa procesima ili drugim nitima ali u adresnom prostoru nekog od procesa sa procesima ili drugim nitima ali u adresnom prostoru nekog od procesa. Za procesne arhitekture se koriste dijagrami klasa.
- Za prikaz **implementacionog modela** se koriste dijagrami komponenti.
- **Fizički model** opisuje fizičke čvorove u sistemu i njihov razmeštaj u prostoru. Za prikaz fizičkog modela koristi se dijagram razmeštaja.

40. Faza isporuke

- Finalizacija softverskog sistema
- Alfa (beta) testiranje
- Izrada korisničke dokumentacije (uputstva)
- Obuka korisnika
- Uvođenje sistema kod korisnika

Rezultatiove faze su:

- Test izveštaji
- Korisničko uputstvo
- Instalacija sistema

41. Test izveštaj

Pregled rezultata testiranja

– Generalna procena testiranog SW-a
– Uticaj test okruženja
– Predložena poboljšanja

Rezultati izvršenja test-case-ova – za svaki test-case navodi se njegova oznaka i da li je test prošao li ne, ukoliko nije navodi se težina greške od najteže – 1, do najlakše 4:

1 – Pad programa
2 – Nepravilan rad programa
3 – Neslaganje sa specifikacijom
4 – Neodgovarajući interfejs

Takođe se navodi i stepen lakoće korišćenja:
1 – Lako
2 – Intuitivno
3 – Uz striktno praćenje uputstva

42. Upravljanje zahtevima

Upravljanje zahtevima znači prevođenje zahteva korisnika u skup njihovih potreba i funkcija sistema. Ovaj skup se kasnije pretvara u detaljnu specifikaciju funkcionalnih i nefunkcionalnih zahteva. Detaljna specifikacija se prevodi u test procedure, projekat i korisničku dokumentaciju. Potrebno je definisati proceduru u slučaju promene zahteva korisnika.

43. Podela zahteva

Zahtevi mogu biti funkcionalni i nefunkcionalni. *(Detaljnije o oba tipa u naredna 2 pitanja)*

44. Funkcionalni zahtevi

- KO (Interfejs) koji ljudi, organizacije, ili sistemi smeju (ili ne smeju) da pristupe sistemu.
- ŠTA (Podaci) Informacije potrebne sistemu u smislu zadovoljenja funkcija.
- GDE (Mreža) Fizička ili Internet lokacija koja mora imati pristup rešenju.
- KADA (Događaji) Vremensko usaglašavanje poslovnih događaja.
- ZAŠTO (Businnes rules) Poslovna pravila kojima mora biti prilagođen sistem.
- KAKO (Proces) Koje zadatke ili procesiranje i funkcije sistem mora da izvrši.

45. Nefunkcionalni zahtevi

Ograničenja u projektu se moraju takođe definisati.
Kategorije ograničenja:

- Nivo kvaliteta
- Nivo cene
- Funkcionalnost
- Dizajn
- Politička
- Kulturološka
- Pravna

46. Uloga “Software Quality Assurance Specialist” stručnjaka

Svaki od funkcionalnih i nefunkcionalnih zahteva evoluira kroz različite fokuse i perspektive, definišući različite nivoe detalja u zahtevima. Svaka nenamerna praznina u zahtevima povećava RIZIK kraha projekta ili dodavanja drugog defekta. Uloga “Software Quality Assurance Specialist” stručnjaka je da identifikuje nedostajuće zahteve.

47. Šta omogućava korišćenje standarda u specifikaciji zahteva?

Omogućava KUPCU da postavi zahteve na način da ga isporučioc razume.
Omogućava ISPORUČIOCU da razume zahteve kupca i analizom zahteva bude siguran da je u mogućnosti da proizvede softverski proizvod odgovarajućeg kvaliteta.

48. Šta sve obuhvata specifikacija zahteva?

Organizacija mora da definiše zahteve kupca koji obuhvataju:

- Zahteve za proizvod koje je kupac definisao, uključujući i zahteve u pogledu dostupnosti, isporuke i podrške proizvodu
- Zahteve koje kupac nije definisao, ali koji su neophodni za planiranu ili željenu upotrebu
- Obaveze koje se odnose na proizvod, uključujući propise i zakonske zahteve

49. Šest preporuka "dobre prakse" za upravljanje zahtevima.

Upravljanje zahtevima je više od dokumentovanja istih:

1. Odvojite dovoljno vremena za proces definisanja zahteva.
2. Izaberite pravi pristup za iznošenje zahteva.
3. Prenesite – saopštite zahteve svim interesnim stranama.
4. Razvrstajte zahteve po prioritetu.
5. Zahtevi za ponovno korišćenje – reuse.
6. Pratite zahteve kroz životne cikluse softvera.

50. Proces upravljanja zahtevima

Proces upravljanja zahtevima sastoji se od međusobno nerazdvojivih podprocesa, koji se ne mogu zasebno i sekvencijalno odvijati:

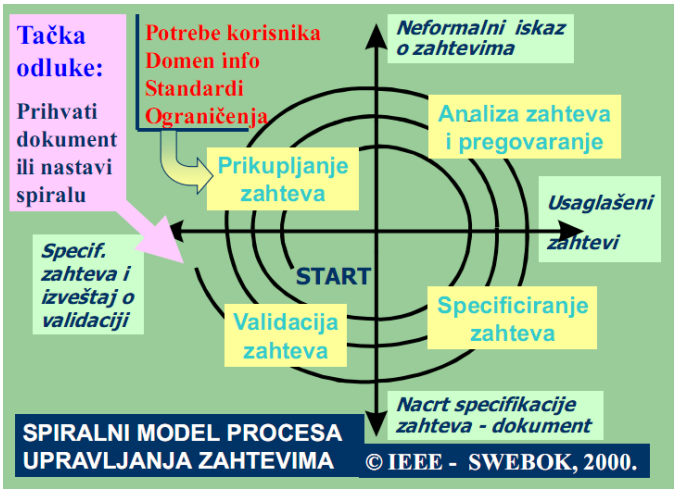
- Prikupljanje zahteva
- Analiza zahteva
- Specifikacija zahteva
- Validacija zahteva

51. Spiralni model procesa upravljanja zahtevima

Spiralni model procesa upravljanja zathevima podrazumeva kružni prolaz kroz 4 definisane faze upravljanja zahtevima. Svaki put kada se prođe pun krug potrebno je odlučiti da li smo zadovoljni kreiranim dokumentom specifikacije zahteva ili želimo da prođemo još jedan krug kroz sve 4 faze. Faze koje su sastavni članovi kruga su:

1. **Prikupljanje zahteva** – Sakupljaju se informacije o potrebama korisnika, informacije o domenu, o standardima i ograničenjima. Nakon ove faze postoji definisan neformalni iskaz o zahtevima
2. **Analiza zahteva i pregovaranje** – Vrši se analiza zahteva prikupljenih u prethodnoj fazi i pregovaranje sa korisnikom u cilju usaglašenja mišljenja i boljeg razumevanja. Kao rezultat ove faze potrebno je da izvršilac i korisnik imaju usaglašene zahteve
3. **Specificiranje zahteva** – Zahtevi usaglašeni u prethodnoj fazi se formalizuju. Kao rezultat ove faze imamo nacrt dokumenta specifikacije zahteva
4. **Validacija zahteva** – Zahtevi formalizovani u prethodnoj fazi se validiraju tj. proverava se da li odgovaraju očekivanjima korisnika. Kao izlaz iz ove faze imamo specifikaciju zahteva i izveštaj o validaciji

Nakon 4. faze potrebno je odlučiti da li se kreirani dokument prihvata ili se smatra da je potrebno makar još jednom proći kroz čitav niz faza 1. do 4.



52. Prikupljanje zahteva

Tri glavne kategorije učesnika su:

- Kupac
- Korisnik
- Developeri

NIKO od učesnika u ovoj fazi nema kompletnu sliku o softverskom proizvodu. Stoga je glavni problem neadekvatna komunikacija.

Tokom procesa prikupljanja zahteva mora se uspostaviti partnerski odnos među interesnim stranama. Učešće kupca je najkritičniji faktor od kojeg ovisi kvalitet softvera, i najteži problem je deljenje vizije o proizvodu sa kupcem. Ako interesne grupe ne komuniciraju efektivno, svaka od njih tražiće način da iskaže nadmoć i uticaj nad ostalima, što jeste cilj u politici, ali ne i u upravljanju zahtevima za proizvod.

53. Lista prava kupca SW-a

Pravo da:

1. Očekuje da analitičar govori njegovim jezikom
2. Očekuje da analitičar razume njegovo poslovanje i ciljeve, i u tom svetlu mesto i ulogu novog softverskog proizvoda
3. Očekuje da analitičar informacije koje mu on prezentuje evidentira, razradi i izradi specifikaciju zahteva za sistem
4. Analitičar prezentuje ideje i alternative i za njegove zahteve i za implementaciju
5. Da mu projektanti objasne zahteve za softver na nivou podsistema i nižim nivoima
6. Očekuje da ga projektanti tretiraju sa poštovanjem i da održavaju klimu saradnje i profesionalnosti
7. Da mu se opišu karakteristike koje će obezbediti lako korišćenje proizvoda i da se opišu scenariji upotrebe proizvoda
8. Budu prikazane mogućnosti da se već postojeće softverske komponente uključe ili prilagode njegovim zahtevima
9. Mu se ukaže poverenje na njegovu procenu cene i uticaja na softver kada zahteva promenu zahteva za softver
10. Primi sistem koji zadovoljava zahteve za funkcionalnost i kvalitet, do nivoa koji je utvrđen u komunikaciji sa projektantima, usaglašen i dokumentovan u specifikaciji i dopunama specifikacije

54. Lista odgovornosti kupca SW-a

Kupac softvera ima odgovornost da:

1. Upozna analitičara sa svojim poslovanjem i definiše žargon.
2. Potroši vreme da obezbedi zahteve i pojasni ih u svetlu svojih poslovnih potreba.
3. Bude precizan u vezi sa zahtevima za sistem.
4. Blagovremeno donese odluke u vezi zahteva, kada se to zahteva od njega.
5. Definiše prioritete za svaki pojedinačni zahtev, osobinu sistema i scenario primene.
6. Izvrši preispitivanje dokumenata (zahteva) i prototipova.
7. Odmah dostavi promene u zahtevima za proizvod.
8. Postupa saglasno dokumentovanom procesu promena zahteva razvojne organizacije.
9. Poštuje proces upravljanja zahtevima koji koristi razvojna organizacija.
10. Poštuje projektantske ocene u vezi cene i izvodljivosti zahteva.

55. Kategorije zahteva

- **Funkcije**: "šta" sistem mora da bude sposoban da uradi.
- **Osobine**: "koliko dobro" će funkcije biti izvršavane.
- **Cena**: koliko će koštati (bilo koji ulazni resurs: novac, ljudi, ili vreme) kreiranje i održavanje funkcija i njihovih osobina.
- **Ograničenja**: bilo koja restrikcija u slobodi definisanja zahteva ili dizajnu.

56. Tipovi zahteva

Zahtev može da bude kvantifikovan na nekoj mernoj skali i može da ne bude kvantifikovan ali da testiranjem može da se utvrdi da li je realizovan ili ne.

Funkcije se ne mogu kvantifikovati, onu ili prisutne ili nisu prisutne. Sve osobine i cena se mogu i moraju kvantifikovati. Ograničenja mogu biti bilo kvantifikovana bilo nekvantifikovana.

57. Karakteristike zahteva

1. Svi zahtevi su testabilni na prisustvo u realnom svetu njihove implementacije.
2. Svi zahtevi reflektuju nečije subjektivne vrednosti i prioritete.
3. Za bilo koji set zahteva, postoji neki drugi set zahteva koji verovatno isto tako dobro ili bolje zadovoljava realne potrebe, vrednosti i prioritete kupca.
4. Svi zahtevi su u prirodnom konfliktu sa svim ostalim u odnosu na resurse.
5. Zahtevi nisu statični, nego se stalno menjaju kako se menja svet menjajući naše potrebe, vrednosti i prioritete.
6. Upravljanje zahtevima je sistematičan proces utvrđivanja kompletnog seta relevantnih vrednosti kojima raspolažu interesne strane, i procesiranja istih sve dok se od njih ne dobije zadovoljavajući nivo "isporuke zahtevanih krajnjih stanja". To implicira da se mora uključiti dizajn, testiranje, kontrola kvaliteta, upravljanje projektom, specifikacija jezika i sve ostale relevantne discipline da se omogući da projekat uspe.

58. Lažni zahtevi

1. Zahtevi nisu iskazani na kvantifikovan i merljiv način:
   - "redukovana cena"
   - "poboljšana kontrola upravljanja"
   - "visoka upotrebljivost"
   - "povećano zadovoljstvo kupca"

2. Ideje o projektovanju umesto stvarnih zahteva.
    - “zahteva se objektno orjentisana baza podataka”
    - “potrebno je poboljšano uputstvo za rukovanje”
    - “obezbediti Windows XP interfejs”

59. Razlozi uspeha softverskih projekata

1. Učešće korisnika
2. Podrška izvršnog menadžmenta
3. Jasni zahtevi
4. Odgovarajuće planiranje
5. Realistična očekivanja
6. Smaller Milestones
7. Kompetentno osoblje
8. Vlasništvo (Ownership)
9. Jasna vizija i ciljevi
10. Hard-Working Staff

60. Razlozi neuspeha softverskih projekata

1. Nekompletni zahtevi
2. Nedostatak učešća korisnika
3. Nedostatak resursa
4. Nerealna očekvanja
5. Nedostatak podrške rukovodstva
6. Promenljivi zahtevi
7. Nedostatak planiranja
8. Nije više potreban
9. Nedostatak IT menadžmenta
10. Tehnološka nepismenost

61. Šta su metode inženjeringa zahteva?

Procesi inženjeringa zahteva su obično vođeni odgovarajućom metodom. Metode inženjeringa zahteva definišu sistematske načine za dobijanje modela sistema.

62. Koje su osnovne karakteristike i po čemu se razlikuju metode za inženjeringe zahteva?

- Pogodnost za usaglašavanje sa krajnjim korisnikom
- Preciznost definicije i odgovarajuće notacije
- Pomoć kod formulisanja zahteva
- Fleksibilnost
- Podržanost odgovarajućim SW alatima

63. Koje su najpoznatije metode za inženjering zahteva?

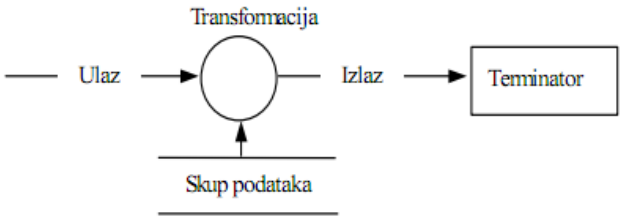
- Metoda praćenja toka podataka (Data Flow diagram)
- Metoda koja koristi relacioni model podataka
- Objektno orijentisane metode
- Formalne metode
- Metode zasnovane na ponašanju sistema (Use-case specifikacija i viewpoint orijentisane metode)

64. Metoda praćenja toka podataka (Data Flow diagram)

Prva metoda koja se koristila za prikupljanje zahteva. Tok podataka je najbitniji u ovoj metodi. Osnovni elementi:

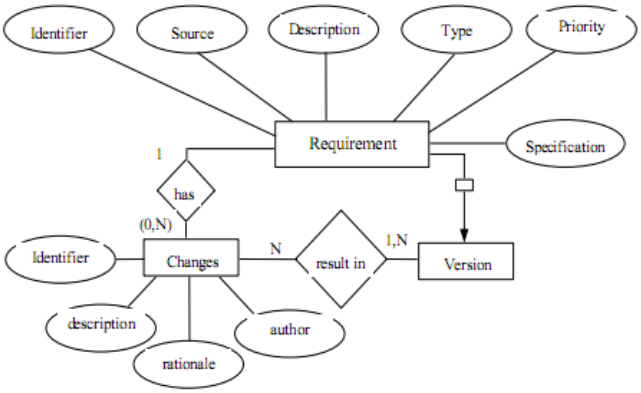
- ulazni tok podataka – uglavnom se na (tj. ispod nje, iznad nje ili na njoj) ulaznoj strelici piše koji je tok podataka u pitanju
- krug – neki proces, obrada, transformacija podataka, na krugu ili ispod njega ide ime obrade
- izlazni tok podataka sa svojim nazivom
- terminator – pravougaonik – objekat koji bismo prepoznali u realnom svetu za domen za koji pravimo specifikaciju
- Skup podataka – kada god imamo neke fajlove, bazu, ... gde čuvamo podatke (na ili ispod piše koji je tip skupa podataka i tako se zove)

+ može da bude dat DFD dijagram i da mi treba da opišemo šta sistem radi na osnovu toga što pročitamo sa njega



65. Metoda koja koristi relacioni model podataka

Koristi za opis zahteva prošireni ER model podataka. Za svaki zahtev se pamti entitet i za njega atributi. Svaki zahtev će verovatno imati mogunosti da bude menjan pa se zahtev vezuje za promenu koja mora da ima svoje atribute. To omogućava da se vidi cela istorija izmena jednog zahteva i da se lako vrši rollback na određenu verziju.



66. Objektno orijentisane metode

Nasledile ER model u specifikaciji zahteva. Osnovno je prepoznati objekte iz realnog sveta, veze među njima, metode, atribute i servise. Ako se koristi OO metoda prepliće se specifikacija zahteva već sa nekom fazom projektovanja što je u saglasnosti sa RUP metodologijom. Objekti se traže među uređajima sa kojima sistem interaguje, sistemima koji sarađuju sa sistemom koji se razvija, ogranizacionim jedinicama, stvarima o kojima moraju da se čuvaju podaci, fizičkim lokacijama, i specifičnim ulogama ljudi u celom tom procesu koji obrađujemo. Koraci pri korišćenju ove metode u specifikaciji:

- Identifikacija osnovnih objekata i klasa
- Definisanje objektne strukture i veza između klasa
- Definisanje atributa i servisa objekta
- Definisanje poruka koje objekti razmenjuju

Time se na kraju dobiju već potencijalni objekti koji se posle koriste u samoj fazi projektovanja sistema.

Ako ćemo da poštujemo baš OO pristup onda ćemo za svaki atribut da imamo 2 servisa – get i set servise. Ali treba da prepoznamo i druge servise najčešće u vezama tj. načinima na koji objekti mogu da interaguju.

67. Metode zasnovane na ponašanju sistema

Postoje dve podvrste:

- Use-case specifikacija i scenariji ponašanja – RUP preporučuje korišćenje ove metode
- Viewpoint orijentisane metode (metode orijentisane na različite poglede na sistem)

1. *Use-case metoda*

   Sistem se posmatra sa stanovišta korisnika sistema. Opisuju se različiti slučajevi korišćenja sistema koji su sada bazirani na raznim vrstama scenarija onoga kako će taj sistem da se koristi u realnom svetu. Koristi se UML notacija. Svaki use-case je baziran na scenariju upotrebe sistema. Jedan scenario je tipično opis interakcije korisnika i samog sistema. To su primeri kako će sistem biti realno korišćen u životu. Jedan scenario mora da sadrži opis početne situacije, opis normalnog toka događaja, opis izuzetaka (slučajeva kada se izlazi iz normalnog toka), informacije o konkurentnim aktivnostima i opis stanja gde se scenario završava. Jedan use-case je jedan posao koji može da se obavi u sistemu bilo da smo ga mi inicirali kao korisnik ili je to neka sistemska stvar, tj. desilo se nešto u sistemu (npr. otkucao neki tajmer), takav use-case mora da ima svoj kraj kada može da se startuje nešto drugo, neki durgi use-case. Dakle, to je posao koji ima prepoznatljiv početak i kraj. Ukoliko prepoznamo sve moguće slučjeve korišćenja aplikacije i detljano ih opišemo ovim dijagramima mi onda dobijamo detaljne specifikacije zahteva koje onda možemo da implementiramo. (U use-case <<uses>> je isto što i <<includes>>). Use-case specifikacija je izlaz iz faze razrade i sadrži opis slučajeva korišćenja, definisanje aktera u sistemu i određivanje arhitekturno najznačajnijih slučajeva korišćenja, ovo poslednje je bitno zato što one slučajeve koje prepoznamo kao najznačajnije možemo dodatno da specificiramo što se najčešće radi dodavanjem senkvencnih dijagrama gde dodatno opisujemo objekte vezane za taj posao, poruke koje se tu razmenjuju, ... Svaki od use-case-eva treba da bude propraćen sa dve tabele, jedna je opšta tabela opisa tog use-case-, a druga tabela je neki redosled događaja gde su opisane akcije aktera i to kako bi taj sistem trebalo da reaguje na te akcije aktera.

   Naziv use case-a, kratak opis, preduslovi, opis (tok), izuzeci, posledice.

   Treba da se zna šta je use-case i use-case specifikacija, šta su scenarija, kako izgledaju UML oznake za aktera i use-case, da ova specifikacija predstavlja deo korisničkog pogleda u RUP 4 + 1 modelu sistema. Može da nam bude dat neki mali problem a da mi napravimo specifikaciju kao dijagram koji je dodatno objašnjen ili tabelom ili sekvencijalnim dijagramom (Može čak da bude i neki primer sa časa)

2. *Viewpoint orijentisane metode*

   Osnovna ideja je da se sistem posmatra sa različitih tačaka (aspekata) posmatranja tj. iz ugla različitih korinika sistema. Kada projektujemo neki veliki sistem neće svi ljudi koji ga koriste imati isti pristup tom sistemu (npr. radnik u marketingu i radnik u proizvodnji ili direktor neće imati isti pogled), svaki od njih će imati neki skup funkcionalnosti koji bi voleo da mu sistem pruža. Na nama kao projektantu je prepoznamo sve te viewpoint-ove i da se stavimo u svaku tu ulogu i da sagledamo skup funkcionalnosti koje sistem treba da ima za svakog od njih. Prednosti ove metode su to što eksplicitno otkriva sve izvore zahteva za sistem i vrši organizovanje i struktuiranje tih zahteva po viewpoint-ovima, ako za sve viewpoint-ove sagledamo sve zahteve onda imamo osećaj kompletnosti tj. da smo prepoznali baš sve zahteve. Postoje različiti tipovi viewpoint-ova:

   - Direktni aspekti posmatranja – ljudi ili drugi sistemi koji imaju direktnu interakciju sa sistemom (kod bankomata korisnici bankomata i baza podataka sa računima)
   - Indirektni aspekti posmatranja – korisnici koji ne koriste sistem direktno ali imaju uticaja na zahteve (kod bankomata obezbeđenje)
   - Domenski aspekti posmatranja – ograničenja i karakteristike domena koje imaju uticaja na zahteve (kod bankomata standardi za komunikaciju između banaka)

   Moguće viewpoint-ove treba tražiti među:

   - Pružaocima i korisnicima servisa sistema
   - Sistemima sa kojima novi sistem treba da interaguje
   - Propisima i standardnima
   - Izvorima poslovnih i nefunkcionalnih zahteva
   - Inženjerima koji će razvijati i održavati sistem
   - Marketinškim i ostalim poslovnim aspektima

   Šta je osnovna ideja, koji su tipovi i šta sve spada u viewpoint. Takođe može da nam bude dat neki mali sistem a da mi prepoznamo sve viewpoint-ove za njega.

68. Šta je projektovanje softvera?

Projektovanje softvera je faza u životnom ciklusu razvoja softvera koja sledi nakon faze specifikacije zahteva. Izlazi iz faze specifikacije zahteva predstavljaju ulaz u fazu projektovanja softvera. Daje odgovor na pitanje **KAKO** realizovati delove sistema koji će ispuniti zahteve definisane specifikacijom. Deli se u dve podfaze projektovanja softvera - *arhitekturno projektovanje* i *detaljno projektovanje.* U fazi arhitekturnog projektovanja se definiše arhitektura sistema, u smislu celina i komponenata i njihovih veza. U detaljnom projektovanju razmatra se realizacija sistema do najsitnijih detalja, kao što su sami algoritmi i strukture podataka koje se koriste. *(NAPOMENA – ove opšte karakteristike podfaza bi trebalo navesti ako se padne 29. pitanje; postoje posebna pitanja za obe podfaze gde je potrebno detaljnije ih opisati; može).*

69. Koje faze sadrži projektovanje softvera?

Sadrži arhitekturno i detaljno projektovanje. U fazi arhitekturnog projektovanju se definiše arhitektura sistema, u smislu celina i komponenata i njihovih veza. U detaljnom projektovanju razmatra se realizacija sistema do najsitnijih detalja, kao što su sami algoritmi i strukture podataka koje se koriste

70. Šta je arhitekturno projektovanje softvera?

To je projektovanje koje podrazumeva definisanje arhitekture sistema. Tu arhitekturu čine međusobno zavisni softverski moduli i komponente, a u arhitekturnom projektu se takođe moraju navesti i vidljive osobine modula i komponenata, kao i specifikacija njihovih međusobnih veza. Nakon što se uoče zavisnosti između ovih celina (podsistema), neophodno je definisati interfejse svakog od njih. Iterfejs podsistema predstavlja skup funkcija iz jednog podsistema koje će biti vidljive ostalim podsistemima, kao i parametri koji moraju biti preneti pri pozivu tih funkcija.

Vidljive osobine modula i komponenata su:

a. Skup funkcionalnosti softvera
b. Korišćenje deljivih resursa
c. Performanse
d. Upavljanje izuzecima

71. Šta je detaljno projektovanje softvera?

Detaljno projektovanje sledi nakon faze arhitekturnog projektovanja. U ovoj fazi, neophodno je raščlaniti na detalje svaku celinu uočenu pri arhitekturnom projektovanju i razmotriti na koji će način biti realizovana. Dakle, treba raščlaniti podsisteme iz arhitekturne faze. Ukoliko se koristi klasična metoda projektovanja, podsistem se deli na module, a ukoliko se koristi objektno orijentisani pristup, detaljnim projektovanjem dobijaju se klase. Takođe, sada je neophodno definisati interfejse za komunikaciju između manjih celina dobijenih detaljnim projektovanjem. Kod klasa, ti interfejsi su servisi koje klase pružaju ostalim klasama, kao i parametri tih servisa.

Detaljno projektovanje podrazumeva:

a. Projektovanje struktura podataka i baza podataka – detaljno razmatranje potrebnih struktura i projektovanje kompletne šeme baze podataka
b. Projektovanje algoritama za funkcije i module – definisanje algoritama kojima će se realizovati konkretne funkcije softvera; moguće je uočiti koje funkcije će nam biti potrebne i u arhitekturnom projektovanju, ali u ovoj fazi je neophodno tačno specificirati kako će ta funkcija biti realizovana i koje će algoritme koristiti, kako bi se olakšao posao programerima
c. Projektovanje korisničkog interfejsa za interakciju korisnika sa softverskim sistemom – korisnički interfejs je deo sistema sa kojim interaguje korisnik, i od njegove strukture i izgleda zavisi kako će korisnik doživeti softver

72. Osobine dobrog projekta.

Hijerarhija – softver bi trebalo da bude organizovan u dobru hijerarhiju komponenata. Nije pogodno da postoji mnogo komponenata sa previše veza koje nije moguće ispratiti.

Modularnost – sistem treba da bude dekomponovan u posebne celine (module, podsisteme) sa jasno definisanim interfejsima, tako da svaka celina zna šta može koristiti iz drugih. Primer dekompozicije može biti podela u posebnu celinu koja sadrži podatke, i celinu koja te podatke obrađuje.

Nezavisnost – pri dekompoziciji, treba grupisati slične stvari u jednu isti modul. Na ovaj način, ako se menja nešto bitno, izmene će biti lokalizovane i vršiće se u zatvorenim modulima. Takođe, ako je modul dobro definisan i ako su slične stvari lepo grupisane, biće ih lako ugraditi i u neke druge sisteme.

Jednostavan interfejs – korisnički interfejs treba biti jednostavan, da ne bi zbunio i odbio korisnika. Treba izvegavati komplikovan interfejs, interfejs sa velikim brojem mogućnosti i interfejse čija izmena dovodi do neželjenih efekata.

73. U kojim fazama RUP-a se vrši koja vrsta projektovanja *(u kojoj arhitekturno, a u kojoj detaljno)?*

Arhitekturno projektovanje se sprovodi u fazi elaboracije i podrazumeva *definisanje arhitekture sistema, definisanje najbitnijih klasa i realizaciju arhitekturno najznačajnijih slučajeva korišćenja.* Koriste se UML dijagrami klasa, a kao rezultat se dobija dokumenta *Arhitekturni projekat sistema.*

Detaljno projektovanje se sprovodi u fazi izrade i podrazumeva *arhitekturni projekat razvijen u detalje, detaljne dijagrame klasa i „4+1“ model sistema.* Rezultata ove faze je dokument *Detaljni projekat sistema.*

74. „4 + 1“ model sistema *(pitanje iz oblasti RUP-a, jedno od najčešćih pitanja na ispitu).*

Sistem se posmatra iz 4 glavna ugla, i jednog korisničkog.

Korisnički pogled ne utiče na stvari kao što su projektovanja klasa, modula i tome slično. Za ovaj pogled vezano je definisanje Use Case-ova sistema, odnosno sastoji se iz kompletnog opisa toga šta će sistem konkretno da radi i na koji način (šta radi korisnik i koji odgovor daje sistem na njegove akcije). Ovaj pogled predstavlja dobru specifikaciju za projektovanje.

Logička arhitektura je najbitnija u ovom modelu, jer se u njoj daje kompletan dijagram klasa sistema koji će biti realizovan, pa se ova arhitektura direktno reflektuje na kod. Za ovu arhitekturu se koriste se UML dijagrami klasa.

Procesna arhitektura je arhitektura u kojoj se projektuje sistem tako da su vidljivi svi procesi i sve niti koje postoje u toku izvršenja sistema. Koristi se takođe UML dijagram klasa, s tim što se klasi dodaju stereotipi <<process>> ili <<thread>>. Ovde se prikazuje koji procesi i niti postoje, šta se izvršava u paraleli i kako komuniciraju.

Implementacioni model koristi dijagram komponenti i predstavlja fizičke komponente u smislu posebnih fajlova koji čine sistem (dll, exe, konfiguracioni fajlovi). Treba navesti i veze između komponenata.

Fizički model koristi deployment dijagrame UML-a. Kockama se predstavljaju procesni elementi, a linijama se navode veze određenog tipa (LAN, internet). Kao fizički čvorovi u ovom dijagramu javljaju se svi računari sistema, ali i ostale fizičke komponente. Za manje sisteme, moguće je kombinovati fizički i implementacioni model, tako što se u kocku procesnog elementa fizičkog modela ubaci dijagram komponenata koje se fizički nalaze na računaru na koji se ta kocka odnosi.

75. Metode projektovanja softvera

Tri osnovne grupe metoda za projektovanje softvera su *strukturne metode, objektno orijentisane metode* i *kombinovane metode. (Nije naglasio, ali pretpostavljam da može da se padne posebno pitanje za svaku od ove tri metode).*

a. **Strukturne metode** – dominantan način pre pojave OO pristupa. Sistem se posmatra sa funkcionalnog stanovišta, odnosno kao skup funkcija koje se obavljaju nad podacima, gde se podacima predstavlja stanje sistema. Funkcije su se po srodnosti grupišu u module, a oni zatim čine sistem. Za predstavljanje strukturnh metoda se često koriste CASE alati i grafičke notacije.
b. **Objektno orijentisane metode** – sistem čine objekti u interakciji, gde stanje objekta čine njegovi podaci (atributi) kojima se upravlja korišćenjem metoda klasa kojima objekti pripadaju. Objekti komuniciraju razmenama poruka, odnosno međusobnim pozivima metoda. Za ove metode koristi se UML grafička notacija.
c. **Kombinovane metode projektovanja** – koriste dobre elemente obe prethodno pomenute grupe. Često se nazivaju „hibridne metode“.

76. Tehnike projektovanja softvera

Dva osnovna pristupa su *top-down* i *bottom-up* projektovanje. Kod top-down metode, krećemo se od vrha sistema i silazimo naniže (najviši sloj je korisnički interfejs, najniži sloj čine podsistemi). Kod bottom-up, kretanje je u suprotnom smeru, od podsistema ka interfejsu.

77. Top-down projektovanje softvera

Polazi se od vrha sistema, od njegovih gornjih nivoa i ide se do hijerarhijski najnižijih nivoa sistema. Najniži nivo predstavljaju podsistemi.

Loša strana ovog pristupa je to što se forsira razvoj pojedinih grana sistema, dok druge nisu ni započete. Još, ne sagledava na pravi način postojeće komponente koje se mogu iskoristiti, jer su one najčešće na najnižim nivoima.

Dobra strana ovog pristupa je to što se kreće sa vrha. Na najvišim slojevima je korisnički interfejs, odnosno jedini slojevi sa kojima korisnik direktno dolazi u kontakt. Zbog toga, u ranim fazama mogu da se razreše nedoumice sa korisnikom, jer on rano dobija prototip i inicijalni izlged sistema, na koji može da uloži primedbu. Niže komponente još uvek nisu ubačene u prototip, jer se do njih još uvek nije stiglo u projektovanu. Zbog toga se prave stub-ovi koji predstavljaju zamene za module projektovane tako da imaju isti interfejs kao modul koji menjaju, ali u pozadini funkcije stub-ova ne rade ono za šta je modul namenjen, već samo pružaju mogućnost poziva.

78. Bottom-up projektovanje softvera

Kreće se od najnižih nivoa sistema i kreće se naviše. Koristi se kada ima dosta gotovih komponenata koje se mogu uključiti u sistem.

Dobra strana je što se mogu na višim nivoima koristiti gotove komponente sa nižih nivoa.

Loša strana je što se vrh sistema projektuje na kraju, tako da se poslednji projektuje korisnički interfejs i korisnik se kasno uključuje u ceo razvoj sistema.

Na najnižim nivoima kreiraju se moduli. S obzirom na to da još uvek nemamo projektovane više nivoe koji bi pozvali funkcije nižih modula, kreiraju se takozvani drajveri, koji predstavljaju zamenu za više module i vrše pozive funkcija nižih modula. Drajver nema funkcionalnosti pravog modula čiju zamenu vrši. Kako se penjemo, jedan po jedan drajver menja se pravim modulima. Zamena drajvera je bezbolna, jer ne menja ni na koji način niže, već postojeće module.

79. Osnovni arhitekturni modeli:

- Repository (Skladište)
- Pipe and Filter (Protočna obrada)
- OO model
- Client/Server model
- Slojeviti model (Layered)
- Event-driven model (Implicitno pozivanje)
- Control model

80. Kada se upotrebljava, šematski prikaz i šta su dobre, a šta loše osobine:

Pitanje može glasiti i - dat je sistem koji treba da projektujemo i da pitanje bude šta bismo od arhitekturnih modela izabrali za realizaciju takvog sistema.

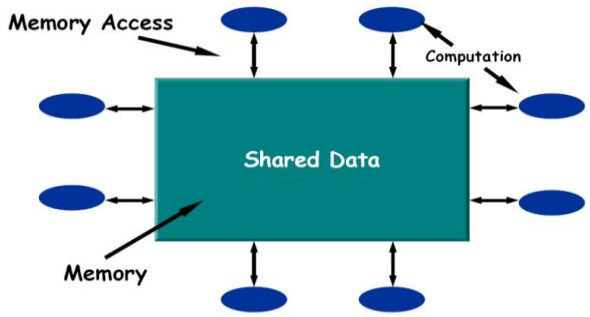
80.1 Repository (Skladište)

Ovaj model se koristi kod sistema kod kojih je neophodno deljenje velikih količina podataka. Tada se organizuje centralno skladište podataka kome pristupaju podsistemi.
Komponente:

- Centralna baza podataka
- Skup SW komponenata koje pristupaju centralnoj bazi.

Konektori:

- Pozivi procedura
- Direktan pristup bazi podataka.



Prednosti:

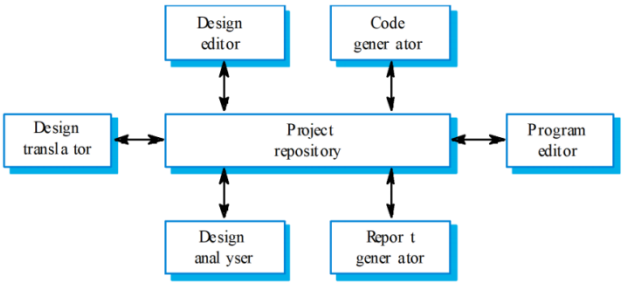
- Efikasan način za deljenje velike količine podataka
- Podsistemi ne moraju da brinu o nekim aspektima upravljanja podacima kao što su backup, sigurnost, ...
- Model deljenja podataka je prikazan u obliku šeme skladišta podataka.

Nedostaci:

- Podsistemi moraju da se slože oko skladišta podataka. Kompromis je neizbežan
- Evolucija podataka je skupa
- Otežana je distribucija podataka

Primeri:

- Informacioni sistemi
- Okruženja za razvoj SW-a
- Grafički editori
- Baze znanja u AI
- Sistemi za reverse engineering



80.2 Pipe and Filter model (protočna obrada)

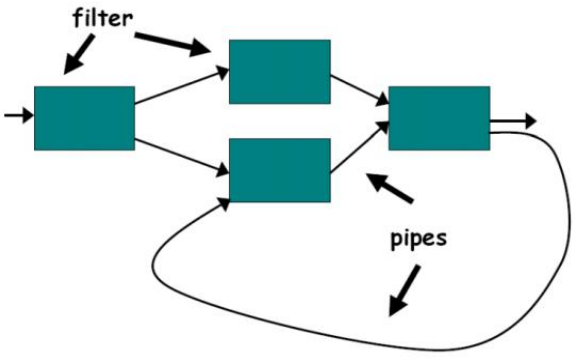
Ovaj model se koristi kod sistema kod kojih je neophodno izvršiti unapred definisane serije nezavisnih obrada nad podacima.

Komponente prihvataju tokove podataka na ulazu i generišu tokove podataka na izlazu.
Komponente:

- Komponente se često zovu i filteri koji vrše određene transformacije ulaznih podataka.

Konektori:

- Konektori se često zovu pipe (cevovod) jer povezuju tokove i filtre (izlaz jednog filtra vode na ulaz drugog filtra).



Prednosti:

- Lako razumevanje ulazno-izlaznog ponašanja celokupnog sistema kao skupa individualnih ponašanja filtera u sistemu
- Lako je ponovno korišćenje filtera (reuse), jer je između dva filtra već definisan format razmene podataka.
- Laka je izmena ili proširenje sistema (zamenom postojećih ili dodavanjem novih filtera).
- Prirodno je podržano konkurentno izvršenje.

Nedostaci:

- Nije najbolji izbor kod interaktivnih sistema zbog velikog broja transformacija
- Povećava kompleksnost sistema i smanjuje efikasnost;

Primeri:

- Unix shell skriptovi – cat file | grep Erroll | wc -| (Unix pipeline)
- Tradicionalni kompajleri (prevodioci) – lexical analysis + parsing + semantic analysis + code generation

80.3 OO model

Ovaj model se koristi kod sistema kod kojih je neophodno izvršiti zaštitu i enkapsulaciju podataka.

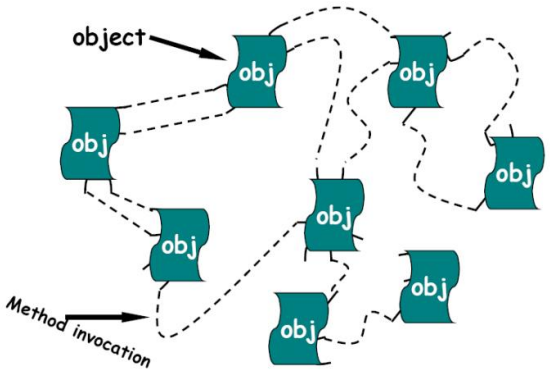
Podaci i pridružene operacije su enkapsulirane u apstraktne tipove podataka (klase, objekte).

Komponente:

- Objekti

Konektori:

- Metodi (servisi) objekata.

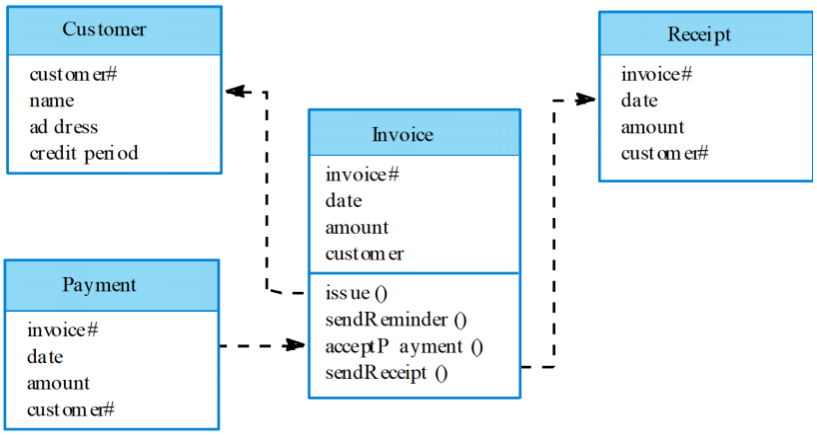


Prednosti:

- Zbog skrivanja informacija od klijenata, moguće je promeniti implementaciju objekata bez uticaja na klijente
- Moguće je projektovanje sistema kao skupa autonomnih agenata
- Moguće je direktno mapiranje entiteta iz realnog sveta u objekte

Nedostaci:

- Neophodno je poznavanje indentiteta objekata. Ukoliko se izmeni identitet nekog objekta, moraju se izvršiti izmene i kod svih objekata koji ga pozivaju
- Mogućnost pojave "bočnih efekata"



80.4 Client-Server Model

Ovaj model se koristi kod distribuiranih sistema.
Sastoji se od skupa stand-alone servera koji obezbeđuju specifične servise (štampa, Web, baza podataka,...), skupa klijenata koji pozivaju te servise i mreže koja omogućuje udaljeni pristup.
Komponente:

- Klijenti, serveri.

Konektori:

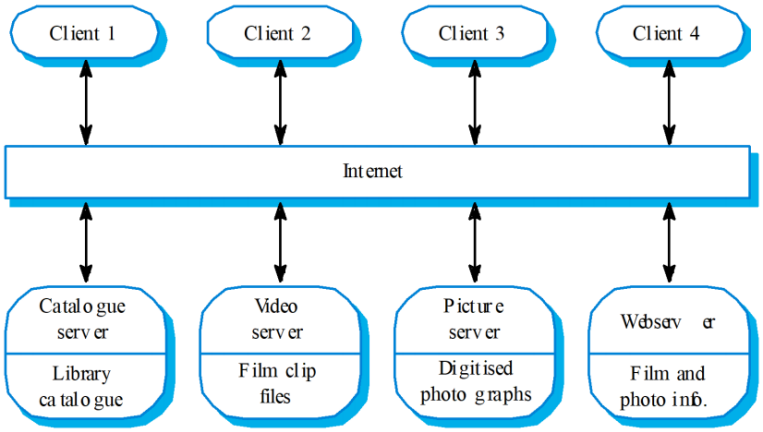
- Mreža, servisi servera.

Prednosti:

- Efikasno korišćenje mrežnih sistema
- Omogućava korišćenje slabijeg HW-a za klijente, obzirom da server odrađuje većinu posla
- Lako dodavanje novih servera i upgrade postojećih

Nedostaci:

- Neefikasna razmena podataka između klijenata (moraju da idu preko servera)
- Redundantnost podataka
- Ne postoji centralni registar imena servera i servisa pa nije lako otkriti koji serveri i servisi su na raspolaganju



80.5 Slojeviti (Layered) Model

Ovaj model se koristi kod modeliranja interfejsa među podsistemima.

Sistem se organizuje u skup slojeva (layera) od kojih svaki obezbeđuje jedan skup funkcionalnosti sloju iznad i služi kao klijent sloju ispod.

Omogućava inkrementalni razvoj podkomponenti u različitim slojevima.

Komponente:

- Slojevi

Konektori:

- Interfejsi

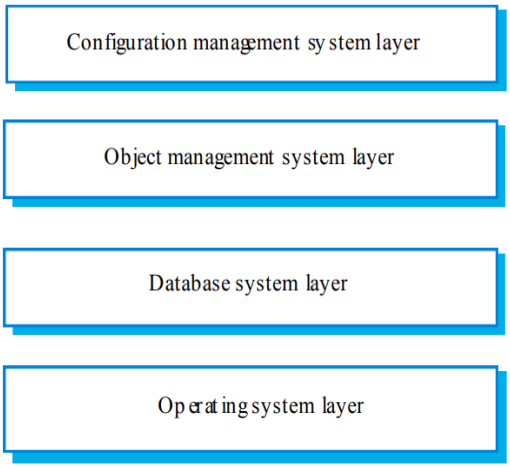
Prednosti:

- Promena interfejsa jednog sloja može da utiče na maksimalno još dva sloja
- Laka zamena jednog sloja drugim ukoliko su im interfejsi identični
- Baziran je na visokom nivou apstrakcije

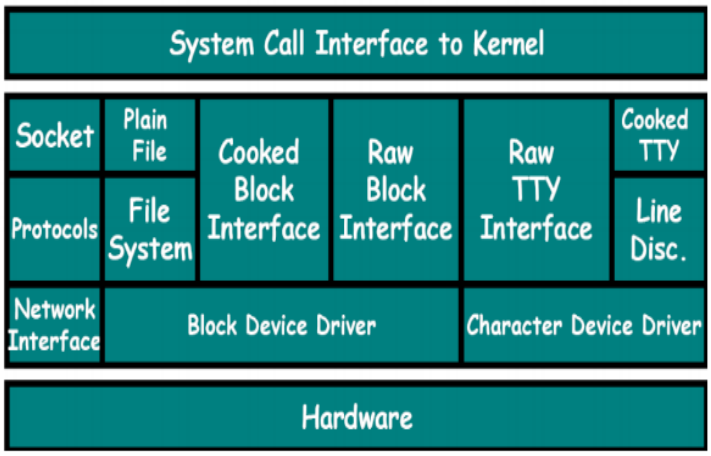
Nedostaci:

- Ne mogu svi sistemi da se lako organizuju po ovom modelu

Sistem kontrole verzija



OS Unix



80.6 Event driven model (Implicitno pozivanje)

Ovaj model se koristi kod sistema koji su upravljani eksterno generisanim događajima (events).
Postoje dve osnovne grupe ovih modela:

- Broadcast modeli
- Interrupt-driven modeli

Komponente:

- Komponente i podsistemi koji generišu ili obrađuju događaje

Konektori:

- Broadcast sistem i event procedure.

Prednosti:

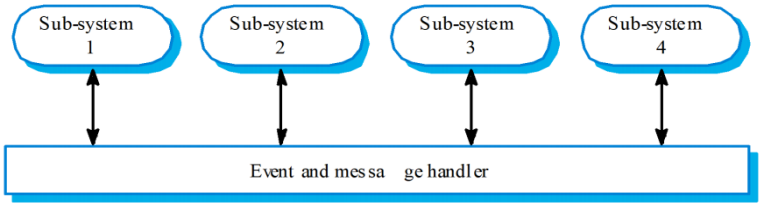
- Podrška višestrukom korišćenju SW-a (reuse);
- Laka evolucija sistema.
- Lako uvođenje nove komponente u sistem (jednostavno se registruje za neki event).

Nedostaci:

- Kada komponenta generiše događaj ona ne može da zna da li će neka komponenta da odgovori na njega i kada će obrada događaja biti završena

80.6.1 *Broadcast modeli*

Kod ove grupe modela, generisani događaj (event) se prosleđuje svim komponentama i podsistemima u sistemu.
Svaka komponenta koja upravlja generisanim događajem može da obradi događaj.
Podsistemi ne znaju da li će i kada eventi biti obrađeni.
Podsistemi se registruju za određene događaje i kada se oni generišu upravljanje se prenosi na podsistem koji upravlja tim događajem.
Efikasni su kod integracije podsistema koji se nalaze na različitim računarima u mreži.

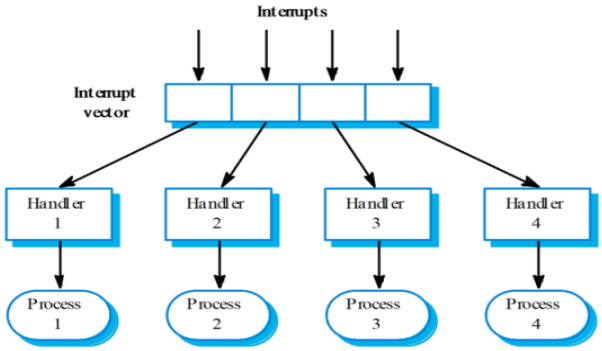


Ovaj model se često koristi kod razvojnih okruženja za integraciju alata:

1) Debager se zaustavi na prekidnoj tački i generiše događaj da je to uradio.
2) Editor odgovara na taj događaj tako što skroluje sadržaj koda na liniju gde je postavljena prekidna tačka.

80.6.2 *Interrupt-driven modeli (modeli upravljani prekidima)*

Koriste se kod sistema za rad u realnom vremenu gde je osnovna stvar brzi odgovor sistema na neki događaj.
Obezbeđuju brzu reakciju sistema na događaje, ali su komplikovani za realizaciju i pogotovo za testiranje i validaciju sistema.



80.7 Control model

Koristi se kod sistema gde je potrebna centralizovana kontrola.

Kontrolni podsistem upravlja tokom informacija između ostalih podsistema.

Postoje četiri osnovne grupe ovih modela:

- Call-return modeli
- Manager modeli
- Feed-back modeli
- Open-loop modeli

Komponente:

- Kontrolni algoritam i podsistemi.

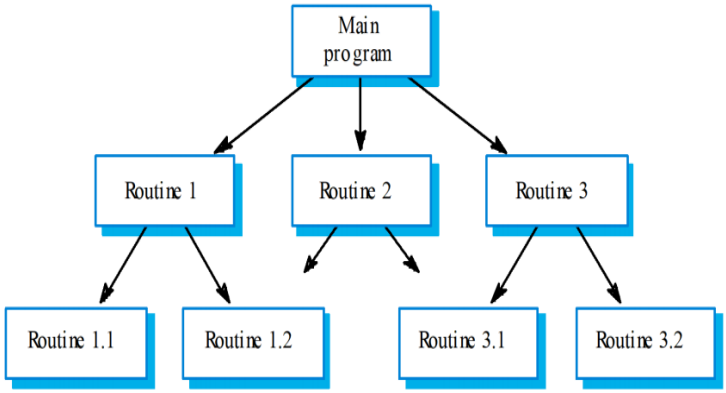
Konektori:

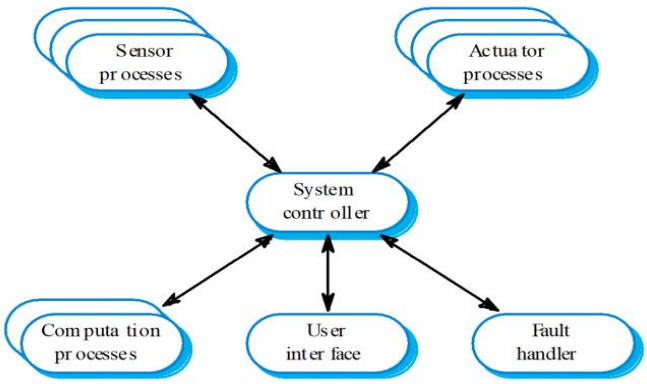
- Relacije između tokova podataka.

80.7.1 *Call-return modeli*

Kod ove grupe modela, kontrola kreće od vršnih podsistema i proteže se naniže (top-down pristup).

Pogodni su za sekvencijalne sisteme.





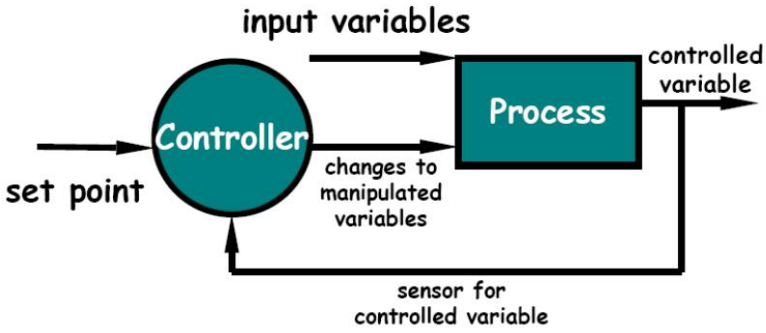
80.7.2 *Manager modeli*

Ova grupa modela se primenjuje kod konkurentnih sistema.

Jedna sistemska komponenta određuje početak, zaustavljanje i koordinaciju rada svih procesa u sistemu.

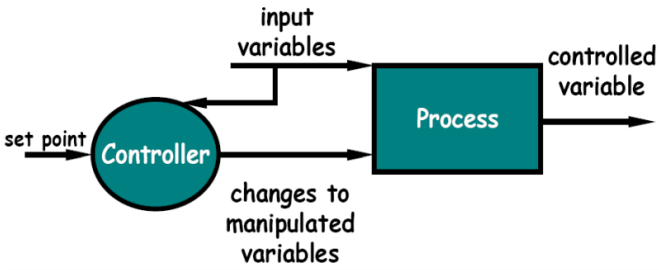
80.7.3 *Feed-back modeli*

Kod ove grupe modela, neke promenljive sistema se kontrolišu i njihove vrednosti se koriste za podešavanje sistema.



80.7.4 *Open-loop modeli*

Kod ove grupe modela, neke promenljive sistema se kontrolišu ali se njihove vrednosti ne koriste za podešavanje sistema.



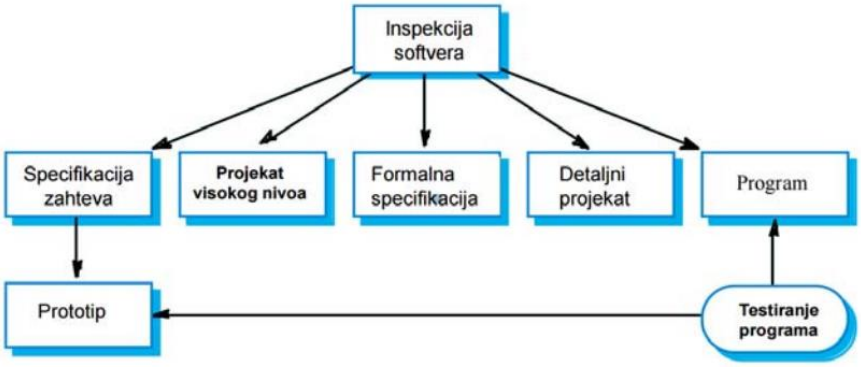
81. Šta je verifikacija, a šta validacija?

Verifikacija predstavlja pitanje da li na pravi način razvijamo softver tj. da li softver koji pravimo ispunjava sve zahteve navedene u specifikaciji. Validacija predstavlja pitanje da li pravimo pravi proizvod tj. da li je proizvod koji razvijamo saglasan sa onim što od njega očekuje korisnik za kojeg ga razvijamo. Ukoliko specifikacija nije dobro napisana može da se desi da proizvod prođe verifikaciju i zadovolji sve uslove iz specifikacije ali ne prođe validaciju jer korisnik jednostavno nije zadovoljan i proizvod ne poredstavlja ono što je on očekivao i ono što mu je zapravo potrebno.

82. Statička i dinamička verifikacija

Statička verifikacija odnosi se na inspekciju softvera i podrazumeva analizu statičkih reprezentacija sistema kako bi se otkrili problemi. Primeri statičkih reprezentacija sistema su kod i dokumenti pa se u okviru statičke verifikacije često koriste alati za analizu koda ili pregled dokumentacije koja je predmet inspekcije.

Dinamička verifikacija odnosi se na testiranje softvera i podrazumeva pokretanje sistema sa test podacima i praćenje rezultata njegovog izvršenja tj. njegovog ponašanja u radu.



83. Inspekcija softvera

Inspekcija softvera predstavlja statičku verifikaciju softvera i podrazumeva proveru statičkih reprezentacija sistema i potragu za greškama u okviru njih. Statičke reprezentacije sistema su npr. kod i dokumenti. Često se u okviru inspekcije softvera koriste alati za analizu softvera ili za pregledavanje dokumentacije koja je predmet testiranja.

Cilj inspekcije softvera je ispitivanje izvorne reprezentacije softvera i traženje anomalija i grešaka u okviru nje. Ne podrazumeva pokretanje programa tako da se može vršiti pre implementacije. Može biti primenjena na bilo koju reprezentaciju softvera (na zahteve, projekat, konfiguracione podatke, ...). Inspekcija softvera pokazala se kao vrlo efikasna tehnika u otkrivanju grešaka u softveru.

84. Metode testiranja VRLO ČESTO

Metode testiranja se svrstavaju u tri grupe:

- Metode crne kutije
- Metode sive kutije
- Metode bele kutije

Metode bele i crne kutije su krajnji slučajevi, a metode sive predstavljaju neku mešavinu prethodno pomenuta dva tipa. Kriterijumi za kategorizaciju su to da li se pri razvoju test slučajeva pristupa izvornom kodu softvera koji se testira ili se testiranje vrši preko korisničkog interfejsa ili nekih predviđenih API-ja.

Metode crne kutije – Ne znamo kako je softver iznutra urađen već mu pristupamo kao bilo koji drugi korisnik preko korisničnog interfejsa, ili ukoliko je to neka serverska komponenta koju testiramo i ona ima neki API, mi onda testiramo taj API. Ali suština je da test inženjer ne zna šta se odvija iznutra.

Metoda bele kutije – Obrnuta situacija od metode crne kutije, osoba koja testira ima uvid i kontrolu nad izvornim kodom u toku testiranja. Ovu vrstu testiranja uglavnom ne obavljaju testeri već developeri prilikom razvoja odgovarajućih komponenti.

Metode sive kutije – Podrazumevaju da nemamo uvid u kod ali možemo da pratimo neke međureprezentacije. Npr. imamo neki sistem koji radi sa bazom podataka pa mi samom sistemu pristupamo preko korisničkog interfejsa ili API-ja a onda vršimo monitoring baze podataka

85. Podela u particije ekvivalencije

Ulazni podaci i izlazni rezultati se razvrstavaju u neke klase, na taj način ćemo moći da izvršimo bolje testiranje jer ćemo tražiti da imamo za svaku klasu par predstavnika. Te klase se nazivaju particije ekvivalencija. Ideja je da u okviru particije ekvivalencije se program ponaša na ekvivalentan način za sve članove. Potrebno je identifikovati sve particije ekvivalencije za sistem koji se razvija i zatim odabrati makar po jedan test slučaj iz svake particije, uglavnom se ipak bira više od jednog i to po preporuci je najbolje po tri, po jedan na svakoj od granica particije ekvivalencije i jedan iz sredine particije. Najprimeljivije je na neke numeričke ulaze kada se podela u particije vrši po vrednosti. Najbitnije je u suštini istestirati sve granične slučajeve jer se u okviru njih najčešće javljaju greške, a test slučajevi sa sredine particije su tu da testiraju normalno ponašanje.

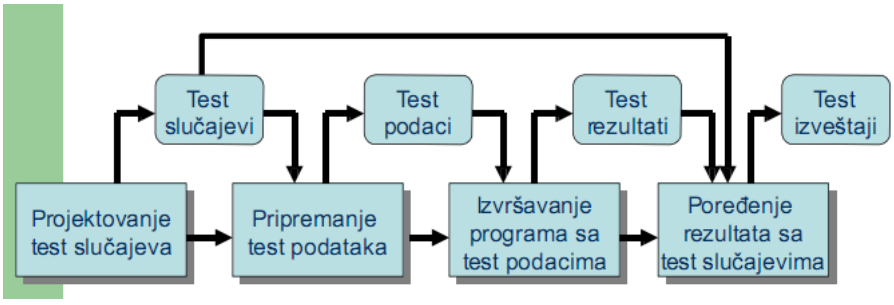
86. Testiranje puteva *(ne pada se često)*

Cilj je obezbediti takav skup test slučajeva da se svaki nezavisni put programa izvrši bar jedanput. Ako se to ispuni tj. ako se izvrši svaki nezavisni put programa to znači da će se i svaka naredba, čak i svaka uslovna naredba, u programu izvršiti bar jednom. Početna tačka ovog tipa testiranja je graf toka programa tj. algoritam izvršenja tog programa – čvorovi predstavljaju naredbe, a grane tok programa kroz te naredbe. Nezavisan put u programu predstavlja svaki put koji prolazi makar jednom novom granom (granom kroz koju se do sada nije prošlo). Ovaj tip testiranja ne koristi se na visokim nivoima već se koristi za neko bazično testiranje pojedinačnih funkcionalnosti tj. najčešće matematičkih izračunavanja.

Ciklomatska složenost predstavlja jednu od mera složenosti programa, u ovom slučaju je bitna jer označava broj testova potrebnih za testiranje svih upravljačkih naredbi. Kod strukturnih programa ciklomatska složenost je 1 + broj uslova u programu. Bitno je pomenuti da izvršavanje svih puteva programa ne podrazumeva izvršavanje svih kombinacija puteva programa.

87. Model procesa testiranja softvera

Prva faza je projektovanje test slučajeva i kao izlaz iz nje dobijamo test slučajeve. Oni se koriste za pripremanje test podataka, kao rezultat imamo test podatke koji se koriste ta izvršavanje programa sa test podacima nakon čega dobijamo test rezultate. Zatim vršimo poređenje tako dobijenih test rezultata sa test slučajevima dobijenim u prvoj fazi i dobijamo test izveštaje.



88. Struktura test slučaja VRLO ČESTO

U osnovi ima 3 celine – uvod, aktivnosti i očekivani rezultati.

Uvod se sastoji od osnovnih informacija o test slučaju – identifikator, verzija definicije (ako je bilo više verzija treba da se stavi oznaka za to), naziv, id zahteva koji je povezan sa tim test slučajem, namena tj. širi opis testiranja, zavisnosti.

Aktivnosti – okruženje (šta je potrebno od sw i hw da bi se izvršio test tslučaj), preduslovi tj. inicijalizacija, akcije tj. korak po korak toka testiranja, finalizacija tj. opis akcija koje treba izvršiti nakon akcija testa i ulazni podaci
Očekivani rezultati – šta se može videti nakon obavljenog testa kao rezultat njegovog izvršenja

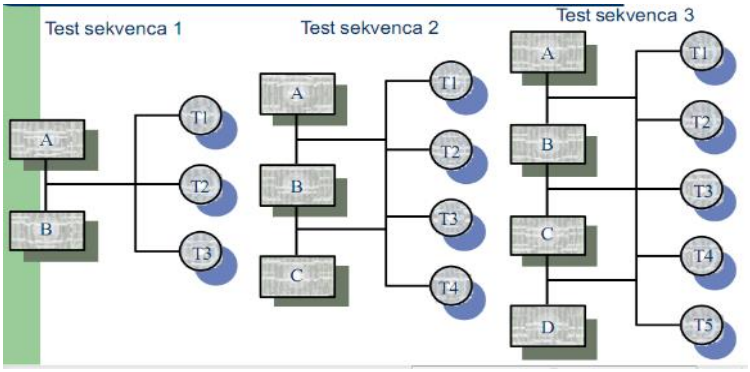
89. Faze testiranja

- Komponentno testiranje – testiraju se pojedinačne komponente
- Testiranje modula – testiraju se povezane kolekcije zavisnih komponenti
- Testiranje podsistema – moduli se integrišu u podsisteme koji se testiraju, ovde je akcenat na testiranju interfejsa među modulima jednog podsistema
- Testiranje sistema – testiranje sistema u celini tj. njegovih najbitnijih karakteristika
- Prijemno testiranje – testiranje sa podacima korisnika da bi se proverilo da li je sistem prihvatljiva za korisnika

90. Integraciono testiranje

Kompletan sistem dobijamo sastavljanjem podsistema koji su dobijeni sastavljanjem modula koji sadrže komponente. U integracionom testiranju treba da po modelu crne kutije vidimo da li ceo sistem radi zapravo sve ono što treba. Postavlja se pitanje zašto se vrši ovo testiranje tj. ako svaka od komponenti radi kako treba zasebno zašto ne bi radilo i sve skupa kako treba, ali postoji puno razloga zašto stvari nakon integracije mogu da krenu po zlu, jedna od njih je da se developeri nisu dobro razumeli prilikom dogovora oko interfejsa pa se jave problemi i pri samoj integraciji i pri čitavom funkcionisanju integrisanog sistema. Postoje 2 pristupa integracionom testiranju:

- Monolitno – koje podrazumeva integraciju celog sistema odjednom i tek onda testiranje čitave celine. Problem kod ovog pristupa je teškoća lokalizacije greške
- Inkrementalno – deo po deo se integriše u celinu i pri povezivanju svakog od delova vrši se testiranje. Na ovaj način se omogućava lokalizacija problema obzirom na to da u svakom trenutku imamo samo jednu komponentu koja je dodata a netestirana u celini za razliku od monolitnog pristupa kod kog imamo slučaj da su sve komponente dodate a netestirane u celini.



91. Alfa, beta i gama testiranje

Testiranja se identifikuju po tome u kom ciklusu se obavlja.

Alfa testiranje – vrši se u alfa fazi životnog ciklusa softvera tj. u toku razvoja softvera. Obzirom na to da tada imamo uvid u kod možemo da vršimo metode bele kutije, pored toga može da se vrši inspekcija softvera.

Beta testiranje – kada želimo da izbacimo novu verziju programa najpre izbacimo verziju koju ne želimo da označimo kao potpuno pouzdanu već želimo da nam ona posluži za masovno testiranje, ta verzija programa naziva se beta verzija i to testiranje beta testiranje. Ono podrazumeva da korisnici koji su voljni da se izlože tom nekom riziku korišćenja beta verzije aplikacije služe kao testeri i da se prilikom njihovog korišćenja aplikacije šalje neki automatski feedback tj. logovi korišćenja kojiće da služe razvojnom timu kao naznake funkcionalnosti softvera i kao informacije o verziji. Ovakvo testiranje daje veće mogućnosti za testiranje pod različitim uslovima i na različitim uređajima od alfa testiranja, jer koliko god da je velika firma koja se bavi razvojem sigurno će u opštoj populaciji korisnika da ima više različitih kombinacija uređaja, programa i okruženja u kojima softver može da se izvršava

Gama testiranje – testiranje koje se obavlja tokom same eksploatacije softvera, fukncioniše uglavnom na principu analitike tj. slanja logova o greškama i problemima koje se javljaju kod korisnika

92. Upravljanje softverskim projektima

Pod upravljanjem projektima se podrazumeva organizovanje, planiranje i raspoređivanje softverskih projekata. Odnosi se na aktivnosti koje treba da osiguraju da se softver isporuči na vreme i po planu u skladu sa zahtevima organizacije koja ga kreira i organizacije koja ga nabavalja. Predstavlja bitan aspekt jer je razvoj softvera uvek limitiran budžetom i raspoloživim vremenom za rad. Upravljanje softverskim projektom je specifično i razlikuje se od upravljanja projektima u drugim inženjerskim disciplinama zato što:

- Proizvod je nevidljiv, tj. neopipljiv je za razliku od drugih inženjerskih diciplina u kojima je lako videti dokle se stiglo sa razvojem (npr. kad gradimo kuću možemo na osnovu toga koliko smo sagradili da znamo tačno dokle smo stigli). Kod softverskog proizvoda je mnogo teže odrediti dokle se stiglo sa razvojem, postojanje neke funkcionalnosti ili nekog skupa funkcionalnosti ne podrazumeva nužno da se daleko stiglo, može da posoji neki nefunkcionalni zahtev koji nije ispunjen ili još neki funkcionalni koji zavisi od navedenog skupa ili od kojeg navedeni skup zavisi. Ta veza između implementacionih delova je znatno kompleksna i nije tako lako odrediti dokle se stiglo kada se posmatra procentualno ispinjenje finalnog cilja. Nekada može da nam deluje da imamo dosta ali da zapravo zbog neispunjenosti nekih nefunkcionalnih zahteva sistem u globalu nije funkcionalan po standardima koje je trebao da zadovolji i da mi zapravo nemamo ništa.

- Proizvod je fleksibilan – možemo da ga menjamo u srži u bilo kom trenutku razvoja, često i kupac koji je zahtevao kreiranje može u toku samog razvoja da menja zahteve i da traži da se oni ispune u bilo kom trenutku.

- Softversko inženjerstvo je mlada oblast, skoro je otkrivena u odnosu na ostale inženjerske discipline što čini njeno polje jako promenljivim i u stalnom razvoju.

- Sam proces razvoja softvera nije standardizovan, postoji mnogo različitih načina da se pristupi samom razvoju u zavisnosti od samog proizvoda.

- Mnogi softverski proizvodi su unikatni, tj. sam pojam softverskog porizvoda je toliko širok da može da ne postoji apsolutno ništa zajedničko za neka dva softverska proizvoda što dovodi do toga da i njihovi sami procesi razvoja moraju korenito da se razlikuju jedan od drugog, što nije slučaj kod drugih inženjerskih disciplina. (Kada gradimo kuću znamo kako ta gradnja treba da teče tačno, ima nekih specifičnosti ali generalno osnova je ista)

93. Upravljanje rizikom (Risk management)

Upravljanje rizikom predstavlja idnetifikovanje rizika i skiciranje plana minimizacije uticaja rizika na projekat, gde je rizik verovatnoća da se neka nepovoljna prilika pojavi. Rizici mogu da se razvrstaju u 3 kategorije:

- Rizici projekta – utiču na raspored ili resurse

- Rizici proizvoda – utiču na kvalitet ili performanse softvera koji se razvija

- Rizici poslovanja – utiču na organizaciju koja razvija ili nabavlja softver

94. Proces upravljanja rizikom

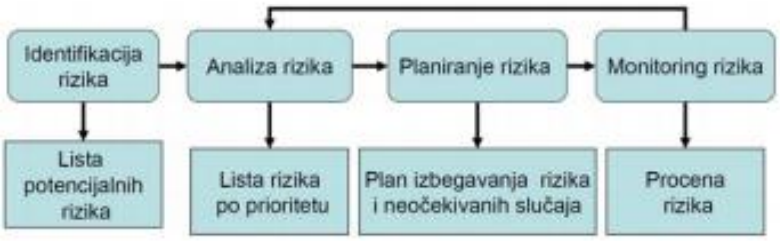
Proces upravljanja rizikom se sastoji od 4 koraka:

- **Identifikacija rizika** – identifikovati sve rizike iz kategorija rizici projekta, proizvoda i poslovanja. Kao izlaz daje listu potencijalnih rizika. Nema nekih striktnih pravila kako se obavlja ova faza, bitno je da onaj ko radi ima neko iskustvo i da zna otprilike koji se to problemi često javljaju i da ne bude previše optimističan povodom toka projekta već da realno sagledava šta bi sve moglo da pođe po zlu. Dobra polazna tačka za ovu fazu je da imamo neku unapred definisanu listu mogućih rizika kroz koju možemo samo da prođemo i da čekiramo koji od tih rizika bi mogli da se dese.

- **Analiza rizika** – Vrši se procena verovatnoće i posledice identifikovanih rizika, kao i ozbiljnost svakog od rizika tj. koliko je on opasan. Kao izlaz daje listu rizika po prioritetu (definisano je koliko je svaki od njih bitan). Verovatnoća u ovoj fazi se računa nekom procenom tj. daje joj se neka lingvistička vrednost – vrlo niska, niska, srednja, visoka i vrlo visoka. Takođe posledice rizika se označavaju nekim semantičkim nivoom značaja – katastrofalni, ozbiljni, tolerišući ili beznačajni.

- **Planiranje rizika** – ako se desi neki od rizika šta raditi, tj. kako izbeći ili makar minimizovati uticaj tog identifikovanog rizika na projekat. Kao izlaz daje plan izbegavanja rizika i neočekivanih slučaja. Ne postoji neki prost proces koji će da dozvoli lako planiranje rizika već u mnogome zavisi od samog menadžera projekta.

- **Monitornig projekta** – nadgledamo rizike u toku samog projekta. To je neka kontinuirana aktivnost u kojoj pokušavamo da uočimo da li se neki od predviđenih rizika dešava. Kao izlaz ima procenu rizika u odgovarajućem trenutku. Ako se uoči neki rizik onda se posmatraju njegovi efekti u odnosu na očekivane i to kako njegovo pojavljivanje utiče na verovatnoću pojavljivanja tog rizika.



95. Kategorije strategija za upravljanje rizikom

- **Strategija izbegavanja** – Smanjuje se verovatnoća pojave rizika. Npr. ako je rizik da će neko bitan da nam napusti tim mi se onda izuzetno trudimo da ga zadržimo, dajemo mu povlastice, povišice i razne neke benefite kako bi imao što veću želju da ostane tj. kako bi rizik njegovog odlaska bio što manji.

- **Strategija minimizacije** – Smanjiti uticaj datog rizika na na projekat ili proizvod. Opet ako je rizik odlazak nekoga iz tima mi se onda trudimo da u timu ne postoji niko ko neki posao radi skroz sam i da niko drugi nije upoznat sa tim delom posla, već se raspoloživo ljudstvo tako raspoređuje tako da ko god da odluči da ode moglo bi da dođe do preraspodele tako da neko drugi pokupi posao koji je on do tada radio.

- **Planovi za nepredviđene događaje** – Ukoliko prve 2 strategije nisu opcija onda treba napraviti ceo plan šta raditi u slučaju da se rizik desi, tj. kako nastaviti sa radom sa prisutnim rizikom. Npr. za slučaj da organizacija naiđe na finansijske probleme pri razvoju projekta onda treba ići na ubeđivanje rukovodstva u to da je baš vaš projekat bitan i da njega treba održati u životu.

96. Evolucija I servisiranje

Evolucija

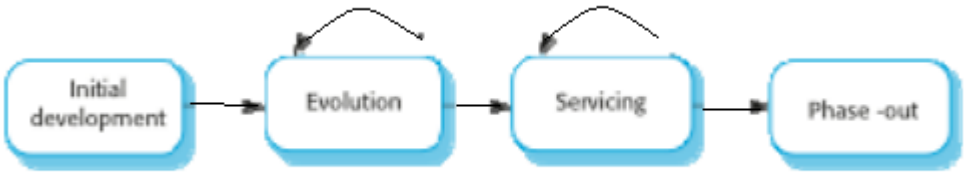
- Faza u životnom ciklusu softverskog sistema gde se softver nalazi u operativnoj upotrebi i evoluira kako se novi zahtevi implementiraju u sistem.

Servisiranje

- U ovaj fazi softver se i dalje operativno koristi, a izmene koje se vrše imaju za cilj održavanje njegove operativnosti.
- Ovo praktično znači da se ne dodaju nove funkcionalnosti, već se samo otklanjaju greške i vrše izmene usled promena u okruženju softvera.

Postepeno ukidanje (eng. phaseout)

- Softver se može i dalje koristiti, ali se nad njim ne vrše više nikakve izmene.



97. Tipovi održavanja softvera

- **Održavanje u cilju ispravke softverskih grešaka** - Izmena sistema kako bi se otklonili nedostaci koji sprečavaju da sistem radi u skladu sa svojom specifikacijom.
- **Održavanje sa ciljem prilogođenja softvera za drugačije radno okruženje** - Promena sistema tako da može da radi u drugačijem okruženju (računar, OS, itd.) od onog za koje je inicijalno implementiran.
- **Održavanje u cilju dodavanja nove ili izmene postojeće funkcionalnosti sistema** - Modifikacija sistema kako bi zadovoljio nove zahteve.

98. Šta je reinženjering sistema

Reč je o restruktuiranju ili ponovnom pisanju dela ili celog nasleđenog sistema bez promene njegove funkcionalnosti.
Primenjiv je onda kada neki ali ne svi podsistemi većeg sistema zahtevaju često održavanje.
Reinžinjering predstavlja ulaganje truda sa ciljem lakšeg održavanja. Sistem može biti restruktuiran i redokumentovan.

99. Upoređivanje reinženjeringa sa refaktorisanjem

//nešto o reinženjeringu prvo, pa onda nešto o refaktorisanju pa onda upoređivanje

Refaktoring je proces unapređenja programa kako bi se smanjila njegova degradacija kroz izmene tj. predstavlja „preventivno održavanje“ koje smanjuje probleme pri budućim izmenama. Refaktoring uključuje izmenu programa kako bi se poboljšala njegova struktura, smanjila složenost ili povećala razumljivost. Kada se refaktoriše program treba izbegavati dodavanje novih funkcionalnosti, već se skoncentrisati na njegovo unapređenje.

Reinženjering se obavlja nakon što je sistem bio održavan neko vreme i cena održavanja je porasla. Reinženjering nasleđenih sistema se vrši korišćenjem automatskih alata, kako bi novodobijeni sistem bio lakši za održavanje
Refaktorisanje je kontinuirani proces unapređivanja kroz razvoj i evoluciju. Koristi se kako bi se izbegla degradacija strukture i koda, a u cilju umanjenja cene i složenosti održavanja sistema