**Predmet SE201 UVOD U SOFTVERSKO INŽENJERSTVO**

**ISPITNA PITANJA**

1. **Lekcija 1: UVOD U SOFTVERSKO INŽENJERSTVO**
   1. Šta je Softversko inženjerstvo? Zašto je ono važno? Šta je razlika softverskog inženjerstva i računarskih nauka?

Softversko inženjerstvoje inženjerska disciplina koja se bavi svim aspektima proizvodnje

softvera. Softversko inženjerstvo podržava timski, a ne individualni razvoj profesionalnog

softvera i uključuje tehnike koje podržavaju specifikaciju programa, njegovo projektovanje

i evoluciju.

Od ranih faza specifikacije sistema, do održavanja nakon početka njegove upotrebe.

To je softverska disciplina jer obezbeđuje rešavanje problema i u nedostatku svih potrebnih

teorija i metoda, pri čemu se uzimaju u obzir i organizacijska i finansijska ograničenja.

Softversko inženjerstvo se bavi svim aspektima proizvodnje softvera, jer se ne bavi samo

procesima razvoja tehničkih procesa razvoja softvera, već uključuje i aktivnosti, kao što su

upravljanje softverskim proizvodom, kao i alatima, metodima i teorijama koje podržavaju

proizvodnju softvera.

Softversko inženjerstvo je mlada inženjerska disciplina koja još nije uspela da dostigne zrelost drugih inženjerskih disciplina (industrija automobila, građevinarstva i dr.) u kojima su vrlo retki slučajevi da njihovi proizvodi imaju „ugrađene greške“

* 1. Šta je softverski proizvod? Koje dve vrste softverskih proizvoda postoje?

Softver čine i program, ali i njegova dokumentacija, kao i podaci o konfigurisanju softvera koji omogućavaju da ovi program normalno rade.

Softver nije isto što i program. Softver čine i program, ali i njegova dokumentacija, kao i podaci o konfigurisanju softvera koji omogućavaju da ovi program normalno rade. Kako profesionalni softver najčešće koristi više programa, neophodno je da postoji informacija o konfiguraciji softvera, tj. o programima koje treba pozivati i izvršavati tako da softver ostvari svoju funkciju. Zato mora da postoji dokumentacija o sistemu koja opisuje strukturu softverskog sistema, korisničku dokumentaciju, koja objašnjava kako se koristi sistem, kao i informacija o veb sajtovima sa kojih korisnik može da preuzme najnovije informacije o proizvodu.

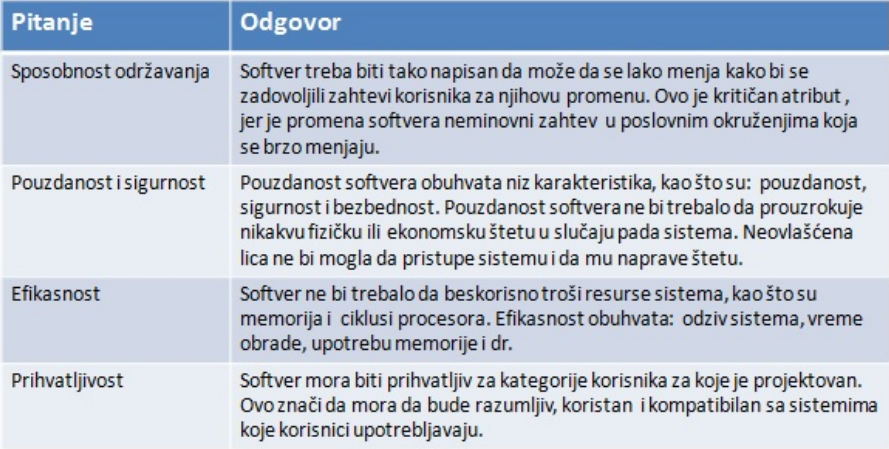
To je osnovna razlika između razvoja profesionalnog softvera i razvoja amaterskog softvera. Kako se profesionalni softver koriste ljudi koji nisu razvijali softver, to je neophodno da se on dokumentuje tako da može biti pravilno korišćen. Postoje dve vrste softverskih proizvoda:

**1. Generički proizvodi:** To su softverski proizvodi koji su razvijeni radi prodaje na tržištu. Mogu ih koristiti različiti korisnici koji nisu poznati u vreme razvoja. Organizacija koja razvija softver definiše programsku specifikaciju. Primeri: obrađivači tekstova, sistemi za upravljanje baza podataka, CAD sistemi.

**2. Proizvod razvijeni po narudžbini:** Firma koja razvija softver, to radi po zahtevu kupca, koji definiše specifikaciju za softver. Primeri: kontrolni sistemi elektronskih uređaja, sistemi za upra

* 1. Koji su atributi dobrog softvera?

Softver nije samo program, već program sa dokumentacijom. Bitni atributi softverskog proizvoda su: sposobnost održavanja, pouzdanost, bezbednost, efikasnost i prihvatljivost.



* 1. Šta je softverski proces? Koje su četiri osnovne aktivnosti softverskog procesa?

Sistemski pristup koji se koristi u softverskom inženjerstvu se često naziva i softverskim procesom. Softverski procesje redosled aktivnosti koje vode proizvodnji softverskog proizvoda. Softverski proces čine četiri osnovne aktivnosti, koje su prisutne u svim softverskim procesima:

1. Specifikacija softvera - kada kupci i inženjeri definišu softver koji treba da se proizvede, kao i ograničenja u njegovom radu.

2. Razvoj softvera - kada se vrši projektovanje softvera i programiranje.

3. Validacija softvera - kada se proverava da li softver zadovoljava sve ono što kupac zahteva.

4. Evolucija softvera - kada se vrši promena softvera u skladu sa promenjenim zahtevima kupca i tržišta.

* 1. Koji su ključni izazovi softverskog inženjerstva?

Postoji puno različitih vrsta softvera. Nema univerzalnog metoda softverskog inženjerstva koji je primenljiv za sve te vrste. Ipak, postoje opšta pitanja koja su relevantna za mnoge različite vrste softvera: rad u heterogenom okruženju, brzo prilagođavanje novim zahtevima i bezbednost informacija.

Svi ovi faktori čine softversko inženjerstvo vrlo izazovnom disciplinom. Izazovi pri razvoju softvera zahtevaju od inženjera softvera da uzmu u obzir sve uticajne faktore na softverski sistem, prilikom njegovog projektovanja.

* 1. Koji tipovi softverskih aplikacija postoje?

Nema univerzalnih metoda softverskog inženjerstva koje odgovaraju svim sistemima i svim organizacijama. Izbor najpogodnije metode zavisi najviše od vrste aplikacija za koju se softver razvija:

1. Samostalna aplikacija: Ovi sistemi se izvršavaju izolovano na nekom računaru, npr. PC.

2. Interaktivne transakcione aplikacije: Aplikacija se izvršava na udaljenom računaru kojoj se pristupa sa korisničko PC računara ili terminala. To su najčešće veb aplikacije (za e-poslovanje), poslovni sistemi, i drugi sistemi kojima se pristupe preko veb prikazivača. Interaktivne aplikacije obično sadrže velike baze podataka koje se osvežavaju transakcijama koje iniciraju korisnici.

3. Ugrađeni upravljački sistemi (Embedded Control Systems): Ugrađeni sistemi (hardver i softver) upravljaju radom hardverskim uređajem. Ovo su najbrojniji računarski sistemi. Primeri: mobilni uređaji, televizori i dr.

4. Sistemi sa paketnom obradom: Ovo su poslovni sistemi projektovani da vrše masovnu obradu podataka. Na osnovu velikog broja pojedinačnih unosa podataka, kreiraju se odgovarajući izveštaji. Primeri: priprema računa za struju.

5. Zabavni sistemi: Razvijeni sa ličnu upotrebu s namerom da zabave korisnika (video igre). Najvažniji deo sistema je interakcija sa korisnikom.

6. Sistemi za modelovanje i simulaciju: Ovi sistemi razvijaju naučnici i inženjeri da bi modelovali fizičke procese ili situacije. Ovi sistemi rade intenzivne proračune i zahtevaju paralelne sisteme visokih performansi za svoje izvršenje.

7. Sistemi za prikupljanje podataka: Ovi sistemi prikupljaju podataka iz svog okruženja preko odgovarajućih senzora. Podaci se zatim obrađuju.

8. Sistemi sistema: Ovo su sistemi koji čine veći broj drugih sistema.

* 1. Koje su osnovne postavke softverskog inženjerstva koje su primenljive za sve tipove softverskih sistema?
  2. Koji su troškovi izraženi u softverskom inženjerstvu?

Najveći troškovi su najčešće vezani za aktivnosti promene u softveru, troškovi održavanja

* 1. Do kojih promena je došlo u Softverskom inženjerstvu zbog uticaja Interneta?
  2. Zašto dolazi do grešaka u radu softvera?

Greške u softveru su posledica povećanih očekivanja korisnika, s jedne strane, i niskog stepena primene metoda i tehnika softverskog inženjerstva, s druge strane.

Bez obzira na svakodnevnu primenu, postoji mnogo primera neuspešnog korišćenja softvera koji obuhvata i mnogo grešaka u samom radu softvera. Do tih grešaka najčešće dolazi iz sledećih razloga:

1. Povećani zahtevi. Razvoj softvera sve je više izložen novim zahtevima koji dovode do čestih promena i dopuna koje čine da softver bude sve složeniji. Ti zahtevi traže da se u što kraćem vremenskom roku realizuju. Postojeće metode softverskog inženjerstva ne mogu da se nose sa ovakvim zahtevima, jer ne mogu da ih zadovolje te se moraju razviti nove tehnike softverskog inženjerstva u cilju rešavanja tog problema.

2. Niska primena metoda softverskog inženjerstva: Moguć je razvoj računarskih programa i bez primene metoda i tehnika softverskog inženjerstva. Često, organizacije koriste sisteme koji rade nepouzdano, te razvijaju programe isključivo za svoje potrebe. Sistemi su nepouzdani najčešće iz neznanja ili ne koriste metode i tehnike softverskog inženjerstva, već njihovi programeri počnu izradu programa bez odgovarajućih pripremnih aktivnosti, a i bez kasnijih odgovarajućih testiranja. Kao rezultat, često dobijaju softverske sisteme koji rade nepouzdano, te moraju stalno da nalaze i otklanjaju greške u softveru, a to utiče na povećanje troškova razvoja i održavanja softvera. Međutim, te organizacije mnogo veće troškove imaju u oblasti primene softvera, gde kvarovi u softverskim sistemima dovode do prekida ili ometanja u njihovom svakodnevnom poslovanju (na primer, zaustavljanje proizvodnje ili montaže proizvoda). Troškovi posledica do kojih dovode zastoji u softverskim sistemima su mnogo veći, nego troškovi otklanjanja tih grešaka u samom softveru. Očigledno, nedostaje veći stepen obrazovanja iz softverskog inženjerstva i obuke ljudi koji razvijaju softverske sisteme da bi se izbegao problem razvoja skupih (zbog stalnih prepravki i dorada), a nepouzdanih softverskih sistema (jer često ulaze u neočekivane zastoje, zbog grešaka u softveru, tzv. bagove).

Pravilnom upotrebom tehnika softverskog inženjerstva, ovi problemi i greške mogu biti smanjeni ili eliminisani, pa je zato od velikog značaja razumeti šta predstavlja softversko inženjerstvo i na koji način se adekvatno primenjuje.

* 1. Koji su standardi prihvatljivog ponašanja softverskih inženjera?
  2. Šta su principi ponašanja? Koji su osam principa ponašanja softver inženjera?

Ako ste softverski inženjer, očekuje se da primenjujete sve uobičajene standarde poštenja i integriteta ličnosti. Ne bi trebalo da upotrebite vaše znanje i veštine na nepošteni način koje mogu ugroziti reputaciju softverskog inženjerstva. Pored zakonskih ograničenja, postoje i pravila struke koje treba poštovati:

1. Poverljivost: Morate poštovati i čuvati poverljive informacije vašeg poslodavca i vaših klijenata, nezavisno od toga da li ste potpisali sporazum o poštovanju poslovne tajne.

2. Kompetencija:Ne smete da ističete kompetencije koje nemate. Ne smete prihatiti posao za koji niste kompetentni.

3. Prava intelektualne svojine:Morate biti upoznati sa zakonima koji regulišu pitanja intelektualne svojine, kao što je korišćenje patenata i autorska prava. Morate poštovati autorska prava poslodavca i vaših klijenata.

4. Zloupotreba kompjutera:Ne bi smeli da koristite vaše tehničke veštine i da zloupotrebite pristup računarima drugih. Bilo da se radi o trivijalnoj zloupotrebi (igranje igrica) pa do ozbiljnih dela (širenje virusa).

1. **Lekcija 2: SOFTVERSKI PROCESI**
   1. Šta je softverski proces? Koje su osnovne aktivnosti softverskog procesa?

Softverski proces je skup povezanih aktivnosti koji vodi proizvodnji softverskog proizvoda. Ove aktivnosti mogu da dovedu do razvoja potpuno novog softvera u nekom od standardnih programskih jezika (npr. Java ili C), ili do usavršavanja nekog postojećeg softvera, što je najčešći slučaj kod poslovnih aplikacija. Najčešće se softverski procesi sadrže sledeće četiri osnovne aktivnosti koje čine osnovu softverskog inženjerstva:

1. Specifikacija softvera: Definiše funkcionalnost softvera i ograničenja na rad softvera.

2. Projektovanje (dizajn) i razvoj (implementacija) softvera: Definiše kako softver ostvaruje svoju specifikaciju i kako se on pravi.

3. Provera (validacija) softvera: Softver se proverava da bi se utvrdilo da li zadovoljava potrebe i očekivanja korisnika.

4. Evolucija softvera: Softvter mora da ima svoju evoluciju (stalni razvoj i prilagođavanje) da bi zadovoljio nove zahteve korisnika

* 1. Navedita dve osnovne vrste softverskih procesa i ukratko ih opišite.

Procesi vođeni planomsu procesi kod kojih su sve aktivnosti unapred planirane i napredak u razvoju softvera se određuje stepenom ostvarivanja tog plana.

Agilni procesi su procesi kod koji se planirane radi postupno (inkrementalno) kako bi se proces lakše prilagodili promenljivim zahtevima korisnika.

* 1. Šta je model softverskog procesa? Navedite tri opšta metoda softverskih procesa. Objasnite svaki od njih. Koje su međusobne razlike?

Model softverskog procesa je uprošćeno predstavljanje nekog softverskog procesa. Svaki model procesa predstavlja neki proces iz neke posebne perspektive, te zbog toga, obezbeđuje samo delimične informacije o procesu, tj. informacije važne za tu perspektivu (npr. funkciju softvera). Zato se najpre daju vrlo uopšteni modeli softverskih procesa (tzv. „paradigme procesa“) koji se prikazuju iz perspektive arhitekture procesa. Oni prikazuju tok procesa sa aktivnostima, ali ne prikazuju i detalje o svakoj aktivnosti. Zbog toga, ovi opšti modeli predstavljaju samo apstrakcije procesa (uprošćeni prikaz) razvoja softvera. Njihova namena je da ukažu na različite pristupe u razvoju softvera. Oni su samo osnova za razvoj detaljnijih modela procesa. Najčešće se navode sledeća tri opšta modela softevrskih procesa:

1. Model vodopada (engl., the whaterfall model): On prikazuje osnovne aktivnosti procesa : specifikacija, razvoj, provera (validacija) i evolucija. Ove aktivnosti su prikazane kao posebne faze procesa, kao što su: specifikacija zahteva, projektovanje (dizajn) softvera, primena (implementacija), testiranje i dr. koje se jedna za drugom realizuju.

2. Inkrementalni (postepeni) razvoj: On povezuje aktivnosti specifikacije, razvoja, i provere, kao niz serija verzija (inkremenata) softvera, pri čemu svaka verzija dodaje određenu funkcionalnost na prethodnu verziju.

3. Softversko inženjerstvo zasnovano na višestrukoj upotrebljivosti (eng. Reuseoriented software engineering): Ovaj pristup se oslanja na korišćenje komponenata softevera koje se mogu višestruko koristiti. Ovim procesom se integrišu postojeće softverske komponente u sistem, umesto da se razvijaju nove.

* 1. Detaljno opišite model vodopada za razvoj softvera. Šta je proces projektovanja sistema? Šta je projektovanje softvera? Šta je implementacija softvera? Šta je testiranje jedinica? Šta je operativni rad i održavanje? Kada se koristi model vodopada?

Model vodopada je planski vođeni proces jer se ceo proces mora planirati i odrediti termini za sve aktivnosti procesa, pre nego što počne njegovo izvršenje.

Model vodopada je primer planski vođenog procesa jer se ceo proces mora planirati i odrediti termini za sve aktivnosti procesa, pre nego što počne njegovo izvršenje (slika 1 ). Glavne faze modela vodopada su direktno povezane sa osnovnim aktivnostima razvoja softvera:

1. Analiza i definisanje zahteva: Definišu se servisi sistema, ograničenja i ciljevi, definisani uz konsultaciju sa korisnicima sistema. Detaljnije se opisuju kao sistemska specifikacija.

2. Projektovanje sistema i softvera : Proces projektovanja sistema raspoređuje zahteve svim komponentama sistema i uspostavljanje celokupne arhitekture sistema, kao i utvrđivanje i opisivanje osnovnih apstrakcija softverskog sistema i njihove međuzavisnosti (relacije).

3. Implementacija i testiranje jedinica:U ovoj fazi je projektno rešenje softvera realizovano skupom programa ili programskih jedinica. Testiranje jedinica utvrđuje da li svaka jedinica ostvaruje svoju planiranu funkciju.

4. Integracija i testiranje sistema: Sve programske jedinice u ovoj fazi se integrišu u sistem i testiraju se kao kompletan sistem radi provere da li softver zadovoljava postavljene zahteve, tj. ostvaruje svoje funkcije i performanse. Posle ovog testiranja, softverski sistem se isporučuje kupcu.

5. Operativni rad i održavanje: Ovo je obično najduža faza životnog ciklusa softvera. Sistem je instalisan i pušten u operativni rad, tj. u upotrebu. Održavanje obuhvata ispravljanje grešaka koje nisu otkrivene u ranijim fazama životnog ciklusa, poboljšavanja primene programskih jedinica i poboljšanje usluga softverskog sistema u skladu sa novopostavljenim zahtevima.

* 1. Šta je inkrementalni razvoj softvera? Zašto se koristi? Šta je inkrement? Koje su prednosti inkrementalnog razvoja u odnosu na model vodopada? Kako se primenjuje inkrementalni razvoj softvera? Koji su problemi u primeni?

Inkrementalni razvoj se zasniva na postupku po kome se razvije neka početna implementacija softvera, koja se pokazuje korisniku. Na osnovu njegovih primedbi, napravi sa nova verzija softvera, i tako preko više verzija, dolazi se do konačne verzije, koja predstavlja razvijeni softver. Inkrementalni razvoj, koji čini osnov agilnih pristupa u razvoju softvera, je bolji od modela vodopada za najveći broj poslovnih aplikacija, e-poslovanje i personalnih sistema.

Inkrementalni razvoj odražava način kako mi često rešavamo probleme. Mi retko pravimo detaljan plan za budućnost, već idemo u budućnost pristupom korak-po-korak. Pri tome se često vraćamo nazad, da bi ispravili uočene greške. Inkrementalni razvoj softvera je jeftiniji i lakši za rad kada se često traže promene u softveru tokom njegovog razvoja.

Svaki inkrement, tj. nova verzija softvera, sadrži neke od potrebnih funkcija (tj. funkcionalnost) koje traži korisnik. Prve verzije obično sadrže najvažnije funkcije, ili najhitnije zahtevane funkcije. Ovo omogućava da korisnik softvera može da u ranim fazama razvoja softvera oceni da li softver obezbeđuje funkcije koje se od njega zahtevaju. Ako se utvrdi da to softver ne obezbeđuje, trenutna verzija softvera se onda menja, a nova funkcija se ostavlja za neku kasniju verziju, tj. inkrement.

* 1. Zašto je inkrementalni razvoj najefektniji pristup u razvoju poslovnih softverskih sistema? Zašto je ovaj model manje pogodan za razvoj sistema u realnom vremenu?

Inkrementalni razvoj preko više verzija, vršio poboljšanje siftvera. Svaki softverski inkrement se može i iterativno razvijati, pri čemu svaka verzija poboljšava prethodnu.

Iteracija znači ponavljanje. Iterativni razvoj znači da se proizvod poboljšava ponavljanjem nekih aktivnosti u procesu razvoja da bi se izvršila njegova poboljšanja. Iterativni razvoj se može primeniti i u slučaju razvoja inkrementa. Svaki inkrement se razvija u više verzija, pre čemu svaka naredna verzija prestavlja poboljšanje prethodne verzije.

Korišćenje inkremenata i iteracija su bitni delovi više poznatinh inkrementalnih metoda razvoja softvera (npr. Srum, RUP). Slika 4 prikazije princip njihovog kombinovanja.

DOBRE I LOŠE STRANE PRIMENE MODELA INKREMENTALNOG RAZVOJA

Inkrementalni razvoj obezbeđuje niže troškove razvoja, mišljenje korisnika i brži razvoj softvera. Nedostatci su u nevidljivosti procesa i postepeno urušavanje arhitekture sistema.

Inkrementalni razvoj, u odnosu na model vodopada, obezbeđuje sledeće prednosti:

1. Niži troškovi realizacije zahteva korisnika. Količina analiza i dokumenata koja treba da se ponovo uradi je znatno manja nego što je u slučaju primene modela vodopada.

2. Lakše je obezbediti mišljenje korisnika na rezultat razvoja, jer on može da komentariše demonstraciju radne verzije softvera i da vidi da li su njegovi zahtevi primenjeni. Korisnici se teško snalaze u dokumentaciji, te ne mogu na osnovu nje da ocenjuju da li su njihovi zahtevi zadovoljeni. Tek kada vide kako softver radi, mogu da komentarišu, a to inkrementalni razvoj obezbeđuje.

3. Brža isporuka i instalacija softvera kod korisnika, iako nema sve tražene funkcionalnosti (funkcije). Korisnici mogu softver da koriste ranije, iako bez svih traženih funkcija, nego što je to slučaj kod modela vodopada.

* 1. Predložite model opšteg softverskog procesa za razvoj sledećih sistema:
     1. Rezervacija avionskih karata i hotela
     2. Sistema za prodaju proizvoda preko veba
     3. Upravljanje sistemom podzemne železnice u gradu
  2. Šta je softversko inženjerstvo za višestruku upotrebu? Šta su softverske komponente? Koje tipove komponenata imamo?Opišite proces razvoja softvera baziranog na ponovno upotrebljivim komponentama. Opišite ukratko svaku aktivnost?

U većini softverskih projekata, prisutna je neka softverska komponenta koja je ranije razvijena. Ovo se obično dešava neformalno kada ljudi koji rade na projektu poznaju dizajn ili kod koji je sličan onome koji je tražen. Oni ga nalaze, modifikuju kao što je zahtevano i objedinjuju ga u sistem.

Razvoj zasnovan na ponovnoj upotrebi komponenata zasniva se na velikoj bazi ponovno korišćenih softverskih komponenti kojima se može pristupiti, kao i nekom integrativnom okviru za ove komponente. Ponekad ove komponente su sistemi (COTS ili Commercial-offthe-shelf sistemi) koji mogu biti korišćeni da obezbede posebnu funkcionalnost kao što je formatiranje teksta, numerički proračuni, itd. Dok su faza specifikacije početnih zahteva i faza validacije uporedive sa drugim procesima, međufaze u ovom procesu su različite. Ove faze su:

1. Analiza komponenti– U skladu sa datom specifikacijom zahteva, sprovedena je pretraga za neophodnim komponentama. Obično, ne postoji potpuna podesivost i komponente koje mogu biti korišćene obezbeđuju samo deo zahtevane funkcionalnosti.

2. Modifikacija zahteva– Tokom ove faze, analizirani su zahtevi korišćenjem informacija o komponentama koje su pronađene. Oni su zatim modifikovani da budu u skladu sa dostupnim komponentama. Kada su modifikacije nemoguće, aktivnost analize komponenti može biti ponovno urađena da bi se pronašla alternativna rešenja.

3. Projektovanje sistema sa komponentama- Tokom ove faze, projektovan je strukturni okvir (framework) sistema ili je korišćen postojeći okvir. Projektanti uzimaju u obzir komponente koje mogu da se ponovno koriste i organizuju okvir da bude prilagođen njima.

4. Razvoj i integracija– Razvijen je softver koji ne može biti kupljen i komponente i COTS sistemi su integrisani da stvore sistem. Integracija sistema, u ovom modelu, pre može biti deo procesa razvoja nego odvojena aktivnost.

* 1. Koje su prednosti korišćenja razvoja softvera baziranog na upotrebi komponenata? Koji su nedostaci korišćenja?

Model razvoja zasnovan na ponovnoj upotrebi komponenata ima očiglednu prednost da smanjuje količinu softvera koji treba da bude razvijen i takođe smanjuje troškove i rizike. To obično vodi ka bržoj isporuci softvera. Međutim, kompromisi zahteva su neizbežni i ovo može voditi ka sistemu koji neće u potpunosti ispuniti stvarne potrebe korisnika. Postoje tri tipa komponenti koje se koriste:

1. Veb servisikoji se razvijaju u skladu sa standardima za servise i koji se mogu aktivirati iz udaljene lokacije.

2. Kolekekcije objekata koji su razvijeni kao paketi radi kasnije integrisanja u sisteme , a u okviru komponentinih okvira, kao što su .NET i J2EE.

3. Samostalni softverski sistemikoji se konfigurišu za upotrebu u određenom okruženju. Razvoj zasnovan na ponovnoj upotrebi komponenata ima jasnu prednost u količini softvera koji treba razviti, jer je ona znatno smanjena, što se odražava na niže troškova i kraće vreme razvoja. Međutim, softverski sistem zbog upotrebe unapred definisanih komponenti, neminovno ne zadovoljava sve potrebe i zahteve korisnika. Takođe, gubi se i kontrola nad evolucijom softvera, sa upotrebom novih verzija komponenata, jer su one razvojene najčešće u drugim organizacijama.

* 1. Opišite glavne aktivnosti procesa razvoja softvera i dobijene rezultate ovih aktivnosti. Upotrebom dijagrama, pokažite relacije između izlaza iz ovih aktivnosti.
  2. Šta je specifikacija softvera? Opišite proces inženjeringa zahteva. Koje su četiri glavne faze procesa inženjeringa zahteva? Opišite svaku od ovih faza.

Specifikacija softvera treba da ustanovi koji servisi su zahtevani od strane sistema i ograničenja rada i razvoja sistema. Ova aktivnost se često zove inženjering zahteva.Inženjering zahteva je posebno kritična faza softverskog procesa zato što greške u ovoj fazi neizbežno vode kasnijim problemima u projektovanju i implementaciji sistema.

Ovaj proces vodi ka stvaranju dokumenta zahteva koji je specifikacija za sistem. Zahtevi su u ovom dokumentu obično predstavljeni u dva nivoa detalja. Krajnim korisnicima i mušterijama potreban je visoki nivo zahteva; inženjerima koji razvijaju sistem potrebna je jos detaljnija specifikacija sistema.

Postoje četiri glavne faze procesa inženjeringa zahteva:

1. Studija izvodljivosti– Procena je napravljena u skladu sa tim da li identifikovane potrebe korisnika mogu biti zadovoljene korišćenjem postojećeg softvera i hardverskih tehnologija. Na osnovu studije se donosi odluka da li je predloženi sistem troškovno-efektivan iz poslovne tačke gledišta i da li može biti razvijen poštujući postojeća ograničenja budžeta. Studija izvodljivosti trebalo bi da bude relativno jeftina i treba da bude brzo urađena. Rezultat studije izvodljivosti treba da pruži informaciju koja će odlučiti da li ići napred sa mnogo detaljnijom analizom.

2. Izvođenje i analiza zahteva– Ovo je proces izvođenja zahteva sistema preko posmatranja postojećih sistema, diskusije sa potencijalnim korisnicima, analize zahteva, itd. Ovo može obuhvatiti razvoj jednog ili više različitih modela sistema i prototipova. Ovo pomaže analitičarima u razumevanju sistema za koji treba da se uradi specifikacija.

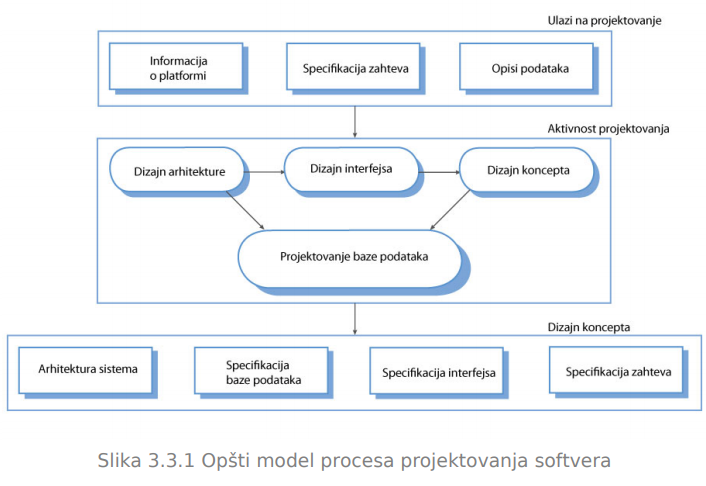
3. Specifikacija zahteva- Specifikacija zahteva je aktivnost prevođenja informacija skupljenih tokom analize u dokument koji definiše skup zahteva. U ovaj dokument mogu biti uključena dva tipa zahteva. Korisnički zahtevi su abstraktni iskazi zahteva sistema za mušteriju i krajnjeg korisnika; sistemski zahtevi su mnogo detaljniji opis funkcionalnosti koja treba da bude obezbeđena.

4. Validacija zahteva– Ova aktivnost proverava zahteve za realizmom, konzistencijom i potpunošću. Tokom ovog procesa, greške u dokumentaciji o zahtevima se neizbežno otkrivaju. Oni zbog toga moraju biti modifikovani da bi se korigovali ovi problemi.

* 1. Čime se bavi aktivnost softverskog procesa koja se bavi projektovanjem i implementacijom softvera. Nacrtajte model procesa projektovanja i implementacije softvera. Objasnite aktivnosti navedene u modelu.

Implementaciona faza razvoja softvera je proces pretvaranja specifikacije sistema u izvršni sistem. Ona uvek obuhvata procese projektovanja i programiranja softvera, ali ako je korišćen evolucionarni pristup razvoja, on može takođe obuhvatiti prečišćavanje specifikacije softvera.

Projektovanje softvera je opis strukture softvera koji treba da bude razvijen, podataka koji su deo sistema, interfejsa između komponenti sistema, i ponekad korišćenih algoritama. Projektanti ne stižu do konačnog projekta trenutno, već razvijaju projekat kroz različite verzije. Proces projektovanja obuhvata dodavanje formalnosti i detalja dok se razvija projektno rešenje, sa čestim vraćanjem unazad da bi se korigovali prethodne verzije.



* 1. Šta je validacija softvera? Nacrtajte dijagram procesa testiranja. Koje su tri osnovne faze testiranja? Opišite svaku od ovih faza.

Validacija (potvrđivanje) softvera treba da pokaže da li sistem zadovoljava svoju specifikaciju i očekivanje korisnika sistema. Za to, koristi se testiranje sistema sa simuliranim test podacima. Validacija može da obuhvati i procese provere (inspekcija i recenzija) svake faze softverskog procesa, počev od zahteva korisnika, pa do razvoja programa. Zbog dominantne aktivnosti testiranja, najveći troškovi validacije se vrše za vreme i posle implementacije sistema.

Ako se i moduli i pod-sistema smatraju jedinicama (komponentama softvera), onda se proces validacije može podeliti u tri osnovne faze:

1. Razvojno testiranje: Komponente sistema se posebno testiraju od strane onih koji su ih i razvili, u toku procesa njihovog razvoja. Svaka komponenta se posebno testira. Komponenta može biti vrlo jednostavna jedinica, koja obezbeđuje samo jednu funkciju, ali može biti i grupa ovakvih entiteta (npr. paket klasa). U ovoj fazi koriste se alati za automatsko testiranje, kao što je JUnit. 1. Sistemsko testiranje: Komponente sistema su integrisane i čine jedinstveni sistem. Testiranje sistema se vrši da bi se našle greške usled neodgovarajuće interakcije između komponenata i problema sa njihovim interfejsima. Testiranje sistema takođe treba da pokaže da li sistem ostvaruje sve funkcionalne i nefunkcionalne zahteve (npr. performanse). Kod velikih sistema (kao što je ovaj prikazan na slici), ovaj proces testiranja sistema se može izvršavati u fazama, jer se prvo testiraju integrisan jedinice u modeli, moduli u podsisteme, a na kraju, u ceo sistem.

2. Test prihvatanja: Ovo je krajnja faza procesa testiranja, pre njegovog prihvatanja za operativnu upotrebu. Sistem se testira korišćenjem podataka koje je obezbedio korisnik (kupac). Test prihvatanja može da ukaže greške koje potiču od neadekvatnih zahteva na sistemskom nivou, što testiranje sa simuliranim podacima nisu mogla da pokažu. Testiranje prihvatanja treba da pokaže da sistem zadovoljava sve zahteve korisnika i ostvaruje zahtevane performanse.

* 1. Kako se vrši testiranje u slučaju primene ekstremnog programiranja. Kako se vrši testiranja u slučaju planskog razvoja softvera? Nacrtajte faze testiranja u okviru modela procesa planskog razvoja softvera. Šta je alfa testiranje? Šta je beta testiranje?

U stvarnosti, razvoj komponenata i proces testiranja su izmešani. Programeri rade sopstvene testove sa svojim podacima i inkrementalno testiraju kod u toku njegovog razvoja. To je sa ekonomskog stanovišta ispravan pristup, jer programer najbolje poznaje komponentu koju je razvio, te je najbolja osoba i da generiše slučajeve (scenarije) za testiranja. U ekstremnom programiranju, testovi se razvijaju zajedno sa zahtevima, i pre nego što razvoj i počne. Ovo omogućava testerima i programerima da bolje razumeju zahteve i da obezbede da nema kašnjenja pri testiranju. Pri primeni softverskih procesa koji su vođeni planom (slika 2 ), testiranje se vrši u skladu sa planovima testiranja. Nezavistan tim testera pripreme te planove, a na osnovu dokumentacije sa specifikacijom sistema i projektnom rešenju. Slika 2 pokazuje kako planovi testiranja povezuju aktivnosti testiranja i razvoja.

Alfa testiranje vrši razvojni tim, sam, ili sa naručiocem softvera (kao test prihvatanja). Beta testiranje se vrši kod grupe odabranih korisnika sistema, tj. u njihovom radnom okruženju

Testovi prihvatanja se često nazivaju i „alfa testovima“. Alfa testiranje traje sve dok se naručilac softvera i razvojni tim sistema ne slože da razvijeni sistem zadovoljava postavljene zahteve.

U slučaju softverskih proizvoda (koji se prodaju na tržištu), koristi se i proces testiranja koji se naziva „beta testiranje“. Beta testiranje sistema se vrši kod grupe odabranih korisnika sistema, tj. u njihovom radnom okruženju. Oni šalju izveštaj razvojnom timu sistema. Na ovaj način se utvrđuju greške koje se javljaju u stvarnim uslovima rada sistema. Na osnovu ovih izveštaja, razvojni tim vrši promene u sistemu i proizvodi novu verziju ili za novo beta testiranje, ili za prodaju na tržištu.

* 1. Šta je evolucija softvera? Zašto dolazi do promena zahteva? Koka se mogu smanjiti troškovi evolucije softvera?

Softversko inženjerstvo je evolucionarni proces gde se softver kontinualno menja tokom njegovog životnog veka da bi mogao da odgovori na promene zahteva i potreba korisnika.

Fleksibilnost softverskih sistema je jedan od glavnih razloga zašto je sve više i više sofvera objedinjeno u velike, kompleksne sisteme. Jednom kada je doneta odluka da se napravi hardver, vrlo je skupo praviti promene u projektovanom hardveru. Međutim za softver promene mogu biti napravljene u bilo kom trenutku tokom ili nakon razvoja sistema. Ove promene mogu biti vrlo skupe, ali još uvek mnogo jeftinije u poređenju sa odgovarajućim promenama u hardveru sistema.

Istorijski gledano, uvek je postojala linija razgraničenja između procesa softverskog razvoja i procesa evolucije (održavanja) softvera. Razvoj softvera se odnosio na kreativnu aktivnost gde je razvijan softverski sistem od početnog koncepta preko radnog sistema. Održavanje softvera je proces promene sistema jednom kada je sistem krenuo sa radom.

Mada su troškovi održavanja softvera često nekoliko puta veći od početnih troškova razvoja, procesi održavanja se smatraju manje izazovnim nego razvoj originalnog softvera. Ova linija razgraničenja postala je skoro nebitna. Malo softverskih sistema su kompletno novi sistemi i ima mnogo više smisla razmatrati razvoj i održavanje kao kontinuum. Pre nego dva odvojena procesa, mnogo je više realno razmatrati softverski inženjering kao evolucionarni proces gde se softver kontinualno menja tokom njegovog životnog veka da bi mogao da odgovori na promene zahteva i potreba korisnika. Ovaj evolucionarni proces je prikazan na slici.

* 1. Zašto se koristi prototip softvera? Kada se koristi prototip? Nacrtajte model procesa razvoja prototipa. Objasnite faze razvoja prototipa datih u modelu procesa razvoja prototipa. Šta prototip treba da sadrži, a šta ne mora da sadrži? U kom obliku se predstavljaju prototipovi?

Prototip je početna verzija softverskog sistema koja se upotrebljava radi pokazivanja koncepata, probe projektnih opcija, i boljeg razumevanja problema i njegovih mogućih rešenja. Brz, iterativan razvoj prototipa sa niskim troškovima, je neophodan, kako bi korisnici softverskog sistema mogli da eksperimentišu sa prototipom u ranim fazama softverskog procesa.

Softverski prototip se koristi u procesu razvoja softvera da bi se olakšala primena zahtevane promene:

1. U procesu inženjeringa zahteva, prototip može da pomogne da se izmame i potvrde sistemski zahtevi.

2. U procesu projektovanja sistema, prototip se koristi radi ispitivanja određenog softverskog rešenja i radi podrške projektovanog korisničkog interfejsa.

Prototipovi sistema pomažu korisnicima da vide koliko dobro sistem podržava njihov rad. Tada mogu da im se jave nove ideje za zahteve, a i da utvrde područja u kojima je softver slab ili jak. Tada oni predlažu nove zahteve. Pored toga, primena prototipova može da ukaže na greške i nedostatke u zahtevima koje su predloženi. Funkcija definisana u dokumentaciji, sama za sebe, može izgledati da je prikladna. Međutim, ista funkcija, kada se kombinuje sa drugim funkcijama, može se pokazati kao neprihvatljivom ili nekorektnom. U tom slučaju se menja specifikacija sistema kako bi bolje odražavala promene zahteva.

Prototip sistema se može koristiti u fazi projektovanja sistema radi sprovođenja eksperimenata i radi ocenjivanja ostvarljivosti predloženog projektnog rešenja. Primena brzih metoda radzvoja radi provere korisničkog interfejsa, na primer, je često u upotrebi.

Prototipovi se najčešće koriste da bi se proverili i verifikovali zahtevi ili da bise testirala tehnologija koja će se koristiti pri razvoju softvera.

Ovde ćemo rezimirato do sada izloženo. Najšće se prototipovi koriste u dva slučaja pri razvoju softvera:

1. Utvrđivanje zahteva korisnika: Pokazivanjem prototipa softvera korisniku, daje mu se prilika da potvrdi d ali je ton što očekuje ili da dopuni svoje zahteve. Tada se često utvrđuju suprostavljeni i nejasni zahtevi

2. Procenjivanje tehničkih i arhitektonskih rizika: Pri projektovanju softvera se najčešće koriste projektni šabloni. Međutim, kod složenih softvera njihova primena nije uvek moguća. Primena prototipa je neophodna d abi se testirala ipostavljena arhitektura softverskog sistema. U slučaju primene novih tehnologija, na ovaj način se proverava njena upotrebljivost u slučaju softvera koji se razvija. Pri razzvoju softvera, prototipovi se koriste u dva oblika kao:

a) model procesa i kao

b) tehničko rešenje, tj. Tehnologija

* 1. Šta je inkrementalna isporuka softvera? Nacrtajte model inkrementalnog razvoja softvera. Koji su nedostaci inkrementalnog razvoja softvera? Kada inkrementalni pristup nije dobro rešenje?

Inkrementalna isporuka softvera je pristup u razvoju softvera koja omogućava da se kupcima softvera isporučuju razvijeni delovi sistema (inkrementi), koji se onda instaliraju i puštaju u operativnu upotrebu. Na taj način, kupci utvrđuju servise koje treba da im obezbedi sistem. Oni određuju prioritetne servise, a koji su im najmanje važni. Onda se definiše određeni broj inkremenata, tako da svaki od njih obezbeđuje određen podskup funkcija sistema. Prvo se razvijaju inkrementi (moduli) sa najvećim prioritetom.

Inkrementalna isporuka softvera nudi niz prednosti:

1. Kupci mogu da koriste prve inkremente kao prototipove i steknu iskustvo, koje koriste pri definisanju zahteva za kasnije inkremente. Za razliku od prototipova, to su delovi stvarnog sistema, te nema potreba da ponovo uče njegovo korišćenje, kao u slučaju korišćenja prototipova.

2. Kupci ne moraju da čekaju da se razvije ceo sistem da bi im bio isporučen, te ranije dobijaju korist od korišćenja delova sistema. Obično prvi inkrement repava njihove najvažnije zahteve.

3. Proces omogućava relativno lako ubacivanje novih promena u sistem. 4. Kako se prvo isporučuju inkrementi sa funkcijama najvećeg prioriteta, to oni i dobijaju i najdetaljnije testiranje, te taj, najvažniji deo sistema onda im najmanje zadaje probleme.

Međutim, inkrementalna isporuka softvera ima i nekih nedostataka:

1. Najveći broj sistema zahtevaju ispunjenje osnovnih funkcija, a koje obezbeđuju pojedini delovi sistema. Kako zahtevi nisu detaljno definisani sve dok se jedan inkrement ne primeni, teško je da se utvrde osnovne, zajedničke funkcije koje će biti realizovane od strane svih inkremenata.

2. Otežan je iterativni razvoj kada se razvija novi sistem. Korisnici žele celu funkcionalnost starog sistema i često nisu voljni da eksperimentišu sa nekompletnim novim sistemom. Zato, teško je dobiti korisne odzive korisnika sistema.

3. Srž iterativnog razvoja je u paralelizmu definisanja specifikacije i razvoja softvera. Međutim, to je u suprotnosti sa praksom ugovaranja i plaćanja kada se ugovara i plaća kompletan sistem. Kod inkrementalnog pristupa, nema specifikacije kompletnog sistema, sve dok se ne specifikacija ne pripremi i za poslednji inkrement. To zahteva novu formu ugovora, koje najčešće ne odgovaraju velikim državnim agencijama.

* 1. Šta je spiralni model razvoja softvera? Nacrtajte model spiralnog modela razvoja? Opišite sve sekcije u spirali tazvoja. Koji su rizici primene spiralnog modela razvoja softvera?

Spiralni model softverskog procesa (prikazan na slici 1 ) prvobitno je bio predložen od Boehma (1998) i sada je naširoko poznat. Proces je predstavljen kao spirala pre nego sekvenca aktivnosti sa povratnom spregom. Svaka petlja u spiralnom modelu predstavlja fazu procesa. Prema tome, krajnja unutrašnja petlja može se odnositi na izvodljivost sistema, sledeća petlja na definiciju zahteva sistema, sledeća petlja na projektovanje sistema, itd

Svaka petlja u spirali je podeljena u četiri sekcije kao što je prikazano na slici 3.7. Sekcije spiralnog modela su:

1. Definisanje ciljeva – Definisani su specifični ciljevi za ovu fazu projekta. Identifikovana su ograničenja procesa i proizvoda i skiciran je detaljni plan upravljanja. Identifikovani su rizici. Mogu biti planirane alternativne strategije zavisno od rizika.

2. Određivanje i redukcija rizika - Za svaki od identifikovanih projekata rizika, izvedena je detaljna analiza. Da bi redukovali rizik uvedeni su koraci. Na primer, ako postoji rizik za koji su zahtevi nezadovoljavajući, može biti razvijen prototip sistema.

3. Razvoj i validacija – Nakon proračuna rizika, izabran je razvojni model za sistem. Na primer, ako su rizici korisničkog interfejsa dominantni, odgovarajući razvojni model može biti evolucionarni prototip. Ako su rizici sigurnosti dominantni, može biti pogodan razvoj baziran na formalnoj transformaciji, itd. Model vodopada može biti najpogodniji razvojni model ako je glavni identifikovani rizik integracija pod-sistema.

4. Planiranje – Projekat je razmotren i doneta je odluka da li nastaviti sa narednom petljom spirale. Ako je doneta odluka da se nastavi, crtaju se planovi za narednu fazu projekta.

* 1. Objasni zašto je spiralni model prilagodljiv model koji može da podrži neophodne promene i zašto toleriše aktivnosti promena? Objasnite zašto se u praksi često koristi?
  2. Opišite glavne aktivnosti procesa razvoja softvera i dobijene rezultate ovih aktivnosti. Upotrebom dijagrama, pokažite relacije između izlaza iz ovih aktivnosti.

specifikacija zahteva, projektovanje softvera, programiranje i testiranje

* 1. Šta je RUP? Koje perspektive prikaza softverskog procesa podržava. Koje su četiri osnovne faze razvoja softvera po RUP-u? Objasnite svaku od ovih faza. Opišite radne tokove RUP-a. Za koje slučajeve je RUP pogodan, a za koje nije? Koja je razlika između faza i radnih tokova RUP-a?

RUP model se bavi analizom rizika i podržava razvoj koji primenjuje slučajeve korišćenja. Posebnu pažnju posvećuje arhitekturi softvera i ona je u centru njegove pažnje.

Rational Unified Process ili RUP je model procesa koji se bazira na inkrementalnom i iterativnom proces razvoja softvera. RUP model se bavi analizom rizika i podržava razvoj koji primenjuje slučajeve korišćenja. Posebnu pažnju posvećuje arhitekturi softvera i ona je u centru njegove pažnje.

RUP ie takođe i okvir za modeliranje procesa. Omogućuje prilagođavanje procesa potrebama korisnika (process customization) i omogućava definisanje procesa. Na taj načim, omogućava dobijanje različitih konfiguracija procesa. Te specifične konfiguracije omogućavaju:

• Podršku razvojnim timovima različite veličine (male, srednje i velike)

• Primenu vrlo formalizovanog ili malo formalizovanog procesa razvoja softvera, ttj. metoda razvoja softvera.

Da bi jasnije pozicionirali RUP u odnosu na bitne parametre koje klasifikuje različite pristupe u postavljanju proceesa razoja softvera, koristićemo dve ose sa po dve ekstremne vrednosti tih parametara

RUP model ima četiri faze softverskog procesa. Za razliku od modela vodopada, u kome su ove faze povezane sa aktivnostima, kod RUP modela, ove faze su povezane u skladu sa poslom, a ne tehnikom (slika). Naredna faza počinje kada se prethodna završi, ali u okviru svake faze postije više iteracija. Te faze su:

1. Početak :Cilj početne faze je postavljanje poslovnog scenarija (slučaja) sistema, tj. određivanja šta sistem treba da radi. Utvrđuju se spoljni akteri (ljudi i sistemi) koji su interakciji sa sistemom i definišu se te interakcije. Vrši se procena efekata rada sistema na poslovanje i procena rizika. Ako su efekti mali, može da dođe i do zaustavljanja projekta.

2. Razrađivanje (elaboracija): Cilj ove faze je razvoj razumevanja domena problema, postavljanje arhitektonskog okvira sistema (koncepcijsko rešenje arhitekture), razvoj plana projekta, i utvrđivanje ključnih rizika projekta. Na kraju ove faze, dobija se model zahteva sistema u vidu skupa UML slučajeva primene, opis arhitekture i razvijen plan projekta.

3. Konstrukcija: Faza konstrukcije obuhvata projektovanje sistema, programiranje i testiranje. Delovi sistema se paralelno razvijaju u ovoj fazi. Na kraju ove faze, sistem je u radnom stanju, zajedno sa pratećom dokumentacijom, te je spreman za isporuku korisnicima.

4. Tranzicija: Konačna faza RUP-a se bavi prenosom sistema iz razvojnog okruženja u korisničko okruženje, i stavlja ga u rad u stvarnom okruženju. Na kraju ove faze, sistem je opremljen kompletnom softverskom dokumentacijom i ispravno radi u operativnom (radnom) okruženju

1. **Lekcije 3 : MODELIRANJE STRUKTURE SISTEMA PRIMENOM UML** 
   1. Šta je modelovanje sistema? Šta je model? Zašto služe modeli? Koje vrsti modela se koriste u softverskom inženjerstvu? Zašto? Koje se perspektive modela koriste? Navedite i ukratko opišite glavne UML dijagrame. Koja su tri načina upotrebe grafičkih simbola pri modelovanju?
   2. Šta je konteksni model sistema?Šta on uzima u obzir? Dajte jedan primer. Koji su najčešći UML modeli interakcije? Za šta se koriste i kako se koriste UML slučajevi korišćenja? Opišite slučaj korišćenja sistema po Vašem izboru.

Kontekstni modeli pokazuju kako se sistem, koji se modeluje, uklapa u okruženje sa drugim sistemima i procesima. Oni pomažu da se definišu granice sistema koji se razvija.

* 1. Šta su sekvencijalni dijagrami? Dajte jedan primer. Kako se određuje stepen apstrakcije sekvencijalnog dijagrama.
  2. Strukturni modeli – navedite UML dijagram strukture. Zašto se oni koriste? Objasnite dijagram klasa. Dajte primer. Šta je generalizacija? Kako se obeležava u dijagramu klasa? Zašto se generalizacija koristi? D ajte primer generalizacije u vidu UML dijagrama klasa. Šta je agregacija? Kako se obeležava u UML?

1. **Lekcija 4: Modeliranje ponašanja sistema u UML**
   1. Šta su modeli ponašanja u UML? Opišite modele koji su vođeni podacima (dijagram aktivnosti, sekvencijalni dijagram…). Šta je modelovanje vođeno događajima? Opišite UML dijagrame stanja. Dajte primer. Koji su problemi kod modelovanja stanja?

Modeli ponašanja su modeli koji modeluju dinamičko ponašanje sistema kada radi. Oni pokazuju kako se sistem ponaša (reaguje) na neki podsticaj iz svog okruženja. Postoje dva tipa podsticaja:

• 3. Podaci: Dolazak podataka koje sistem treba da obradi.

• 4. Događaji: Neki događaj se desi i on aktivira obradu sistema. Događaji mogu da imaju pridodate podatke, ali to nije uvek slučaj.

Mnogi poslovni sistemi su vođeni podacima. Pri unosu podataka, oni počinju njihovu obradu, što podrazumeva izvršavanje definisanih akcija po rednom redosledu. (npr. priprema računa za telefonske pretplatnike). S druge strane, sistemi koji rade u realnom vremenu, su vođeni događajima, jer reaguju na razne događaje koji dolaze iz njihovog okruženja.

Svi sistemi koriste neki vid interakcije sa svojim okruženjem. To mogu biti interakcije sa korisnikom sistema, interakcije sa drugim sistemima ili interakcije između komponenata sistema. Modelovanje interakcija pomaže u utvrđivanju zahteva korisnika. Interakcija sa drugim sistemima može da ukaže na probleme komuniciranja sa drugim sistemima. Interakcije među komponentama sistema pomaže razumevanju rada sistema, i to u skladu sa zahtevanim performansama sistema i zavisnosti.

Sekvencijali dijagram (sequence diagram) je najčešća vrsta dijagrama interakcije koji se fokusira na razmenu poruka između određenog broja "linija života". Sekvencijalni idjagram opisuje interakciju fokusirajući se na redosled poruka koje se razmenjuju, zajedno sa njihovim odgovarajućim specifikacijama pojavljivanja na linijama života. Linja života (lifeline) je element koji predstavlja nekog individualnog.

Dijagram aktivnosti prikazuje tok upravljanja od aktivnosti do aktivnosti. Upotrebljava se za modeliranje dinamičkog aspekta sistema. Modelira se redosled koraka u procesu računanja (modeliranje operacije). Dijagramom aktivnosti se modelira tok menjanja stanja nekog objekta u različitim trenucima toka upravljanja (tok rada).

* 1. Šta je modelima vođeno inženjerstvo? Šta je modelima vođena arhitektura sistema? Zašto se koristi modelima vođeno inženjerstvo? Koji su argumenti protiv primene modelima vođenog inženjerstva? Navedite tri različita apstraktna modela i definišite ih. Prikažite dijagrame transformacije u slučaju modelom vođenim transformacijama. Šta je to izvršni UML? Šta je OCL?
  2. Opišite odgovarajućim UML modelima sve faze razvoja softverskog sistema za sledeće primere:
     1. Razvoj softverskog sistema za izradu rasporeda časova, ili
     2. Razvoj softverskog sistema za prikaz i naručivanje jela u restoranu (svaki sto ima touch screen računar sa prikazom jelovnika i izborom jela i pića.
     3. Razvoj sistema za odobrenje putnih naloga i za izveštavanje i obračun po obavljenom putu.
  3. Definišite UML slučajeve korišćenja, prateće sekvencijalne dijagrame i scenarije za softverski sistem dat sa sledećim opisom: *Sistem upravlja procesom realizacije službenih putovanja zaposlenih u firmi ABC. Sistem treba da obezbedi odobrenje za službeni put, izvore finansiranja puta, rezervacije karata, hotela i dr., izveštaj lica koje je išlo na službeni put, isplatu dnevnica, plaćanje računa za karte, hotel i dr., izradu konačnog finansijskog obračuna, i arhiviranje celokupne dokumentacije vezane za službeni put.*
  4. Nacrtajte sledeće UML dijagrame: a) Slučajevi upotrebe, b) Sekvencijalni c)Dijagrama klasa, d) Dijagram stanja za slučaj razvoja softverskog sistema koji treba da omogući naručivanje pice (ima ih 4 vrste) preko odgovarajućeg veb sajta, sa isporukom kući.
  5. Isključivo primenom detaljnog sekvencijalnog dijagrama, odredite atribute i metode neophodnih klasa za sledeći slučaj korišćenja sistema za rezervaciju hotelskog smeštaja: Korisnik želi da vidi u izabranom hotelu sve vrste hotelskih soba, njihovu cenu prenoćišta sa doručkom i raspoloživot u određenom terminu.

1. **Lekcija 5: INŽENJERSTVO ZAHTEVA**
   1. Šta su zahtevi za softver? Čime se bavi inženjerstvo zahteva? Kakvih vrsta zahteva ima? Objasnite ih detaljnije. Šta su funkcionalni zahtevi? Dajte primere. Šta su nefunkcionalni zahtevi. Dajte primere. Kako se pišu zahtevi? Šta je kompletnost, a šta konsistentnost zahteva?

Zahtevi za softverskim sistemom određuju šta sistem treba da radi i definišu ograničenja njegovog rada i implementacije.

Zahtevi za razvoj softverskog sistema su opisi onoga što treba sistem da radi. To su servisi koji obezbeđuju i ograničavaju svoju operaciju. Ti zahtevi odražavaju potrebe korisnika za sistemom koje služe za određene svrhe, kao što su: kontrola uređaja, postavljanje naloga, ili pronalaženje informacije.

Inženjerstvo zahteva je proces nalaženja, analize, dokumentovanja i provere ovih servisa i ograničenja.

Zahtevi korisnika su opšti apstrakti zahteva, a zahtevi sistema predstavljaju detaljne opise onoga što sistem treba da radi.

Funkcionalni zahtevi sistema opisuju šta bi sistem trebalo da radi. Korisnici ih definišu na apstraktan način, a inženjeri razvoja ih pretvaraju u detaljne zahteve (ulaz, izlaz, izuzeci…)

Nefunkcionalni zahtevi su zahtevi koji nisu direktno povezani sa servisima koje sistem treba da obezbedi svojim korisnicima, već definišu ograničenja implementacije sistema.

Kompletnost zahteva znači da su definisani svi servisi koje korisnik zahteva. Konsistentnost zahteva znači da nema kontradiktornosti među zahtevima.

Konzistencija – Zahtevi u dokumentu ne treba da se sukobljavaju. To je zato što ne treba da bude kontradiktornih ograničenja ili različitih opisa iste funkcije sistema. 3. Provera

* 1. Šta su funkcionalni zahtevi? Koji su specifični problemi definisanja funkcionalnih zahteva kod velikih sistema?

Funkcionalni zahtevi sistema variraju od opštih zahteva koji opisuju ono što bi sistem trebalo da radi, pa do vrlo detaljnih vrlo specifičnih zahteva, koji odražavaju lokalni način rada ili neki postojeće sisteme organizacije. Na primer, daju se primeri funkcionih zahteva MHC-PMS sistema:

1. Korisnik treba da ima mogućnost da izabere liste sastanaka za sve pacijente.

2. Sistem treba da izradi svakog dana, za svaku kliniku, listu pacijenata koji imaju zakazane pregleda tog dana.

3. Svaki zaposleni koji upotrebljava sistem mora da bude jedinstveno identifikovan sa svojim brojem sa osam cifara.

* 1. Šta su nefunkcionalni zahtevi? Zašto su oni važni? Navedite osnovne grupe nefunkcionalnih zahteva i opišite svaki. Koji se problemi javljaju kod utvrđivanja nefunkcionalnih zahteva? Koja je metrika nefunkcionalnih zahteva? Koje su teškoće u njenoj primeni?

Nefunkcionalni zahtev, kao što su performanse, bezbednost ili raspoloživost, često određuju ili ograničavaju karakteristike celog sistema. Zato su često i kritičniji nego funkcionalni zahtevi. Korisnik može da na drugi način dobije od sistema odgovor na funkciju koju sistem direktno ne obezbeđuje zbog loše definisanog funkcionalnog zahteva. Ali, ako je nefunkcionalna zahtev loše definisan, pa to čini ceo sistem sporim, ili nepouzdanim ili nebezbednim, onda se ceo sistem odbacuje i traži njegova popravka.

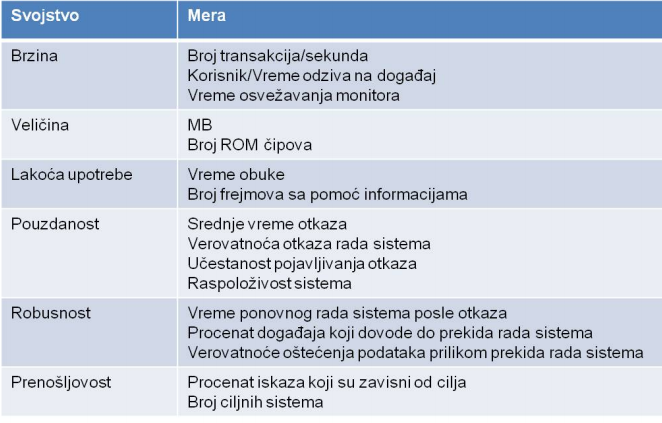
Osnovne vrste nefunkcionalnih zahteva su:

1. Zahtevi proizvoda: Ovi zahtevi opisuju ili ograničavaju ponašanje softvera (npr, performanse).

2. Organizacioni zahtevi: Ovi zahtevi su sistemski zahtevi koji su rezultat organizacionog ustrojstva kupca sistema (npr. poslovni procesi i standardi)

3. Spoljni zahtevi: Oni odražavaju zahteve koji dolaze iz spoljnjeg okruženja sistema (npr., propisi)

Metrika se može koristiti kod specifikacije nefunkcionalnih zahteva. Svuda gde je mogućno, pri definisanju nefunkcionalnih zahteva, potrebno je navesti kvantifikovane pokazatelje, koji se onda mogu testiranjem proveriti. Slika 3 pokazuje metriku koja se može koristiti kod specifikacije nefunkcionalnih zahteva. Navedene karakteristike se mogu meriti za vreme testiranja, te ime i proveriti da li sistem zadovoljava nefunkcionalne zahteve.



* 1. Šta je dokument sa zahtevima? Kako se ovaj dokument koristi kod različitih pristupa razvoju softvera? Ko su korisnici tog dokumenta i zašto? Kako se određuje stepen detaljnosti dokumenta sa specifikacijom zahteva? Koja je struktura dokumenta sa specifikacijom zahteva.

Dokument sa zahtevima za softver (ili specifikacija softverskih zahteva) je zvanični dokument koji razvojni tim treba da primeni. On uključuje i zahteve korisnika, i detaljnu specifikaciju zahteva za sistem. Ponekad i korisnički i sistemski zahtevi se navode u istom dokumentu, tj. sa istim opisom. Ponekad, se oni prikazuju u odvojenim dokumentima.

Dokument sa zahtevima je neophodan kao prilog ugovora po kom se angažuju organizacije koje su podizvođači u razvojnom projektu. Međutim, kod primene metoda agilnog razvoja, ovaj dokument se najčešće i ne koristi. Kod Ekstremnog programiranja, zahtevi se definišu inkrementalno i zapisuju na karticama kao korisničke priče ili scenariji. Kod poslovnih aplikacija, koje nemaju stabilne zahteve, to je i dobra praksa. Ipak, poželjno je i u tim slučajevima sačiniti dokument o poslovnim zahtevima i zavisnostima, i o tome voditi računa kada se zahtevi parcijalno i inkrementalno primenjuju u toku inkrementalnog razvoja sistema.

* 1. Kako se specificiraju zahtevi? Kako treba da budu napisani? Da li postoje veza između arhitekture softverskog sistema i sistemskih zahteva? Koje su mogući načini izražavanja sistemskih zahteva (notacija)? Koje su preporuke za specificiranje zahteva običnim, govornim jezikom?

Specifikacija zahteva je proces pisanja korisničkih i sistemskih zahteva u dokument sa zahtevima. U idealnom slučaju, ovi zahtevi bi trebalo da budu: jasni, nedvosmisleni, lako razumljivi, kompletni, i konsistentni. To je u praksi teško ostvarljivo jer zainteresovani akteri tumače zahteve na različite načine, te skoro uvek imamo suprostavljene zahteve i njihovu nekonsistentnost.

Zahtevi korisnika bi trebalo da opišu funkcionalne i nefunkcionalne zahteve, tako da su razumljiviji od strane korisnika softvera koji nemaju detaljna tehnička znanja. Idealno, ovi zahtevi bi trebalo da opišu samo spoljnje ponašanje sistema. Dokument sa zahtevima ne bi trebalo da uključi detalje o arhitekturi sistema niti o projektnom rešenju sistema. Zato se zahtevi korisnika pišu u običnom jeziku korisnika, sa jednostavnim tabelama, i intuitivnim dijagramima. Zahtevi sistema su proširene verzije zahteva korisnika i koriste se od softverkskih inženjera kao početna tačka projektovanja sistema.

Zahtevi sistema dodaju detalje i objašnjavaju kako sistem treba da realizuje zahteve korisnika.

Zahtevi sistema, pored korišćenja običnog jezika, mogu da koriste i druge načine pisanja, kao što su grafički modeli ili matematički sistemi, a da bi detaljnije objasnile sistemske zahteve.

Za razliku od zahteva korisnika, zahtevi sistema, pored korišćenja običnog jezika, mogu da koriste i druge načine pisanja, kao što su grafički modeli ili matematički sistemi, a da bi detaljnije objasnile zahteve koje sistem treba da ispuni. Grafički modeli su od posebne pomoći kada se treba objasniti kada sistem treba da promeni svoje stanje ili kada se treba navesti neki redosled akcija koje sistem treba da realizuje. Unified Modelling Language (UML) se u ove svrhe najčešće koristi. Matematički sistemi se ponekad koriste kod bezbednostno kritičkih sistema, ali se retko koriste u drugim slučajevima.

* 1. Kako se specificiraju zahtevi primenom strukturnog jezika? Da li se može koristiti i programski jezik? Kako se struktuiraju sistemski zahtevi? Dajte jedan primer. Dajte tipičan sadržaj standardizovanog formulara za specifikaciju zahteva. Zašto se primenjuje i strukturiran način definisanja zahteva?

Strukturisana specifikacija koristi formulare (uzorke) za pisanje zahteva sistema.

Strukturisan običan jezik je način pisanja zahteva koji ograničava slobodu pisanja pisca zahteva i kod koga se svi zahtevi pišu na jedan standardizovani način. Na ovaj način se zadržava izražajnost i razumljivost običnog jezika, ali se obezbeđuje i da određeni nivo uniformnosti specifikacije. Strukturisana specifikacija koristi formulare (uzorke) za pisanje zahteva sistema. Specifikacija može da koristi i iskaze u programskom jeziku da bi se prikazale alternative i iteracije, a može i da označi ključne delove teksta podvlaćenjem zadebljanjem slova, promenom fonta (tip slova) u delu teksta ili senčenjem.

Može se koristiti jedan ili više standardnih formulara za definisanje sistemskih zahteva. Oni se mogu strukturirati prema sistemu, funkciji koju sistem ostvaruje, ili prema događaju koju obrađuje sistem.

* 1. Koje osnovne aktivnosti proces inženjeringa zahteva? Dajte spiralni pogled na proces inženjerstva zahteva i objasnite ga. Opišite metod slučajeva upotrebe u UML za definisanje zahteva.

Otkrivanje i analiza zahteva je jedan iterativan proces koji se može predstaviti spiralom aktivnosti: otkrivanje zahteva, klasifikacija i organizacija zahteva, pregovaranje oko zahteva, i dokumentovanje zahteva.

Procesi inženjerstva zahteva čine četiri osnove aktivnosti:

1. Studija izvodljivosti – ocena da li je proces koristan za poslovanje,

2. Prikupljanje i analiza zahteva – otkrivanje zahteva

3. Specifikacija zahteva – konverzija zahteva u standardni oblik

4. Validacija – provera da zahtevi odražavaju sistem koji kupac/korisnik želi

U praksi, inženjerstvo zahteva je jedan iterativan proces, a ne strogo redni (sekvencijalan). Na slici je prikazan ovaj proces kao iterativni proces u vidu spirale, sa konačnim izlazom u vidu dokumenta sa specifikacijom zahteva sistema. U svakoj vremenskoj fazi troši se u svakoj aktivnosti odgovarajući rad i vreme.

U početku se najviše vremena troši na razumevanje glavnih poslovnih i nefunkcionalnih zahteva, kao i zahteve korisnika sistema. Spiralni model inženjerstva zahteva da se uzime u obzir različit nivo detaljnosti zahteva. Sa povećanjem broja iteracija oko spirale, i broj i detaljnost zahteva se povećava. Ako se koristi agilni pristup razvoju, umesto izrade prototipa, zajedno se radi utvrđivanje zahteva i implementacija sistema.



* 1. Zašto softverski inženjeri treba da razgovaraju sa korisnicima ili kupcima softvera. Šta su akteri softverskog sistema? Koji su ostali akteri sistema, sa kojima treba razgovarati u procesu prikupljanja zahteva? Nacrtajte model prikupljanja i analize sistema, i objasnite ga, tj. njegove aktivnosti. Koje se poteškoće javljaju pri radu sa akterima sistema? Kako olakšati prikupljanje zahteva? Pored razgovora sa akterima, koji su drugi izvori informacija značajnih za definisanje zahteva?

Sistemski akteri često ne znaju da definišu zahteve dovoljno detaljno i jasno. Akteri sistema definišu zahteve koristeći svoje termine i svoje implicitno znanje. Možda inženjeri razvoja, ne razumeju te termine. Različiti akteri imaju različite zahteve i izražavaju ih na različite načine. Inženjer koji prikuplja zahteve mora da otkrije sve potencijalne izvore zahteva i da otkrije i konfliktne zahteve.

(MHC-PMS sistem), akteri sistema su: • pacijenti, • doktori koji leče pacijente, • medicinske sestre, koje administriraju neke terapije i koordinišu konsultacije sa doktorima, • zdravstveni recepcionisti, koji zakazuju preglede pacijenata, • IT inženjeri i tehničari, koji instališu i održavaju sistem, • menadžer za medicinsku etiku, koji se stara o poštovanju etičkih principa u radu sa pacijentima, • menadžeri u zdravlju koji koriste informacije iz sistema

* 1. Zašto služe intervjui u procesu prikupljanja zahteva, i koje dve vrste intervjua postoje? Za šta su intervjui dobri, a za šta nisu? Koje karakteristike treba da ima lice koje obavlja intervju (intrevjuer)?

U intervjuima, tim inženjera postavlja pitanja akterima o sistemu koji oni trenutno koriste i o sistemu koji treba da se razvije. Analizom dobijenih odgovora, utvrđuju se zahtevi.

Najveći broj zahteva se prikupi korišćenjem formalnih ili neformalnih intervjua sa akterima sistema. U njima, tim inženjera postavlja pitanja akterima o sistemu koji oni trenutno koriste i o sistemu koji treba da se razvije. Analizom dobijenih odgovora, utvrđuju se zahtevi.

Postoje dve vrste intervjua:

1. Zatvoreni intervjui, kada akteri odgovaraju na unapred pripremljena pitanja.

2. Otvoreni intervjui, kada ne postoje unapred definisana pitanja.

U praksi, intervjui su najčešće mešavina obe vrste intervjua. Unapred pripremljena pitanja obezbeđuju širinu prostora za prikupljanje informacija, a dobijeni odgovori daju priliku za postavljanje dodatnih pitanja, kako bi se utvrdile dodatne i nove informacije.

* 1. Šta su scenariji i čemu služe u procesu prikupljanja i specificiranja zahteva? Koji su osnovni elementi jednog scenarija? U kojim formama se može opisati scenario? Šta je UML slučaj korišćenja? Kako se on opisuje? Dajte jedan primer. Kako se scenariji koriste u vezi sa UML slučajevima upotrebe? Da li se sekvencijalni dijagram može koristiti za grafičko predstavljanje scenarija slučaja upotrebe? Dajte jedan primer.

Scenariji su opisi interakcije aktera i sistema, tj. interaktivnih sesija. Svakki scenario obično pokriva jednu ili mali broj mogućih interakcija. Postoje različite forme scenarija i one obezbeđuju različite tipove informacija na različitim nivoima detalja opisa sistema. Primena scenarija može biti posebno korisna u cilju dodavanja detalja prethodno definisanih zahteva.

Ako osoba koja vodi intervju, prikaže intervjuisanoj osobi scenarijo u kome opisuje očekivanu interakciju aktera sa sistemom, intervjuisana osoba će mnogo lakše izneti svoje sugestije i zahteve, nego ako se razgovor vodi apstraktno, bez početne skice scenarija. Inženjer zahteva (osoba koja vodi intervju) na taj način prikupi korisne informacije koje mu pomažu da onda formuliše zahteve.

Scenario počinje skicom jedne interakcije aktera sa sistemom. Za vreme otkrivanja zahteva, vrši se dodavanja detalja da bi se dobila kompletan opis te interakcije. Jedan scenario, sadrži sledeće elemente:

1. Opis šta sistem, i korisnici očekuju kada počne scenario.

2. Primarni scenario: Opis normalnog toka događaja u scenariju.

3. Sekundarni scenario: Opis šta može da ide pogrešnim tokom i kako taj problem razrešiti.

4. Informacija o drugim aktivnostima koje mogu da se istovremeno izvršavaju.

5. Opis stanja sistema kada se scenario završi.

* 1. Šta je etnografija? Kako se ona koristi za otkrivanje zahteva? Kako se etnografija može povezati sa korišćenjem prototipova softverskih sistema? Kada je posebno preporučljivo koristiti etnografiju?

Softverski sistemi ne rade u izolaciji. Oni se koriste u određenom socijalnom i organizacijskom kontekstu i oni moraju da budu izvedeni iz takvog konteksta ili njime ograničeni. Da bi softverski sitsem uspešno radio, on treba da zadovolji te društvene i organizacijske zahteve. Ima slučajeva da se neki softverski sistem ne koristi, iako je razvijen u skladu sa zahtevima. U tim slučajevima, nisu uzeti u obzir društveni ili organizacijski zahtevi.

Etnografija je tehnika osmatranja koja se koristi radi razumevanja operacionih procesa Analitičar svakodnevno osmatra i beleži stvarne zadatke koji učesnici rade. Na taj način se mogu otkriti i implicitni zahtevi koji odražavaju stvarni način rada, a ne formalni proces koja je definiše organizacija.

Neki ljudi rade posao, ali ne umeju da objasne kada i kako rade. Naročito, ne umeju da taj rad povežu sa društvenim i organizacijskim zahtevima. Osmatranje njihovog rada pomaže u otkrivanju ovih zahteva.

Etnografija je naročito korisna u otkrivanju dve vrste zahteva:

1. Zahtevi koji proističu iz načina kako ljudi stvarno rade (često i kršeći neke propise).

2. Zahtevi koji proističu iz kooperacije i aktivnosti drugih ljudi.

3. Etnografija se može kombinovati sa izradom prototipova (slika), jer ukazuje na vrstu prototipova koji su potrebni, te time i smanjuje njihov neophodan broj.

Etnografske studije mogu otkriti detalje kritičkih procesa koji često nedostaju primenom drugih tehnika nalaženja zahteva. Etnografija, se mora kombinovati sa drugim tehnikama otkrivanja zahteva, jer ona nije u stanju da otkrije organizacijske i domenske zahteve, a ponekad ne može da ukaže na neophodne nove zahteve.

* 1. Šta je validacija zahteva? Kako validacija utiče na troškove razvoja sistema? Kako se vrši validacija (provera zahteva) dokumenta koji specificira zahteve? Koje se tehnike validacije zahteva primenjuju?

Validacija zahteva provera da li zahtevi stvarno definišu sistem koji kupac želi. Validacija zahteva je važna zato što greške u dokumentu zahteva mogu da vode velikim troškovima prepravke, kada su otkriveni tokom razvoja ili nakon što je sistem pušten u rad. Trošak realizovanja promene sistema koji je uzrokovan problemom zahteva je mnogo veći od popravke projekta ili grešaka kodiranja. Razlog za ovo je da promena zahteva obično znači da projektovanje i implementacija sistema moraju da budu promenjeni i da sistem mora biti testiran ponovo.

Tokom procesa validacije zahteva, različiti tipovi provere treba da budu izvedeni na zahtevima u dokumentu zahteva. Ove provere uključuju:

1. Provera validacije – Korisnik smatra da je neophodno da sistem ima neke funkcije. Međutim, buduća razmišljanja i analize mogu identifikovati dodatne ili različite funkcije koje su potrebne. Sistem ima različite korisnike sa različitim potrebama i neki skup zahteva je neizbežan kompomis u toku komunikacije sa korisnikom.

2. Provera konzistencije – Zahtevi u dokumentu ne treba da se sukobljavaju. To je zato što ne treba da bude kontradiktornih ograničenja ili različitih opisa iste funkcije sistema.

3. Provera kompletnosti – Dokument zahteva treba da uključi zahteve koji definišu sve funkcije i ograničenja planirana od strane korisnika.

4. Provera realizma – Korišćenjem znanja postojećih tehnologija, zahtevi treba da budu provereni da bi se obezbedilo da li mogu stvarno da budu implementirani. Ove provere takođe treba da uzmu u obzir raspoloživi budžet i vreme potrebno za razvoj sistema.

5. Verifikacija – Da bi smanjili potencijalne rasprave između korisnika-nručioca softvera i izvođača, zahtevi sistema treba uvek da budu pisani tako da mogu da budu provereni. Ovo znači da može biti projektovan skup provera koji može da demonstrira da li je isporučeni sistem zadovoljio zahteve.

Tehnike validacije su pregledi zahteva od strane recenzenta, korišćenje prototipa kao modela za proveru definisanih zahteva, i korišćenje testova povezanih sa svakim zahtevom.

Postoji određen broj tehnika validacije zahteva koje mogu biti korišćene zajedno ili pojedinačno:

1. Pregledi zahteva – Zahtevi su sistematski analizirani od strane tima ocenjivača.

2. Prototip – U ovom pristupu validaciji, izvršni model sistema prikazan je krajnim korisnicima i mušterijama. Oni mogu eksperimentisati sa ovim modelom da bi videli da li zadovoljava njihove realne potrebe.

3. Stvaranje test slučaja – Idealno, zahtevi treba da budu tako urađeni tako da mogu da budu testirani. Ako su testovi za zahteve smišljeni kao deo procesa validacije, ovo često otkriva probleme zahteva. Ako je test težak ili nemoguć za projektovanje, ovo obično znači da će biti teško implementirati zahteve i da treba da budu ponovo razmatrani.

* 1. Čime se bavi upravljanje zahtevima? Koji je glavni izazov? Zašto dolazi do promena zahteva? Šta je evolucija zahteva? Zašto se javljaju novi zahtevi i posle puštanja sistema? Šta je upravljanje zahtevima? Koje odluke treba doneti da bi se planski upravljalo zahtevima? Kakva se automatizovana podrška može dati aktivnostima upravljanja zahtevima? Kako upravljati promenama zahteva? U čemu je prednost primene formalizovanog procesa upravljanja zahtevima? Koje su faze procesa upravljanja promenama zahteva? Do kakvog se problema može doći ubacivanjem stalno novih zahteva? Koja je specifičnost promene zahteva kod agilnog razvoja softvera?

Upravljanje zahtevima je proces razumevanja i kontrole promene u zahtevima sistema

Zahtevi kod velikih softverskih sistema se stalno menjaju. Jedan od razloga za ovo je u nemogućnost sagledavanja svih zahteva kod ovih sistema. Po pravilu, zahtevi za ove sisteme nisu potpuno definisani, te su nekompletni. Za vreme softverskog procesa (tj. procesa razvoja softvera), razumevanje problema kod aktera sistema se stalno menja. Zbog toga, i zahtevi sistema mora stalno da evoluiraju, da bi odražavali promene u pogledu na problem.

Kada se novi sistem pusti u upotrebu, neminovno se jave novi zahtevi. Teško je krajnji korisnici i akteri sistema mogu da sagledaju efekte novog sistema na njihove poslovne procese i na način obavljanja posla. Posle sticanja prvih iskustava sa novim sistemom, krajnji korisnici sistema počinju da otkrivaju nove potrebe i prioritete. Za to postoji nekoliko razloga:

1. Poslovno i tehničko okruženje sistema se uvek menja posle instalacije sistema (novi hardver, novi interfejsi, novi poslovni prioriteti, novi propisi).

2. Ljudi koji naručuju ili biraju sistem i ljudi koji koriste sistem, najčešće nisu isti. Kupci sistema postavljaju zahteve zbog organizacijskih i finansijskih ograničenja. To može da bude u sukobu sa zahtevima korisnika sistema, te se pokaže potreba da se posle instalacije moraju da dodaju novi korisnički zahtevi.

3. Veliki sistemi obično imaju šaroliku zajednicu korisnika, sa različitim zahtevima i prioritetima, koji mogu biti i međusobno suprotstavljeni. Konačan sistem je onda obično neki kompromis između tih zahteva, a praksa pokaže da ipak mora pomoći nekim drugim korisnicima, tj. da treba udovoljiti njihovim zahtevima.

1. **Lekcija 6: Analiza i modelovanje softverskog sistema**
   1. Šta je model softverskog sistema? Kada se koriste modeli softverskog sistema? Navedite najčešće korišćene UML dijagrame i navedite njihovu svrhu.

Model softverskog sistema koji se razvija u toku analize je ovde nazvan modelom analize. Ona nastaje iz modela zahteva, pri čemu se vrše sledeće transformacije:

1. Jezik klijenta se zamenjuje jezikom inženjera softvera

2. Umesto spoljnje pogleda na sistem, on sadrži unutrašnji pogled.

3. Umesto opisa funkcionalnosti, on sadrži način realizacije te funkcionalnosti

4. Umesto dogovora sa klijentom, on se bavi realizacijom funkcija

5. Nekonsistentnost i ponavljanja se ostranjuju iz modela

* 1. Šta obezbeđuje model analize sistema? Koja se polazna dokumenhta koriste za analizu?

Model analize za svaki poslovni paket obezbeđuje objekte klasa analize i slučajeve korišćenja, opisane i sekvenciojalnim i VOPC dijagramima. Pri izradi modela analize, prvo što treba uraditi je da se vidi KAKO će sistem realizovati (implementirati) specifikaciju zahteva, pri čemu analitičari koriste sledeće ulazna dokumenta:

• dataljno opisane slučajeve korišćenja (primare i sekundarne scenarije)

• model slučajeva korišćenja

• model iskustva korisnika

• model poslovnih entiteta Model analize ima sledeću strukturu:

1. Folder apsrakcija: Za svaki poslovni paket se utvrđuju:

• Objekti granica (Boundary)

• Objekti kontrole (Control)

• Objekti entiteta (Entity)

2. Folder realizacije slučajeva korišćenja: Za svaki poslovni paket se navode svi slučajevi korišćenja, od kojih su, oni glavni, opisani i sa sekvencijalnim dijagramima, a svi i sa po jednim VOPC dijagramom (:View Of Participating Classes).

* 1. Koje su tri klase analize sistema? Objasni njihovu ulogu.

Granične (Boundary) klase, klase kontrole (Controller) i klase entiteta (Entity) su klase koje su glavni elementi modela analize, jer definišu ponašanje celog softverskog sistema.

**Objekti klasa granice i kontrole** se relativno lako mugu utvrditi na osnovu analize slučajeva korišćebnja. Međutim, objekti klase entiteta (Entity) se teže utvrđuju, te je potreno iskustvo u radu na analiyai domena.Granične klase (Boundary) predstavljaju objekte sa kojima akteri komuniciraju. Na primer, meniji, forme, diijalozi ili interfejsi sa spoljnim sistemima. Granične klase su uglavnom odgovorni za komunikaciju sa korisnicima. One ne uključuju logiku obrade njihovih zahteva. Preporučuje se kreiranje po jedne granične klase za svakog aktera.

**Klase kontrole** (Controll) predstavljaju objekte koji koordiniraju aktivnosti objekata entiteta i komuniciraju sa graničnim objektima, i time određuju ukupno ponašanje softverskog sistema. Odgovornost objekta kontrole je najčešće povezana sa realizacijom jednog slučaja korišćenja. Objekti kontrole odvajaju granične objekte o objekata entiteta, što omogućuje relativno laku promenu korisničkog interfejsa i logike softvera. Obično, svaki slučaj korišćenja se realizuje posebnim objektom kontrole, mada se u nekim slučajevim logika programa realizuje i bez objekata kontrole. U takvim slučajevima, funkciju koordinacije aktivnosti preuzimaju granični objekti i objekti entiteta.

**Klase entiteta** (Entity) prestavljaju objekte koji sadrže informacije i podatke, kao što su relacione tabele podataka ili fajlove s apodacima. Oni skladište podatke i realizaju neke osnovne operacije sa njima. Međutim, po pravilu, ne sadržđe poslovnu logiku obrade podataka. To je funkcija objekata kontrol.

* 1. Šta je inženjerstvo sa modelima? Šta je OCL i čemu on služi? Dajte jedan primer priimene OCL.

Inženjerstvo sa modelima (engl. Model-driven engineering ili MDE) je pristup razvoju softvera u kome su modeli, a ne program, glavni rezultat procesa razvoja. Programski kod se automatski generiše iz modela. Podržavaoci ovog pristupa tvrde da se na ovaj način podiže nivo apstraktnosti u softverskom inženjerstvu tako da inženjer razvoja ne mora više da vodi računa o detaljima programskog jezika ili o specifičnostima računarske platforme koja izvršava program.

OCL - Dinamičko ponašanje sistema se može deklarativno opisati upotrebom jezika ograničenih objekata (engl. Object Constraint Language – OCL) ili se može koristiti jezik akcija UML-a.

Modelima vođeno inženjerstvo polazi od pretpostavke da je potpuno automatsko prevođenje modela u izvršni kod moguće. Ovo zahteva vrlo semantički bogate grafičke modele, jer mora da sadrže i informaciju kako bi trebalo da se primeni projektovani sistem. UML 2.0 to omogućava primenom dela koji se naziva Izvršni UML ili xUML (engl. Executable UML). Originalni UML dijagrami nisu pravljeni da bi podržali programske jezike. Nedostaje im neophodna semantika. Da bi se načinio UML koji može da se koristi i za prevođenje u programski jezik, broj mogućih modela je smanjen na samo tri:

1. Domenski modeli utvrđuju osnovne zahteve sistema. Oni se realizuju uz pomoč dijagrama klasa, uključujuti i objekte, atribute i asocijacije.

2. Model klasa, u kome se definišu klase, zajedno sa atributima i operacijama.

3. Modeli stanja, koji imaju dijagrame stanja povezane sa svakom klasom i koriste se da bi se opisao životni ciklus neke klase.

Primer - Poruka pisana u OCL u formi OclMessage instrukcije, se izvršava aktiviranjem operacije getNextPieceOfData predstavljena na intefejsu klase GarbageCan.

* 1. Šta je VOPC dijagram (View Of Participating Classes)? Dajte primer jednog VOPC dijagrama.

VOPC je skračenica engleskog naziva View Of Participating Classes. VOPC je prikaz objekata sistema koji rade zajedno na realizaciji jednog slučja korišćenja. To su objekti koji pripadaju apstraktnim klasama modela analize, a koje se odnose na jedan slučaj korišćenja.

VOPC, dakle, pokazuje sve objekte koji zajedno rade na realizacije jednog slučaja korišćenja, kako u primarnom, tako i u sekundarnim scenarijima slučaja korišćenja.

VOPC koristi jednostavne asocijacije (bez oznaka smera) između klasa koje sadrže.

1. **Lekcija 7: PROJEKTOVANJE ARHITEKTURE SOFTVERSKOG SISTEMA**
   1. Šta je projekat (dizajn) arhitekture? Šta je model arhitekture sistema? Kada se definiše arhitektura sistema? Zašto je arhitektura važna? Šta omogućava arhitektura sistema? Kako se pravi model arhitektura? Dajte jedan primer. Za koje svrhe se koristi model arhitekture sistema?

Arhitektura softvera ukazuje nam kako bi sistem trebalo da bude organizovan i prikazuje ukupnu strukturu sistema. Projektovanje arhitekture je prva faza procesa projektovanja softvera. To je kritična veza između inženjerstva zahteva i projektovanja softvera, jer utvrđuje glavne strukturne komponente u sistemu i veze između njih. Na izlazu iz procesa projektovanja arhitekture se dobija model arhitekture koji opisuje kako je sistem organizovan kao skup komunikacionih komponenti.

* 1. Objasni zašto je potrebno da se projektuje arhitektura sistema pre nego što je završena specifikacija zahteva

Dobra arhitektura omogućava dobru komunikaciju sa drugim sistemima i korisnicima, odgovarajuću anlizu sistema i višestruku upotrebljivost i u drugim softverima.

U idealnom slučaju, specifikacija sistema ne bi trebalo da sadrži informacije o projektnom rešenju softvera. U praksi, sem kod vrlo malih sistema, to se ne može ostvariti. Najčešće je potrebno da se već na početku izvrši dekompozicija arhitekture budućeg sistema da bi se dobila strukturisana i organizovala specifikacija. Zato, već u fazi utvrđivanja zahteva, može se predložiti apstraktna arhitektura sistema, da bi se povezala svojstva sistema sa najvećim i najbitnijim komponentama sistema. Tako predstavljena specifikacija funkcija i zahteva, se onda diskutuje sa akterima sistema.

* 1. Objasni proces projektovanja arhitekture. Na koja pitanja treba naći odgovore u toku procesa projektovanja arhitekture? Šta su arhitektonski šabloni? Šta oni sadrže? O kojim nefunkcionalnim zahtevima se mora voditi računa prilikom izboru arhitektonskog šablona? Kako se ocenjuje arhitektura sistema?

Projektovanje arhitekture softvera primenjuje dva nivoa apstrakcije:

1. Arhitektura u malom – odnosi se na arhitekturu posebnih programa, tj. na strukturu i komponente samog programa.

2. Arhitektura u velikom – odnosi se na arhitekturu složenog organizacijskog sistema koji uključuje druge sisteme, programe i programske komponente. Ovo je u stvari arhitektura distribuiranih sistema, što se na Univerzitetu Metropolitan izučava na posebnom predmetu. Arhitektura softvera je važna jer utiče na performanse, robusnost, distrubutivnost i održivost sistema. Dok komponente sistema ostvaruju funkcionalne zahteve sistema, nefunkcionalni zahtevi najviše zavise od arhitekture sistema, jer ona određuje organizaciju tih komponenti i njihovu međusobnu komunikaciju.

Posedovanje projektnog rešenja arhitekture softvera obezbeđuje sledeću povoljnost:

1. Komunikacija sa akterima sistema: Arhitektura daje pregled celine sistema i dobro je sredstvo za komunikaciju sa različitim akterima sistema.

2. Analiza sistema: Izrada arhitekture sistema u ranim fazama razvoja zahteva sprovođenje odgovarajućih analiza na nivou sistema. Zato, odluke vezane za projektno rešenje arhitekture imaju dubok efekat na to da li će sistem zadovoljiti kritičke zahteve, kao što su performanse, pouzdanost i održivost.

3. Višestruka upotrebljivost: Model arhitekture sistema je kompaktan, upravljiv opis kako je sistem organizovan i kako komponente rade zajedno. Arhitektura sistema je često ista za sisteme sa sličnim zahtevima, te se mogu koristiti više puta, kod drugih sličnih sistema.

**Arhitektonski** **šabloni** (engl., architecural pattern) su opisi organizacije sistema, kao što su arhitektura klijent-server, ili višeslojna arhitektura. Arhitektonski šabloni sadrže bitna arhitektonska rešenja koja su upotrebljena kod različitih sistema. Preporučljivo je da se njihova eventualna upotreba razmotri na početku projektovanja arhitekture sistema.

* 1. Objasni zašto se mogu javite konflikte situacije pri projektovanju arhitekture u kojima su zahtevi za raspoloživost i bezbednost najvažniji nefunkcionalni zahtevi. Dajte jedan konkretan primer.

Pri izboru stila arhitekture i strukture sistema, treba voditi računa o nefunkcionalnim zahtevima sistema:

1. Performanse: Ako su perfomanse bitne za sistem, onda arhitektura treba da obezbedi lokalizaciju kritičnih operacija unutra malog broja komponenti, u okviru istog računara. To znači da je bolje koristiti veliku umesto male komponente da bi se smanjila komunikacija između komponenti, koja odnosi dosta vremena.

2. Bezbednost: Ako je bezbednost bitna, višeslojna arhitektura je dobro rešenje, da bi se najkritičnije vrednosti stavile pod kontrolu u unutrašnjim slojevima sistema, sa visokom dozom zaštite.

3. Zaštita: Ako je zaštita od značaja, onda arhitektura treba da obezbedi da se zaštićene operacije obavljaju u samo jednoj komponenti ili u vrlo malom broju komponenti. To smanjuje troškove zaštite i primenu sistema zaštite koji mogu da isključe sistem u slučaju nekog kvara.

4. Raspoloživost: Ako je raspoloživost kritičan zahtev za sistema, onda arhitektura treba da bude tako projektovana da uključuje redundantne (ponovljive) komponente koje mogu jednadrugu da zamene u u slučaju da jedna otkaže.

5. Lakoća održavanja: Ako je lakoća održavanja kritičan zahtev, arhitekturu sistema treba projektovati sa malim, samodovoljnim komponentama koje se lako zamenjuju. Proizvodnja podataka se odvaja od korisnika a izbegava se deljivost struktura podataka.

Pri projektovanju arhitekture, pojedini zahtevi vode ka konfliktnoj situaciju prilikom projektovanja arhitektura. Na primer, velike komponente poboljšavaju performanse, a male olakšavaju održavanje. Ako su oba zahteva važna, mora se onda tražiti kompromisno rešenje. Ponekad, za pojedine delove sistema se onda, koriste različiti stilovi arhitekture.

Ocena arhitekture se može dobiti i njenim upoređenjem sa referentnim arhitekturama (ranije primenjenim i provereno dobrim) ili sa opštim arhitektonskim šablonima (uzorima). Konačnu ocenu arhitekture daće ocena rada celog sistema, i zadovoljenje svih funkcionalnih i nefunkcionalnih zahteva.

* 1. Šta je arhitektonski pogled? Koji su fundamentalni (osnovni) arhitektonski pogledi? Objasnite ukratko svaki od njih. Pored fundamentalnih pogleda, koristi se i koncepcijski pogled. Šta je to? Da li je detaljna specifikacija arhitekture potrebna?

Svaki model pokazuje samo jedan pogled ili perspektivu sistema. Korisno je predstaviti više pogleda softverske arhitekture kada se različite informacije o sistemu traže u različitim vremenima (fazama razvoja). Krutchen (1995) je postavio svoj 4+1 pogled modela arhitekture softvera, sa četiri fundamentalna arhitektonska pogleda, uz pomoć slučajeva korišćenja i scenarija:

1. Logičan pogled – pokazuje ključne apstrakcije sistema kao objekte ili klase objekata. Zgodan je za povezivanje sistemskih zahteva sa ovim entitetima sistema.

2. Procesni pogled – pokazuje kako u fazi rada, sistem radi korišćenjem povezanih procesa. Koristan je za ocenu nefunkcionalnih karakteristika, kao što su performanse i raspoloživost.

3. Razvojni pogled – pokazuje dekompoziciju softvera radi njegovog razvoja, tj. prikazuje komponente koje su implementirane od strane razvojnog tima. Ovaj pogled je značajan za softverske menadžere i programere.

4. Fizički pogled – prikazuje hardvesrske i softverske komponente i njihovu distribuciju po procesorima sistema. Ovaj pogled je zgodan za planiranje razvoja i instaliranja sistema.

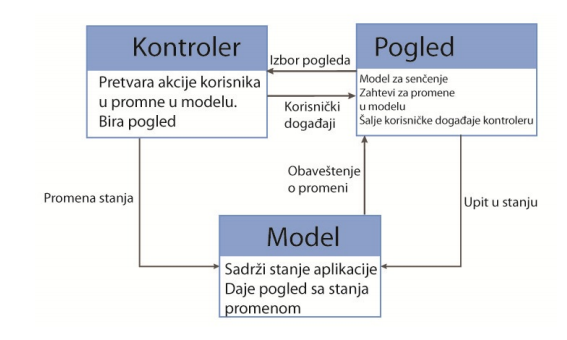
Pored ovih pogleda, koristi se i tzv. Koncepcijski pogled, kao jedan apstraktan pogled na sistem koji može biti osnova za dekompoziciju uopštenih zahteva na detaljnije specifikacije zahteva. To pomaže inženjerima da odluče oko komponenti i njihovoj višestrukoj upotrebi, i da predstave jednu liniju proizvoda umesto jednog sistema.

U praksi, koncepcijski pogledi se uvek razvijaju u toku procesa razvoja da bi se olakšalo donošenje odluka u vezi arhitekture. Oni olakšavaju komunikaciju sa akterima sistema. Ostali pogledi se razvijaju prema potrebi.

* 1. Šta je arhitektonski šablon? Objasni MVC šablon (model-kontroler-pogled). Prikaži arhitekturu veb aplikacije urađenu prema MVC šablonu.

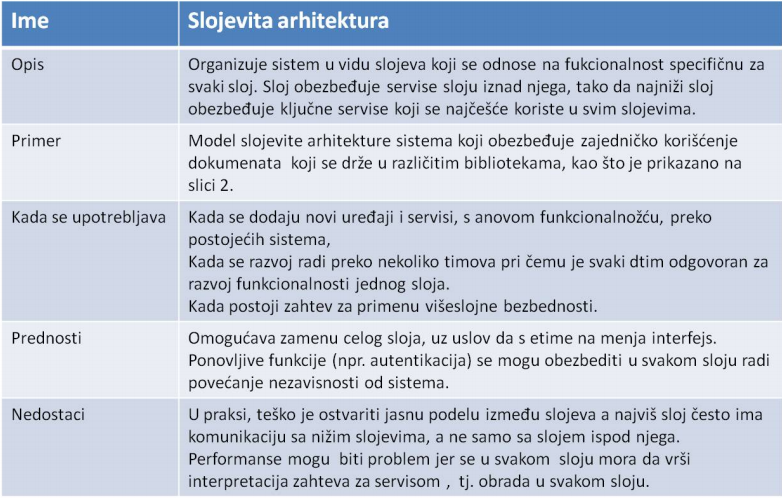
Arhitektonski šabloni opisuju organizaciju sistema koja se pokazala uspešnom u ranijim sistemima

Radi ilustracije, na slici 1 je prikazan dobro poznati šablon Model-Pogled-Kontroler (engl. Model-View-Controller), ili MVC šablon koja se koristi kod mnogih veb sistema.



* 1. Objasnite zašto upotrebljavate nekoliko šablona arhitekture kada projektujete arhitekturi velikog sistema. Pored onoga što je rečenu u tekstu predavanja o šablonima, koje su dodatne informacije koje mislite da su od koristi pri projektovanju velikih sistema?
  2. Šta je slojevita arhitektura? Dajte opis šablona slojevite arhitekture. Prikažite grafički primer slojevite arhitekturu softverskog sistema. Šta softverski slojevi omogućavaju? Dajte primer slojevite arhitekture nekog sistema.

Slojevita arhitektura obezbeđuje odvajanje i nezavisnost komponenata softvera primenom funkcionalnih slojeva (Tabela 1 ). Svaki sloj zavisi od uređaja i servisa koje nudi sloj ispod njega.



* 1. Šta je arhitektura sa skladištem podataka? Ko koristi ovu arhitekturu? Dajte opis šablona arhitekture sa zajedničkim skladištem podataka. Dajte grafički prikaz jedne arhitekture sa zajedničkim skladištem podataka. Koja svojstva ima ova arhitektura? Šta su pasivna, a šta aktivna skladišta podartaka?

Repozitorijum obezbeđuje pristup svih podsistema jedinstveoj strukturi podataka.

Arhitektura softvera uključuje dekompoziciju sistema, globalni tok kontrole, rad sa graničnim uslovima, i protokole komunikacije između podsistema.

Primenom različitih arhitektonskih stilova, oklašan je i ubrzan rad projektanata, jer pri iyboru određenog arhitektonskog stila, oni preuzimaju sva svojstva izabrane arhitekture, te nije potrebno da to samo definišu

U slučaju arhitektonskog stila Repozitorijum, podsistemi pristupaju jednom jedinstvenoj j strukturi podataka, koji se naziva centralni repozitorijum (skladište podataka).

* 1. Šta je model cevi i filtra? Šta je „cev“, a šta je „filter“? Opišite šablon arhitekture sa cevima i filtrima. Dajte prime i grafički prikažite primer jedne arhitekture sa cevima i filtrima. Za koje slučajeve je ova arhitektura pogodna, a za koje nije?

Prima veći broj ulaza i šalje rezultate podsistemima u vidu skupa izlaza Sistem sa arhitekturom cevi i fitra prima veći broj ulaza i šalje rezultate podsistemima u vidu skupa izlaza. Podsistemi se nazivaiju "filtrima" a veze (asocijacije) između podstistema "cevima". Svaki filter zna samo format podataka i sadržaj samo na svojim ulaznim cevima, i ne zna filtre (podsisteme) koji su ih proizveli. Filtri rade istovremeno a sinhronizacija se obavlja preko cevi. Najpoznatija primena ove arhitekture je kod Unix shell

Arhitektura cevi i filtra je pogodna ya sisteme koje vrše transformaciju tokova podataka bez intervencije korisnika. Nije pogodna za sisteme koji imaju složenije interakcije među podsistemima, kao što si informacioni sistemi upravljanja (IMS) ili interaktivni sistemi.

Model cevi i filtra je model izvršne organizacije sistema procesa funkcionalnih transformacija koji, koristeći svoje ulaze, proizvodi izlaze.

* 1. Šta su aplikacioni sistemi-aplikacije? Šta su arhitekture aplikacija, šta sadrže? Kako odlučujete da li kupiti ili napravite softver? Kako koristiti modele arhitekture standardnih softverskih proizvoda?

Aplikacioni sistemi su softverski sistemi koji se koriste radi zadovoljenja poslovnih ili organizacionih potreba neke firme. Firme u istom poslovnom sektoru imaju slične potrebe, te i aplikacioni sistemi u tim sektorima imaju sličnu funkcionalnost, tj. skup funkcija koje podržavaju. Ovi sistemi imaju softverske arhitekture koje odražavaju strukturu i organizaciju premereno organizacijama u koje obavljaju slične funkcije.

Postoje više načina korišćenja modela arhitekture aplikacije. Ako ste projektant softvera možete koristiti model arhitektura aplikacije na različite načine:

1. Kao početnu tačku procesa projektovanja arhitekture: Ako nedovoljno poznajete vrstu aplikacije koju treba da razvijete, možete uzeti, kao početni uzor, arhitekturu opšte aplikacije za tu vrstu poslovanja, i onda prilagođavate tu arhitekturu potrebama aplikacije koju razvijate.

2. Kao referencu za upoređenje: Pošto ste razvili arhitekturu sistema koji projektujete, želite da uporedite vaše rešenje sa arhitekturom opštih aplikacija u tom domenu.

3. Kao način da organizujete rad projektnog tima: Imajući u vidu standardnu arhitekturu aplikacija u određenom domenu, možete raspodeliti zadatke razvoja arhitektura vašeg sistema po uobičajenim komponentama koje će i vaša arhitektura imati.

4. Kao sredstvo za ocenu komponenti radi višestruke upotrebljivosti: Pri projektovanju komponente sistema, želite da je napravite tako da se može koristiti i u drugim sistemima, tj. da se višestruko koristi. Zato je korisno da se ona upoređuje sa komponentama opštih sistema da bi ste videli da li ona ima svojstva koja su slična svojstvima komponentama opštih sistema. Ako ima, to je dobar indikator da se i vaša komponenta može koristiti i u drugim aplikacijama.

5. Kao rečnik termina koji se koristi kod razgovora o aplikacijama određenog tipa: Kod diskusija o specifičnim aplikacijama ili kod upoređivanja aplikacija istog tipa, možete koristiti koncepte opštih arhitektura kada govorite o aplikacijama.

* 1. Šta su sistemi za transakcionu obradu? Koja je struktura sistema za transakcionu obradu? Prikažite arhitekturu softvera ATM sistema (bankomata).

Sistemi za transakcionu obradu (engl. Transaction processing– TP) se projektuju da obezbede obradu zahteva korisnika za informacijama iz baze podataka, ili zahteve za unos novih podataka u bazu podataka. Transakcija u sistemu baza podataka je niz operacija koji se tretira kao jedna jedinica (atomska jedinica) koja se mora izvršiti u celini pre nego što se stanje u bazi podataka trajno promeni (tj. upisani podaci postaju memorisani). To obezbeđuje integritet baze podataka, jer i u slučaju nekog poremećaja u toku procesa upisivanja novih podataka ili čitanja postojećih, ne može doći do nedefinisanog stanja baze. Transakcija (definisan niz operacija) mora se o celosti izvršiti, ili se poništavaju izvršene operacije niza i sistem se postavlja na početak, tj. u stanje u kome je bio pre početka izvršenje prve operacije u transakciji. Sistemi za transakcionu obradu su najčešće u formi interaktivnih sistema u kojima korisnici postavljaju asinhrone zahteve za određene servise sistema.

* 1. Šta su informacioni sistemi? Prikažite opšti model informacionih sistema. Dajte primer nekog informacionog sistema. Kako se upravlja podacima kod informacionih sistema?

Svi sistemi koji koriste interakciju sa zajedničkom bazom podataka su tzv. transakcioni informacioni sistemi. Informacioni sistem dozvoljava kontrolisan pristup velikoj bazi informacija, kao što je katalog bibliotečkih jedinica, red vožnje letova, ili elektronski zdravstveni kartoni pacijenata u nekoj bolnici. Najčešće, informacioni sistemi su i veb sistemi, jer obezbeđuju pristup preko veb pretraživača

* 1. Šta su sistemi za obradu jezika? Kako izgleda arhitektura sistema za obradu jezika? Kakva je arhitektura kompajlera, tj. prevodioca programskih jezika? Dajte prikaz arhitekture sistema za prevođenje programskog jezika koja sadrži skladište

Sistemi za obradu jezika prevode neki prirodan ili veštački jezik u neku drugi način predstavljanja tog jezika. U slučaju programskih jezika, dobija se mašinski kod koji omogućava računarsko izvršenje programskih instrukcija. Drugi jezici u obliku XML forme podataka mogu se prevesti u komande za rad sa bazom podataka ili u neku drugu XML formu predstavljanja podataka. U slučaju prirodnih jezika, prevođenje može da dovede do prevoda u drugom prirodnom jeziku (npr. , sa engleskog u nemački).

* 1. Upotrebom osnovnog modela višeslojne arhitektura nekog informacionog sistema, predloži komponente koje bi mogle da bude deo informacionog sistema koji dozvoljava korisnicima da vide informaciju o dolazećim i dolazećim letovima na jednom aerodromu.

Bezbednost: Ako je bezbednost bitna, višeslojna arhitektura je dobro rešenje, da bi se najkritičnije vrednosti stavile pod kontrolu u unutrašnjim slojevima sistema, sa visokom dozom zaštite

* 1. Definišite arhitekturu sistema definisan sledećim opis om: *Sistem upravlja procesom realizacije službenih putovanja zaposlenih u firmi ABC. Sistem treba da obezbedi odobrenje za službeni put, izvore finansiranja puta, rezervacije karata, hotela i dr., izveštaj lica koje je išlo na službeni put, isplatu dnevnica, plaćanje računa za karte, hotel i dr., izradu konačnog finansijskog obračuna, i arhiviranje celokupne dokumentacije vezane za službeni put.* (povezan zadatak: 5.8)

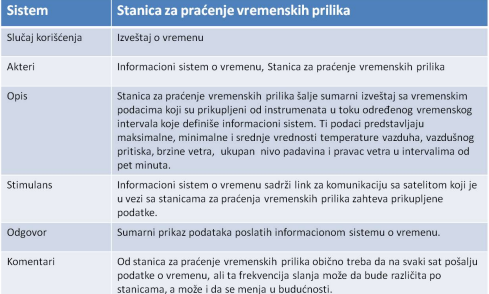
1. **Lekcija 8: PROJEKTOVANJE SOFTVERA I ŠABLONI KREIRANJA OBJEKATA**
   1. Šta su objektno-orijentisani sistemi? Šta je stanje objekta? Koje su osnovne aktivnosti procesa projektovanja? Zašto je važno razumevanje spoljnjeg okruženja u kome treba da radi softverski sistem? Šta je model konteksta sistema? Dajte neki primer modela konteksta sistema.

Jedan objektno-orijentisano softverski sistem čine interaktivni objekti koje održavaju svoje unutrašnje stanje i obezbeđuju operacije na to stanje. To znači da te operacije, kada se izvrše, menjaju stanje jednog ili više objekata, zavisno koliko objekata su u interakciji kada se jedna operacija izvršava.

Stanje objekta odražava trenutno vrednosti njegovih atributa. Stanje nekog objekta je „privatno“, što znači da ne može da se menja direktnom intervencijom nekog spolja. Ako neki objekt želi da promeni stanje nekog drugog objekta, on mora da tom drugom porukom pošalje odgovarajuću poruku koja bi trebalo da aktivira odgovarajuću operaciju, tj. metod koju objekat koji prima poruku poseduje, a samo ona onda može da izvrši promenu stanja (npr. da promeni vrednost nekog atributa objekta). Slanje jedne poruke između dva objekta, koja pobuđuje odgovarajuću operaciju (metod), ili ceo lanac povezanih operacija, nazivamo interakcijom između dva objekta. Na slici 1 dat je primer plaćanja računa za struju preko bankomata. Pojednostavljen objektni model prikazuje dva objekta u interakciji i jednog aktora (čoveka koji daje nalog za isplatu računa).

* 1. Šta je model interakcije? Šta je UML dijagram korišćenja (UML use case). Šta je akter? Nacrtajte dijagram korišćenja nekog sistema i objasnite svaki od slučajeva korišćenja koji ste koristili u dijagramu, u vidu formalnog tekstualnog opisa. Šta su scenariji? Navedite jedan scenario za jedan slučaj korišćenja sistema iz vašeg dijagrama.

Model interakcija daje dodatne informacije, jer ukazuje kako sistem koji projektujemo komunicira sa okruženjem, tj. sa drugim sistemima u svom okruženju (a na koje ukazuje model konteksta). Jeadn od UML model interakcija je UML dijagrama korišćenja (engl., use case), koji opisuju po jednu interakciju sa sistemom koji projektujemo. Simbol aktera označava ili neki drugi sistem ili čoveka koji koristi naš sistem. Na slici 2 je prikazan dijagram korišćenja koji pokazuju slučajeve korišćenja stanica za praćenje vremenskih prilika, tj. sistema koji se projektuje, Svaka elipsa simboličko označava po jedan slučaj korišćenja (koji se bliže opisuje tekstualnom informacijom u vidu scenarija).



* 1. Na početku projektovanja softverskog sistema, polazite od prethodno definisane arhitekture sistema. Zašto? Koje informacije vam ona daje,a relevantne su za vaš dalji rad na projektovanju sistema. Kako vama dijagrami slučajeva korišćenja i prateće tekstualne informacije, npr. scenariji, pomažu da utvrdite osnovne objekte i operacije koje sistem treba da ima? Dajte primer.

1. **Lekcija 9: ŠABLONI PROJEKTOVANJA STRUKTURE SOFTVERA**
   1. Kako se vrši detaljno projektovanje klasa? Šta je UML model klase? Kako određujete njene metode i atribute? Dajte neki vaš primer. Dajte projektno rešenje klase za sledeće objekte (definišite atribute i operacije/metode): telefon, štampač personalnog računara i bankovni račun.
   2. Šta su modeli projektovanja? Šta je nivo apstrakcije modela projektovanja? Navedite strukturne UML modele. Navedite UML dinamičke modele. Definišite sve metode i atribute klasa koje koristite.
   3. Definišite argumente u zaglavlju svakog metoda. Sve dijagrame, klase i metode uradite za sledeći slučaj korišćenja sistema: *Sistem treba da uključi sistem za grejanje vaše kuće svakog jutra u 6 sati, a da vam uključi radio pored vašeg kreveta u 7 sati.*
2. **Lekcija 10: ŠABONI PROJEKTOVANJA PONAŠANJA SISTEMA**
   1. Šta su UML sekvencijalni dijagrami? Koja je veza između UML dijagrama korišćenja i sekvencijalnih dijagrama? Dajte primer dva povezana sekvencijalna dijagrama. Prvi se odnosi na visok nivo apstrakcije sistema, a drugi – na niski, detaljni nivo. Čemu služi sekvencijalni model?
   2. Šta je UML model stanja? Šta je stanje objekta? Kako model menja stanje? Zašto se koriste modeli stanja? Dajte neki primer model stanja i objasnite navedena stanja. Dajte dijagram stanja lifta.
   3. Šta je interfejs? Koja je uloga interfejsa? Koja je veza interfejsa i klasa? Da li jedna klasa može imati više interfejsa? Da li jedan interfejs može da bude povezan sa više klasa? Šta su zaglavlja(signature operacija/metoda? Šta one sadrže? Šta je povratna vrednost metoda? Šta je semantika interfejsa? Šta su servisi, šta oni rade?
   4. Šta je šablon? Šta je šablon projektovanja? Koje informacije sadrži opis jednog šablona za projektovanje. Opišite šablon „posmatrač“ (engl. Observer). Kada se koristi ovaj šablon? Dajte primer primene ovog šablona.
   5. Koja je korist od primene šablona projektovanja? Koji problemi mogu nastati u primeni šablona? Kako se ovi problemi rešavaju? Kako se nalazite pogodan šablon?
3. **Lekcija 11: IMPLEMENTACIJA SOFTVERA**
   1. Šta implementacija sistema? Da li je to programiranje, ili rad sa drugim aspektima razvoja softvera? Koji? Objasnite svaki od ovih aspekata.

Implementacija obuhvata razvoj programa primenom programskih jezika visokog ili niskog nivoa, ili pak, korišćenje opštih, gotovih sistema koji se onda prilagođavaju zahtevima korisnika.

Softversko inženjerstvo obuhvata sve aktivnosti procesa razvoja softvera, počev od specifikacije početnih zahteva sistema, pa do održavanja i upravljanja razvijenim i primenjenim sistemima. Kritična faza ovog procesa je implementacija sistema, kada se kreira izvršna verzija softvera. Implementacija obuhvata razvoj programa primenom programskih jezika visokog ili niskog nivoa, ili pak, korišćenje opštih, gotovih sistema koji se onda prilagođavaju i uklapaju u sistem koji zadovoljava specifične zahteve organizacije.

Ovde se nećemo baviti dobrom programerskom praksom i primenom konkretnih programskih jezika, jer se to izučava u posebnim predmetima. Umesto toga, ovde ćemo razmatrati posebno interesantne i važne aspekte implementacije softvera, tj. razvoja izvršnog softvera, a koje se obično ne izučavaju u predmetima programiranja. To su sledeći aspekti:

1. Ponovno korišćenje: Savremeni softverski sistemi koriste ponovno upotrebljive komponente ili sisteme. Pri razvoju svog softvera, trebalo bi u što većoj meri da koristite već razvijen i provere kod, a to obezbeđuje primena ranije razvijenih i primenjenih softverskih komponenti.

2. Upravljanje konfiguracijom: Prilikom procesa razvoja, kreiraju se mnoge verzije svake od softverskih komponenata. Ako ne vodite računa o njima, primenom posebnog sistema za upravljanje konfiguracijom softvera, može se desiti da koristite pogrešne verzije komponenti u vašem sistemu. Konfiguraciju softvera čini skup softverskih komponenta sa ispravnim verzijama.

3. Razvoj na razvojnom serveru a za rad na ciljnom serveru: Obično se softver razvija na serveru koji ne mora da bude istog tipa kao i server na kome će softver raditi. To otvara puno izazova u razvoju o kojima se mora voditi računa.

* 1. Ponovna upotreba softvera: Šta je to? Koja je korist od ovoga? Koji su nivoi primene ponovljivosti softvera? Objasniti primenu na svakom nivou. Koji su troškovi primene upotrebljenog softvera, odn. ranije razvijenog softvera/softverskog proizvoda?

Razvoj softvera koji se višestruko koristi se vrši na više nivoa: apstraktni nivo (projektni šabloni), nivo objekta (objekti iz biblioteka), nivo komponenti i nivo sistema

Do 90-tih godina, softver je najčešće razvijan „od početka“, za svakog naručioca posebno, korišćenjem programskih jezika višeg nivoa za pisanje koda (programa). Ponovno korišćenje softvera je bilo prisutno samo u slučaju korišćenja programa za pojedine uobičajene funkcije iz programske biblioteke. Međutim, pritisak da se brže i što jeftinije razvije softver, pogotovu poslovnih sistema, doveo je do korišćenja ranije razvijenog softvera, tj. do razvoja softverskih proizvoda koji se višestruko koriste. Ponovna upotreba softvera je moguća na više različitih nivoa:

1. Apstraktni nivo: Na ovom nivou nema direktne ponovne upotrebe softvera, već se koristi znanje primenjeno prilikom projektovanja softvera. Koriste se projektni šabloni i strukturni šabloni kao apstraktna znanja za ponovno korišćenje.

2. Nivo objekta: Na ovom nivou ponovno se koriste objekti iz biblioteke. Pri razvoju softvera, treba naći pogodnu biblioteku objekata i koristiti objekte čije metode obezbeđuju potrebnu funkcionalnost (na primer, JavaMail biblioteka).

3. Nivo komponente: Komponente su kolekcije objekata i klasa objekata koje zajedno rade na obezbeđivanju odgovarajućih funkcija i servisa. Pri ponovnoj upotrebi neke komponente, integracijom nekoliko ranije razvijenih softverskih sistema najčešće je potrebna određena promena softvera ili njegova dopuna, da bi se komponenta prilagodila potrebama novog korisnika, tj. softvera u koji se ugrađuje (na primer, grafički korisnički interfejs - GUI).

4. Nivo sistema: Na ovom nivou se upotrebljava ponovo ceo softverski sistem. U tom slučaju se njegova konfiguracija (komponente i dr.) prilagođava specifičnim potrebama korisnika (npr. SAP). Na ovaj način se može novi sistem kreirati konfigurisanjem novog sistema

* 1. Upravljanje konfiguracijom softvera: Šta je konfiguracija softvera? Koja je korist od upravljanja konfiguracijom? Koji su izazovi povezani sa konfiguracijama softvera? Koje su tri osnovne aktivnosti upravljanja konfiguracijom? Opišite upravljanje konfiguracijom za svaku od ovih aktivnosti. Navedite neki softverski alat za upravljanje konfiguracijom softvera?

Konfiguracija softvera je skup određenih verzija softverskih komponenata. Upravljanje konfiguraciom je opšti proces upravljanja promenama softverskog sistema

Proces razvoja softvera je vrlo dinamičan, jer dolazi do čestih promena planova, zahteva i dr. Zbog toga je vrlo važno uspostaviti dogovarajući sistem upravljanja ovim promenama. Kako sinhronizovati rad članova razvojnog tima tako da se ne mešaju jedan drugom u posao (bez kontrole)? Ako jedan od njih unosi neku promenu, da li se ta promena tiče i rada ostalih? Da li neće ta promena „uništiti“ rad nekog drugog člana tima? Kako obezbediti da svi članovi tima u svakom momentu rade sa najnovijim verzijama softverskih komponenata? Ako mi sa novom verzijom komponente nešto loše ide, kako da se vratim na prethodnu verziju? Sve su ovo pitanja na koje očekujemo odgovor primenom odgovarajućeg sistema za upravljanje konfiguracijom softvera.

Konfiguracija softvera je skup određenih verzija softverskih komponenata. Moguće je da se za različite korisnike koriste različite verzije softverskih komponenata. Te verzije ne moraju da budu samo vremenski uslovljene, već mogu biti rezultat obezbeđivanja nešto različitih funkcija, ili primene različitih tehnologija.

Upravljanje konfiguracijom (engl., Configuration Management) je opšti proces upravljanja promenama softverskog sistema. Cilj primene sistema za upravljanje konfiguracijom je da se podrži proces integracije sistema tako da svi članovi razvojnog tima mogu da na kontrolisan način pristupaju kodu softvera i njegovoj dokumentaciji, da pronađu promene do kojih je došlo, i da izvrše prevođenja u izvršni oblik (tj. da izvrše kompilacije) i da povežu sve komponente koje čine sistem.

Upravljanje konfiguraciom je upravljanje verzijama, integracija sistema i praćenje problema radi prijava grešaka i problema u softveru. Postoj tri osnovne aktivnosti upravljanja konfiguracijom:

1. Upravljanje verzijama, kojim se obezbeđuje praćenje različitih verzija softverskih komponenata, kao i koordinacija rad više programera. Na taj način se sprečava da jedan programer promeni kod koji je drugi programer prethodno pripremio i ubacio u sistem.

2. Integracija sistema, kojom se obezbeđuje pomoć softverskim inženjerima da definišu verzije komponenata koje čine svaku verziju sistema. To omogućava automatsku izradu izvršnog koda sistema (tj. njegovu kompilaciju i povezivanje) sastavljanog od različitih komponenata.

3. Praćenje problema, koji omogućava korisnicima korisnicima da prijave greške (bagove) i druge probleme u radu sistema, a razvojnom timu omogućava da koordiniše rad na otklanjanju prijavljenih grešaka i rešavanju problema.

Svaki alat koji se bavi upravljanju konfiguracijama softvera (npr. ClearCase) mora da podrži sve tri navedene aktivnosti. Ovi sistemi integrišu sve navedene aktivnosti obezbeđujući korišćenje istog korisničkog interfejsa i iste baze podataka.

Pored integrisanih sistema za upravljanje konfiguracijama, koriste se i posebni alati, koji obezbeđuju samo jednu od navedenih aktivnosti. Na primer, alat Subvesrion vrši upravljanje verzijama komponenata i sistema. Integraciju sistema može da obavlja alat koji samo to radi, kao na primer, Unix make. Sistemi za praćenje grešaka, kao što je Bugzilla, se koriste za prijavljivanje grešaka i za druge stvari, i vode evidenciju o tome koja je greška otklonjena, a koja još nije.

* 1. Navedite specifičnosti razvoja sledećih softverskih sistema: a) ugrađeni sistemi, b) posrednički softver (middleware).
  2. Koji se softverski alati koriste u okviru razvojnih platformi? Čemu služi server za razvoj softvera/razvojni server)? Šta su to integrisane razvojne platforme (IDE)? O čemu se mora voditi računa pri primeni razvojnih platformi?
  3. Razvoj softvera sa otvorenim izvornim kodom: Šta je to? Šta je karakteristično za ovakav način razvoja softvera? Ko nosi odgovornost za razvoj ovog softvera? Šta treba da imate u vidu kada odlučujete da li da koristite softver otvorenog koda ili komercijalni softver? Koji je poslovni model primene softvera sa otvorenim izvornim kodom?

Razvoj softvera sa otvorenim kodom (engl., open source development) je pristup razvoju softvera u kome je izvorni kod softvera javno dostupan a pozivaju se dobrovoljci da učestvuju u njegovom daljem razvoju. Organizacija Free Softvare Foundation (www.fsf.org) promoviše ovaj pristup, jer smatra da bi izvorni kod trebalo da bude dostupan korisnicima softvera radi njegove kontrole i eventualno prilagođavanja softvera njihovim specifičnim potrebama. Pretpostavka je da će kod softvera biti kontrolisan i razvijan od tima dobrovoljaca, a ne i od krajnjih korisnika softvera (sem u izuzetnim slučajevima).

Softver otvorenog koda proširuje ovu ideju i primenom Interneta, uključuje mnogo veću populaciju dobrovoljaca u razvoj softvera. Po pravilu, svaki od njih može da izvesti o primećenim greškama u kodu i da ih otkloni, kao i da proizvede nova svojstva i novu funkcionalnost softvera. Međutim, u praksi, uspešan razvoj softvera sa otvorenim izvornim kodom još uvek zavisi od osnovne grupe koja razvija softver i koja kontroliše promene na softveru.

Najpoznatiji softver otvorenog koda je Linux operativni sistem. Drugi sistemi su: Java, Apache veb server i mySQL sistem baza podataka. Mnogi veliki proizvođači softvera, kao što su IBM, Oracle, HP, podržavaju razvoj otvorenog koda i često baziraju svoje proizvode na softverima otvorenog koda. Postoje na hiljade softvera otvorenog koda koji se nudi preko Interneta.

* 1. Licence za korišćenje softvera otvorenog koda: Ko je vlasnik softvera? Ko ima prava da ga menja i koristi? Opište prava i obaveze koje predviđaju licence namenjene sistemima sa otvorenim kodom: a) GNU opšta javna licenca b) GNU ograničena javna licenca c) Berkli licenca standarde distribucije. Koji mogući problema prate korišćenje softvera sa otvorenim kodom? Koje preporuke bi dali vašem direktoru koji razmišlja da razvoj novog softverskog proizvoda zasnuje na korišćenju komponenti koji su otvorenog koda.

Preuzimanje softvera otvorenog koda je najčešće potpuno besplatno. Međutim, ako želite da imate dokumentaciju i podršku tima koji razvija softver, onda najčešće treba da platite tu uslugu. Softver otvorenog kod je često vrlo pouzdan zbog velike populacije korisnika ili učesnika u razvoju, koji sami otklanjaju uočene greške i nedostatke.

Vlasnik softvera može da postavi ograničenja za njegovo korišćenje, uključujući i korišćenje posebne dozvole za korišćenje softvera sa otvorenim kodm.

Iako je u principu korišćenje softvera sa otvorenim izvornim kodom besplatno, to ne znači da njegov korisnik može da radi šta hoće sa njim. S pravne tačke gledišta, onaj koji je razvio kod (kompanija ili pojedinac) je vlasnik softvera, ili dela koji je razvio. On može da postavi ograničenja za njegovo korišćenje, uključujući i korišćenje posebne dozvole za korišćenje softvera sa otvorenim kodm. Neki od njih smatraju da softverski sistem koji se razvije korišćenjem softvera otvorenog koda, takođe treba da bude softver otvorenog koda.

Drugi pak prihvataju da sistem koji je razvijem uz podršku softvera sa otvorenim kodom, može da se koristi bez ikakvih ograničenja.

Licence (dozvole) za korišćenje softvera sa otvorenim kodom se mogu podeliti u tri kategorije:

1. GNU opšta javna licenca (GNU General Public License – GPL): Ona u principu zahteva da softver koji je razvijen sa softverom koji ima licencu tipa GPL, takođe mora da se koristi sa GPL licencom. To znači da i tako razvijen softver mora da bude softver otvorenog koda.

2. GNU ograničena javna licenca (GNU Lesser General Public License – LGPL): Ova je varijanta GPL licence koja dozvoljava da vi napravite softversku komponentu korišćenjem softvera otvorenog koda, bez obaveze da vaša komponenta bude softver otvorenog koda.

3. Berkli licenca standardne distribucije (Berkley Standard Distribution License – BSD): Ovo nije recipročna licenca (kao GPL) jer vas ne obavezuje da vaš softver koji ste razvili korišćenje softvera otvorenog koda bude takođe softver otvorenog koda. To znači da ne morate da ga javno nudite i da kao svoj kod, možete da ga prodajete. Jedino što mora da uradite, to ja da jasno navedete koje komponente koje koristite su otvorenog koda i ko je njihov autor.

* 1. Definišite dijagrame klasa i sekvencijalne dijagrama za softverski sistem dat sledećim opisom: *Sistem upravlja procesom realizacije službenih putovanja zaposlenih u firmi ABC. Sistem treba da obezbedi odobrenje za službeni put, izvore finansiranja puta, rezervacije karata, hotela i dr., izveštaj lica koje je išlo na službeni put, isplatu dnevnica, plaćanje računa za karte, hotel i dr., izradu konačnog finansijskog obračuna, i arhiviranje celokupne dokumentacije vezane za službeni put.* (povezan zadatak: 5.8 i 6.16)

1. **Lekcija 12: TESTIRANJE SOFTVERA**
   1. Šta je testiranje softvera? Koji su ciljevi testiranja? Nacrtajte model testiranja. Šta pokazuje testiranje softvera? Koja je razlika između verifikacije i validacije softvera? Zašto služe verifikacija i validacija?

Testiranje služi da se utvrdi da li program radi ono bi trebalo da radi i da se otkriju greške u programu pre nego što se pusti u upotrebu. Testiranje se vrši puštanjem da softver radi, pri čemu se koriste „veštački“ podaci. Onda se proveravaju rezultati testiranja da bi se videlo da li oni ukazuju na neku grešku, nepravilnost i da bi se došlo do informacija o nefunkcionalnim svojstvima softvera.

Proces testiranja ima dva cilja:

1. Da pokaže inženjeru razvoja i kupcu da softver ostvaruje svoje zahteve. Za softver koji se radi po posebnom zahtevu, sprovodi se po najmanje jedan test za svaki od zahteva iz dokumenta sa zahtevima. U slučaju opštih softverskih proizvoda, testovi treba da provere sva svojstva softvera, kao i kombinacije ti svojstava, onako kako su realizovana u proizvodu koji se pušta u prodaju.

2. Da otkrije situacije u kojima ponašanje softvera nije ispravno, nepoželjno ili da ne odgovara specifikaciji softvera. To su posledice defekata u softveru (pad sistema, neželjene interakcije sistema sa drugim sistemima, netačni proračuni, i nekontrolisana promena podataka), čiji razlozi treba da se testiranjem pronađu.

Prvi cilj se ostvaruje sprovođenje testova za proveru ispravnosti softvera, tj. testovima validacije (engl., validation testing). Drugi cilj se ostvaruje testiranjem na defekte (engl., defect testing). U praksi, ove dve vrste testiranja se mešaju, jer i pri sprovođenju testova za proveru ispravnosti, otkrivaju se defekti (greške) u softveru, a i pri testiranju na defekte, vidi se da li program zadovoljava zahteve.

Krajnji cilj procesa verifikacije i validacije je utvrđivanje da li softver zadovoljava svoju svrhu, tj. da je sistem dobar za planiranu upotrebu.

Proces verfikacije i validacije služi za proveru da li softver u razvoju zadovoljava svoju specifikaciju i da li obezbeđuje funkcionalnost koju očekuju njegovi kupci. Ovaj proces počinje odmah po utvrđivanju zahteva, i sprovodi se u svim fazama procesa razvoja softvera.

Cilj verifikacije je provera da li softver zadovoljava postavljene funkcionalne i nefunkcionalne zahteve. Validacija je opštiji proces. Cilj validacije je da potvrdi da softver zadovoljava očekivanja kupca. Ako specifikacija zahteva nije dobro napravljena, može se desiti da verifikacija bude pozitivna (softver zadovoljava sve zahteve) a da validacija bude negativna (softver ne zadovoljava očekivanja kupca). Zato je validacija i najbitnija.

Krajnji cilj procesa verifikacije i validacije je utvrđivanje da li softver zadovoljava svoju svrhu, tj. da je sistem dobar za planiranu upotrebu. Nivo poverenja koji se zahteva od softvera, zavisi od svrhe softvera, od očekivnja korisnika sistema, i od trenutnog tržišnog okruženja sistema:

1. Svrha softvera: Što je softver važniji (kritičan), to je važnije da bude pouzdan. Na primer, sistemi kod kojih je bezbednost kritična, imaju veći nivo zahtevanog poverenja nego softver razvijen kao protoptip koji treba da pokaže ideje o novom proizvodu.

2. Očekivanja korisnika: Korisnici najčešće tolerišu greške u novom softveru, ali kasnije, očekuju da on postane pouzdan, tj. da se bitno smanji broj grešaka.

3. Tržišno okruženje: Vrši se upoređenje sa konkurentskim proizvodima, utvrđuje se koliko je kupac spreman da plati za proizvod, kao i koji su zahtevani rokovi za isporuku softvera. U želji da se novi proizvod što pre pojavi na tržištu, kompanije često puste proizvod koji još nije dovoljno testiran. Ako je proizvod jeftin, onda i kupci nisu vrlo zahtevni, te tolerišu i greške.

* 1. Koji faktori utiču na nivo poverenja korisnika u softver? Šta je inspekcija i recenzija softvera? Koje su prednosti inspekcije u odnosu na testiranje softvera?

Inspekcije softvera podržava proces verifikacije i validacije u različitim fazama procesa razvoja softvera. Proces verifikacije i validacije sadrži i inspekcije (preglede) i recenzije softvera. Ove aktivnosti analiziraju i proveravaju sistemske zahteve, projektne modele, izvorni kod programa, i predložene testove sistema. To su tzv. „statičke“ tehnike verifikacije i validacije, jer ne zahtevaju da softver radi, da bi bio proveren.

Inspekcija bolje otkriva greške nego testiranje programa. Inspekcije se najčešće usmeravaju na izvorni kod sistema, ali i na ostala dokumenta, kao što je specifikacija zahteva i projektni model. Pri inspekciji, koristi se znanje o sistemu, njegovom domenu primene, a i znanje o programiranju ili modeliranju, da bi se otkrile greške. Inspekcija pruža tri prednosti u odnosu na testiranje:

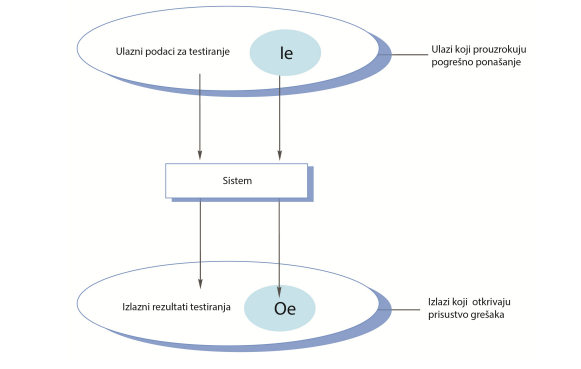
1. Za vreme testiranja, greške mogu da prikriju druge greške. Kada se dobije pogrešan rezultata, nikad se ne zna pouzdano da li je to posledica jedne ili neke druge greške. Kako je inspekcija statički proces, on ne zavisi od interakcije među grešaka. Jedna inspekcija, zbog toga, može da otkrije mnoge greške u sistemu.

2. Nekompletne verzije softverskog sistema se mogu proveravati bez dodatnih troškova. Iako je program nekompletan, vi koristite prednost specijalizovanih testova za testiranje samo delova softvera koji je razvijen. Naravno, na ovaj način se uvećavaju troškovi razvoja softvera.

3. Pored traženja defekata, inspekcija može uzeti u obzir i šire atribute kvaliteta programa, kao što je usaglašenost sa standardima, prenošljivost (na različite sisteme) kao i lakoća održavanja. Analizirate neefikasnost u radu, neodgovarajuće algoritme ili loš način programiranja koji vodi ka dobijanju sistema koji će biti težak za održavanje i menjanje.

Istraživanja su pokazala da je inspekcija bolje otkriva greške nego testiranje programa. Naravno, inspekcija ne može da zameni testiranje, ali omogućava da se veći broj grešaka otkrije i pre testiranja. Inspekcije nisu dobre u otkrivanju defekata koji su rezultata neočekivanih interakcija između različitih delova softverskog sistema, usled problema sa vremenom, ili problema sa performansama sistema. Pored toga, u malim kompanijama je teško da se formira poseban tim za inspekciju softvera, zbog nedostatka ili ljudi ili novca.

* 1. Nacrtajte model tradicionalnog procesa testiranja softvera. Objasnite svaku fazu procesa testiranja. Koja je razlika ručnog i automatskog testiranja? Objasnite zašto testiranje može samo da utvrdi prisustvo grešaka, a ne i njihovo odsustvo.



* 1. Šta je razvojno testiranje softvera? Koji nivoi (granulacije) testiranja u razvoju se koriste? Dajte kratak opis svakog nivoa testiranja softvera u fazi razvoja.

Komercijalni softveri najčešće prolaze kroz tri faze testiranja:

1. Razvojno testiranje, kada se softver testira za vreme razvoja, a da bi se otkrile greške i defekti. To najčešće rade projektanti sistema i programeri.

2. Testiranje softvera za isporuku, kada poseban tim testira kompletnu verziju softvera pre nego što se isporuči korisnicima. Cilj je da se proveri da li sistem zadovoljava zahteve svih aktera.

3. Korisničko testiranje, kada sadašnji ili budući korisnici softvera vrše testiranje u svom radnom okruženju. U slučaju softverskih proizvoda, ovo može da radi unutrašnja grupa iz marketinga koja odlučuje da li proizvod spreman i dobar da ide na tržište. Test prihvatanja (engl., acceptance testing) je tip korisničkog testiranja kada kupca formalno testira sistem da bi odlučio da li da ga prihvati ili da traži dodatni razvoj.

* 1. Testiranje jedinice: Šta je to? Šta je testiranje klase, šta obuhvata? Kako se testiraju operacije klasa u njihovoj hijerarhiji, tj. da li se ista operacija klase mora da testira i u njnoj pod- klasi ? Kako se biraju slučajevi za testiranje jedinica? Koje se strategije testiranja mogu izabrati? Opišite svaku od njih. Šta je regresiono testiranje? Objasnite kako upotreba automatskih testovi i platforme za atestiranje, kao što je Junit, uprošćavaju regresiono testiranje?

Testiranje jedinice je proces testiranja osnovnih softverskih jedinica, kao što su metodi ili klase. Metod se testira tako što se poziva u izvršenje pri čemu mu se menjaju ulazni parametri. Kada se vrši testiranje klase, onda se testovi projektuju da pokriju sva svojstva klase, te sadrže:

• Testiranje svih operacija klase;

• Ubacivanje i proveru vrednosti svih atributa klase;

• Stavljanje objekta u sva moguća stanja.To se radi tako što se vrši simulacija svih događaja koja dovode do promene stanja.

Testiranje klasa se komplikuje usled generalizacije i nasleđivanja klasa. Nije dovoljno testirati operaciju u klasi u kojoj se ona definiše, i očekivati da će u svim pod-klasama on raditi ispravno. Neophodno je testirati istu operaciju u svim pod-klasama klase u kojoj se definiše operacija.

* 1. Testiranje komponenata: Šta se testira kod softverskih komponenti? Koji tipovi grešaka se mogu javiti kod testiranja interfejsa? Koje kategorije grešaka postoje kod interfejsa? Koja su opšta pravila za testiranje komponenata?
  2. Testiranje sistema: Šta obuhvata testiranje sistema? Koja je razlika testiranja sistema i testiranje komponenti? Šta se očekuje of testiranja sistema? Šta je testiranje slučajeva korišćenja? Kako odrediti slučajeve korišćenja za testiranje? Koja svojstva sistema se testiraju? Da li se vrši i automatizovani testiranje sistema?

Testiranje sistema za vreme razvoja obuhvata sve integrisane komponente koje čine jednu verziju sistema, te se na taj način testira integrisan sistem. Testiranje sistema proverava da li su komponente kompatabilne, da li ispravno međusobno komuniciraju i da li prenose prave podatke u pravo vreme preko njihovih interfejsa. Mada ima preklapanja sa testiranjem komponenata, ipak postoje dve važne razlike:

1. Za vreme testiranja sistema, komponente koje su nezavisno razvijene , kao i gotovi sistemi, integrišu se sa novo razvijenim komponentama, te se onda testira ceo sistem.

2. Komponente koje su razvijene od pojedinih članova razvojnog tima se integrišu u ovoj fazi. Testiranje sistema je jedan kolektivni, a ne individualni proces. U nekim kompanijama testiranje sistem vrši poseban tim bez uključenja projektanata i programera koji su razvili sistem.

Kada se integrišu sve komponente sistema, očekuje se da sistem pokaže ponašanje u skladu sa specifikacijom. To se mora testom utvrditi. Mora se razviti test za proveru da sistem radi samo ono što se od njega i očekuje.

Testiranje sistema se fokusira na testiranje interakcija komponenata sistema. Ovaj test interakcija bi trebalo da otkrije komponente sa greškama koje se otkrivaju samo kada je komponenta u interakciji sa drugim komponentama sistema. Testiranje sistema takođe doprinosi uklanjanju nerazumevanja među projektantima komponenata.

* 1. Šta je razvoj vođen testovima? Dajte grafički prikaz procesa testiranja funkcije softvera u fazi razvoja. Kako se vrši automatsko testiranje inkremenata sa JUnit? Koje povoljnosti nudi razvoj softvera koji je vođen testovima? Koji su problemi ručnog testiranja? Koje su prednosti automatskog testiranja? Kada se koristi razvoj softvera vođen testovima? Šta treba da odredi testiranje sistema? Kada se postižu najbolji rezultati primene razvoja vođenog testovima?

Razvoj vođen testovima je pristup razvoju programa u kome se istovremeno vrši i razvoj koda i njegovo testiranje. Razvoj vođen testovima je pristup razvoju programa u kome se istovremeno vrši i razvoj koda i njegovo testiranje. Kod se inkrementalno razvija, i za svaki inkrement se sprovodi odgovarajući test. Dok se ne završi test jednog inkrementa, ne počinje se razvoj sledećeg. Slika 1 prikazuje aktivnosti integrisanog razvoja i testiranja softvera.

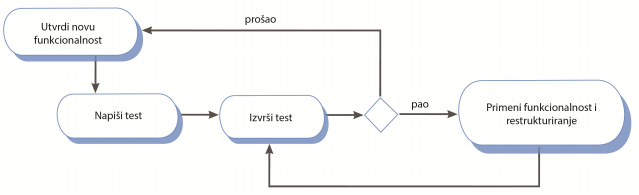
1. Najpre utvrdite inkrement softvera sa odrešenom funkcionalnošću. Inkrement treba d abude mali i primenljiv u nekoliko linija koda.

2. Napišite test za funkcionalnost inkrementa i primenite ga u vidu automatskog testa.

3. Izvršite test, zajedno sa drugim testovima. Trebalo bi da se dobiju uspešni rezultati testa za sve več razvijene inkremente, a negativne za one koji još nisu razvijeni.

4. Sada razvijete inkrement sa svojom funkcionalnalnošću pisanjem novog, odgovarajućeg koda.

5. Ako je sada test pozitivan, postupak se ponavlja sa utvrđivanjem novog inkrementa



* 1. Napišite scenario koji bi mogao da se upotrebi za projektovanje testova za stanicu za prikupljanje podataka o vremenskim prilikama.
  2. Koje su prednosti uključenja korisnika u testiranje softvera za isporuku u ranim fazama procesa testiranja? Da li ima i nedostataka u uključenju korisnika u razvoj testova?
  3. Predložite test testiranja za sistem sa sledećim opisom: *Sistem upravlja procesom realizacije službenih putovanja zaposlenih u firmi ABC. Sistem treba da obezbedi odobrenje za službeni put, izvore finansiranja puta, rezervacije karata, hotela i dr., izveštaj lica koje je išlo na službeni put, isplatu dnevnica, plaćanje računa za karte, hotel i dr., izradu konačnog finansijskog obračuna, i arhiviranje celokupne dokumentacije vezane za službeni put.* (povezan zadatak: 5.8, 6.16 i 7.18

1. **Lekcija 13: EVOLUCIJA SOFTVERA**
   1. Šta je evolucija softvera? Kako izgleda spiralni model razvoja i evolucije softvera? Ko održava softver razvijen po narudžbini?

Softversko inženjerstvo je jedan spriralni proces razvoja i evolucije, sa zahtevima, projektovanjem, implementacijom i testiranjem softvera tokom celog njegovog životnog veka Isporučen softver se mora da menja iz bar dva razloga.

1. da bi se otklanjale uočeni nedostaci i greške.

2. da bi se prilagođavao novim zahtevima korisnika. Zbog toga, jednom urađen softver, ceo svoj radni vek najčešće prolazi kroz razne vrste iymena i dopuna, tj. On je u stalnoj evoluciji. Evolucija softvera je stalna promena softvera posle njegovog inicijalnog razvojai isporuke naručiocu, ili tržištu. . Neke analize pokazuju, da

• oko 85 do 90% troškova vezanih za softver kompanije troše za evoluciju softvera.

• da troškovi evolucije čine dve trećine ukupnih troškova za softver

Kompanija koja je razvila takav softver najčešće predaje odgovornost za dalji razvoj i modifikacije softvera kompaniji koja ga je naručila i koja mora da ima odgovarajući kadar za takav razvoj. Druga varijanta je da kompanija angažuje neku drugu organizaciju da podržava evolutivan razvoj njenog softvera.

Iz navedenih razloga, treba posmatrati softversko inženjerstvo kao jedan spriralni proces sa zahtevima, projektovanjem, implementacijom i testiranjem tokom celog njegovog životnog veka.

* 1. Koja je razlika između evolucije softvera i njegovog servisiranja? Šta pokreće proces evolucije softvera? Koji su razlozi za to? Objasnite ciklični karakter evolucije softvera. Kada se prihvataju predložene promene? Kada se publikuje (objavljuje) novo izdanje softvera? Koje su specifičnosti evolucije softvera kod primene agilnih metoda razvoja softvera?

Evolucija je faza u kojoj se vrše značajne promene u arhitekturi softvera i njegovoj funkcionalnosti. Za vreme servisiranja, vrše se sitnije promene, ali neophodne promene

Firma koja je inicijalno razvila softver, može da sklopi ugovor sa drugom firmom koja onda preuzima posao evolucije tog softvera. To može da dovede do prekida u procesu, tj. do diskontuiniteta. Na primer, može da se desi, da dokumentacija o zahtevima i projektnom rešenju ne bude poznata organizaciji – partneru, koja vrši dalju evoluciju softvera. Komapnije, takođe, se mogu spajati ili reorganiyovati, što dovodi do uvođenja novog softvera., npr. firme sa kojom se spajaju. Onda, najčešće utvrde da moraju da izvrše određene promene na softveru.

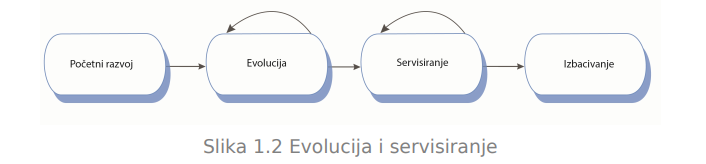
Kada prelazak iz faze razvoje softvera, u fazu evolucije posle isporuke softvera, nije neprimetan, te se ta faza često naziva “održavanjem softvera”.

Na slici 2 prikazan je alternativan model evolucije softvera tokom njegovog životnog veka. Model razlikuje fazu evolucije od faze servisiranja. Evolucija je faza u kojoj se vrše značajne promene u arhitekturi softvera i njegovoj funkcionalnosti. Za vreme servisiranja, vrše se sitnije promene, ali neophodne promene.

Za vreme evolucije, softver se uspešno koristi, i paralelno postoji tok stalnih promena u zahtevima. Međutim, sa promenama softvera, menja se i njegova struktura, što posetepeno dovodi do njegove degradacije, tj. opadanja performansi softvera.

Promene postaju teže i samim tim, i skuplje. To se obično dešava posle nekoliko godina korišćenja softvera, kada se vrše i promene hardverskog i softverskog okruženja u kome softver radi. U jednom momentu, značajne promene softvera, zbog novih zahteva, postaju sve manje isplative. U tom momentu, sistem prelazi iz faze efolucije u fazu servisiranja.

Za vreme faze servisiranja, softver je i dalje koristan, ali se na njemu rade samo male, neophodne izmene. U to vreme, kompanija počinje da razmišlja o zameni softvera novim. U konačnoj fazi (phase-out), softver se i dalje koristi, ali se vrše ne vrše nikakve izmene na njemu. Korisnici moraju sami da reše probleme na koje naiđu.

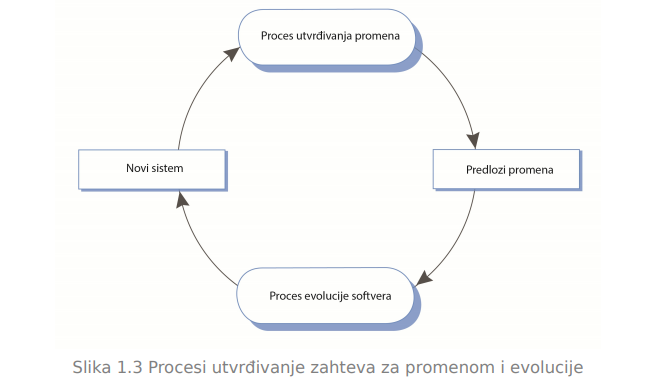


* 1. Nacrtajte proces evolucije softvera i objasnite njegove faze/aktivnosti. Zašto u njemu ima iteracija? Šta bi moglo da se menja, u idealnom slučaju, u softveru, za vreme njegove implementacije? Zbog čega se softver menja i posle početka njegove primene? Koje mogu biti negativne posledice menjanja softvera u toku njegove primene? Koja su dve vrste problema mogu javiti kada različiti timovi razvijaju softver i vrše njegovu evoluciju?

Procesi evolucije softvera zavise od tipa softvera, od proces razvoja softvera koji se koriste u organizaciji i od veština ljudi koji u tome učestvuju. U nekim organizacijama, evolucija softvera je neformalni proces u kome se zahtevi za promenama formulišu razgovorima između korisnika softvera i inženjera razvoja. U drugim organizacijama, to može biti jedan formalizovan proces, sa strukturisanom dokumentacijom u svakoj fazi procesa.

Glavni pokretač evolucije softvera u svim organizacijama su predlozi za promenu sistema. Oni dolaze iz potrebe da se realizuju još nerealizovani zahtevi sistema, zbog utvrđenih novih zahteva, zbog izveštaja o greškama u sistemu, a i zbog javljanja novih ideja za poboljšanje softvera od strane razvojnog tima. Procesi utrvrđivanja zahteva sistema evolucije su cikličnog karaktera i ponavljaju se tokom životnog ciklusa sistema (slika 3)

Zahtevi za promenom softvera bi trebalo da ukazuju na komponentu softvera koja se treba modifikovati. Na taj način se mogu proceniti troškovi realizacije te promene kao i njen efekat.



* 1. Navedite pet zakonitosti evolucije softvera. Ukratko ih objasnite.
  2. Navedite i ukratko opišite Lemanove zakone evolucije softvera.

Lehman-ovi zakoni iskazuju izvesne uočene zakonitosti u evoluciji softvera. Pojedini istraživači (Lehman i Belady) uočili su izvesne zakonitosti u evoluciji softvera:

1. Održavanje sistema je jedan neizbežan proces. Kako se okruženje sistema menja, javljaju se novi zahtevi i sistem se mora menjati. Kada se novi sistem aktivira, on utiče na okruženje, te se i ono menja, te proces evolucije nastavlja svoj ciklični razvoj.

2. Kako se sistem menja, tako se njegova struktura pogoršava. Preventivno održavanje je jedini način da se ovo izbegne. To znači da je neophodno da se investira vreme i novac za promenu strukture softvera bez promene njegove funkcionalnosti.

3. Veliki sistemi imaju svoju dinamiku koja je postavljena u ranoj fazi procesa razvoja. To određuje dalji trend procesa održavanja softvera i ograničava broj mogućih promena. Zakon je posledica uticaja strukturnih faktora koji utiču i ograničavaju promene sistema, i organizacionih faktora (brzina donošenja odluka) koji utiču na proces evolucije.

4. Veliki projekti programiranja rade u stanju „zasićenja“. To znači da neka promena u resursima ili ljudima ima neprimetan uticaj na evoluciju sistema u dužem periodu. To znači, kao i u 3. Zakonu, da evolucija programa je dosta nezavisna od odluka menadžmenta. Zakon potvrđuje da su veliki timovi razvoja često neproduktivni zbog složenih komunikacija koje dominiraju nad njihovim radom.

5. Dodavanje nove funkcionalnosti sistemu neminovno unosi nove greške sistema. Što više novih funkcija unosite, unosite i više grešaka u sistem. To traži ubrzo novo izdanje softvera sa ispravkama ovih grešaka. Znači, ne pravite inkremente sa velikim promenama funckionasti ako niste spremni da ubrzo pravite novo izdanje da bi ispravili unete greške.

* 1. Šta je održavanje softvera? Koji tipovi održavanja softvera postoje? Kakvi su troškovi održavanja u odnosu na troškove razvoja softvera i šta izaziva najveće troškove održavanja? Zašto su troškovi održavanja ugrađenih sistema i četiri puta veći nego troškovi razvoja softvera? Kako se troškovi održavanja mogu smanjivati? Koji softver brže stari: razvijen agilnim metodama ili planski razvijen softver? Zašto?

Održavanje softvera je proces menjanja sistema posle njegove isporuke Održavanje softvera je proces menjanja sistema posle njegove isporuke. Promene obuhvataju otklanjanje grešaka u programiranju, ali i veće greške koje otklanjaju greške u projektnom rešenju ili u specifikaciji sistema, ali u zadovoljenju novih zahteva. Implementacija promene postojećeg sistema se primenjuje promenom postojećih komponenti sistema, ali i dodavanjem novih. Postoji tri tipa održavanja softvera:

1. Popravka greški: Otklanjanje greški u kodu (najmanji troškovi popravke), u projektnom rešenju (veći troškovi) jer izaziva promenu nekoliko komponenti, i u zahtevima (najveći troškovi), jer može da zahteva i redizajn sistema.

2. Prilagođenje okruženju: Ova promena u softveru je izazvana promenama platforme i okruženja u kome radi sistem (hardver, softver).

3. Dodavanje funkcionalnosti: Promene u softveru su nužne zbog zadovoljenja novih organizacijskih ili poslovnih promena. Ove promene sistema su obično znatno ozbiljnije nego u prethodna dva slučaja.

TROŠKOVI ODRŽAVANJA SOFTVERA

Ukupni troškovi održavanja softvera u toku njegovog životnog veka mogu da se smanje u odnosu na troškove razvoja ako se o održavanju vodi računa još u fazi razvoja softvera

U slučaju tzv. ugrađenih sistema koji rade u realnom vremenu, troškovi održavanja su i četiri puta veći nego troškovi razvoja. Razlog za ovo zadovoljenja zahteva pouzdanosti i performansi koje prouzrokuju visok stepen integracije njihovih komponenata, te zbog toga, troškovi održavanja su veći. Doduše, primenom objektno-orijentisanih programiranja u razvoju ugrađenih sistema dovodi do smanjivanju troškova održavanja jer olakšavaju promenu njihovih komponenti. S ciljem da se smanje troškovi održavanja softvera, ulažu se napori da se razviju metodi razvoja softvera koji će voditi računa o održavanju još u fazi razvoja softvera (npr. primena tehnike dobijanje precizne specifikacije, primena objektno-orijentisanog razvoja, upravljanje konfiguracijom softvera).

Kako oceniti složenost odnosa između sistema i okruženja? Uzmite u obzir sledeće:

1. Broj i složenost sistemskih interfejsa: Što ih je više, i što su složeniji, veća je verovatnoća da će se menjati.

2. Broj nasledno nestabilnih sistemskih zahteva: Zahtevi koji odražavaju politiku organizacije i procedure, uvek se češće menjaju nego zahtevi koji se odnose na stabilne karakteristike u nekom području.

3. Poslovni procesu u kojim se sistem koristi: Kako se poslovni procesi razvijaju i menjaju, tako se generišu zahtevi za promenama. Što je veći broj poslovnih procesa koji ih upotrebljavaju, veći je broj zahteva za promenama.

Primenom metrike softvera, meri se njegova složenost. Na taj način se utvrđuju složenije komponente nekog sistema. Što su komponente složenije, to se teže održavaju. Da bi se smanjili troškovi održavanja, preporučljivo je da se složene komponente zamene sa jednostavnijim alternativnim rešenjima.

* 1. Zašto je skuplja nova funkcionalnost u održavanja nego u razvoju? Koji su razlozi za to? Kako se mogu predviđati troškovi održavanja? Kako oceniti složenost odnosa između sistema i okruženja? Koje su preporuke za smanjivanje troškova održavanja?

Skuplje je dodati novu funkcionalnost već postojećem softveru nego je ugraditi pri njegovom razvoju, iz sledećih razloga:

1. Stabilnost tima: Održavanje obično radi novi tim koji ne razume sistem tako dobro kao tim koji ga je razvio, te im treba više vremena za razumevanje sistema, pre nego što ga menjaju.

2. Loša praksa razvoja: Ugovor o održavanju je obično nezavisan od ugovora o razvoju sistema. Taj posao može biti dat i nekoj kompaniji koji i nije razvila softver. Zato. kompanija koja razvija sistem, nema motiv da pri razvoju softvera vodi računa o njegovom kasnijem održavanju. Čak suprotno, razvojni tim je motivisan da smanji što više troškove razvoja, ne vodeći računa da to može da dovede do većih troškova održavanja u budućnosti.

3. Veštine zaposlenih: Zaposleni koji se bave održavanjem je obično manje iskusan i ne poznaje dobro oblast primene softvera. Pored toga, stari sistemi su često pisani u programskim jezicima koji su prevaziđeni, te ljudi u održavanju nemaju dovoljno iskustva sa tim jezicima. Sve ovo povečava troškove održavanja.

4. Starost programa i njegova struktura: S promenama u softveru, postepeno se kvari njegova struktura, te on postaje teži za razumevanje i održavanje. Stari sistemi nisu ni razvijeni primenom modernih tehnika razvoja softvera. Pri njihovom razvoju, njihova struktura nije optimizovana za razumevanje koda, već za njegovu efikasnost. Dokumentacija sistema je ili izgubljena ili je nekonzistentna. Stari sistemi ne koriste upravljanje konfiguracijama, te je često otežano nalaženje prave verzije njegovih komponenti.

Prva tri razloga su vezana za problem shvatanja u industriji softvera. Mnoge organizacije i dalje vide održavanje i razvoj softvera kao dve nezavisne aktivnosti. Održavanje često se smatra aktivnošću nižeg ranga, te se ne poklanja pažnja održivosti sistema u toku faze razvoja sistema. Četvrti problem je lakše rešiv, jer se primenom odgovarajućih softverskih tehnika može se uspešno održavati struktura softvera u toku njegovog životnog veka, a i razumljivost sistema.

* 1. Kakva je metrika procesa potrebna da bi se ocenila sposobnost održavanja sistema? Objasnite posebno svaki od načina za ocenu sposobnosti održavanja softverskog sistema.

Kakva je metrika procesa potrebna da bi se ocenila sposobnost održavanja sistema? Evo nekih primera:

1. Broj zahteva za korektivno održavanje: Ako je broj izveštaja o greškama veći od broja ispravljenih grešaka programa u toku procesa održavanja, onda je održavanje sistema lošije, tj. opada njegova sposobnost održavanja.

2. Prosečno vreme potrebno za izradu anlalize uticaja promena: Ovo odražava broj komponenata koje su pod uticajem neke promene. Ako vreme da se ovo utvrdi raste, to ukazuje da je sve veći broj komponenata zahvaćen promenama, te se smanjuje sposobnost održavanja sistema.

3. Prosečno vreme za realizaciju zahteva za promenom: Ako vam je potrebno više vremena da realizujete (primenite, implementirate) sistem i njegovu dokumentaciju, ukazuje da sposobnost održavanja sistema opada.

4. Broj izuzetnih zahteva za promenama: Ako se povećava ovih vanrednih zahteva, onda to ukazuje da se smanjuje sposobnost održavanja sistema.

* 1. Reinženjering softvera: Zašto softver stari? Šta je reinženjering softvera? Koja je korist od primene reinženjeringa softvera? Nacrtajte dijagram procesa reinženjeringa starog Objasnite svaku aktivnost tog procesa. softvera. Da li u praksi moramo uvek da sprovedemo sve navedene aktivnosti? Zašto?

Stari programi vremenom, zbog zastarele tehnologije, ili zbog menjanja programa tokom njihove evolucije, postaju spori i neefikasni. Pored toga, mnoge stare sisteme, teško je razumeti i menjati. Oni su optimizirani za performanse, ili za korišćenje memorijskog prostora, a na račun jasnoće koda. Vremenom, početna struktura se znatno promeni i uruši stalnim promenama softvera.

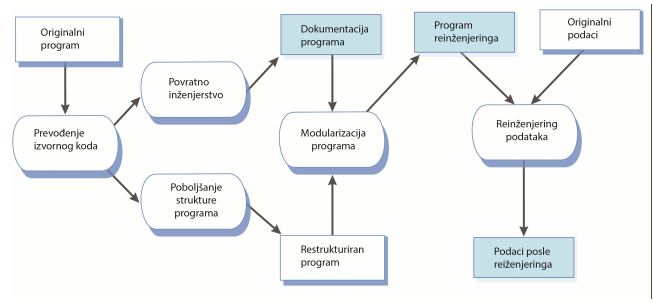
Da bi ovi stari sistemi postali lakši za održavanje, potrebno je izvršiti reinženjering sistema kako bi se poboljšala struktura sistema i njegova razumljivost. Reinženjering softvera je promena strukture arhitekture sistema, ponovno pisanje koda primenom modernih programskih jezika, promena struktura i vrednosti podataka sistema, kao i promena strukture i vrednosti sistemskih podataka:: .

Funkcionalnost softvera se ne menja, i treba izbeći da se vrše velike promene i arhitekturi sistema. Na taj način se mogu ostvarite dve koristi od reinženjering softvera umesto njegove zamene potpuno novim sistemom.

1. Smanjiti rizik: Razvoj i primena potpuno novog sistema, a naročito ako je on kritičan, je uvek rizični potez, jer se mogu javiti greške koje mogu da odlože njegovu primenu. Sa reiženjeringom starog softvera, taj rizik je niži.

2. Smanjeni troškovi: Troškovi reinženjeringa mogu biti višestruko niži nego troškovi razvoja novog sistema.

Proces reeinženjeringa starog softvera sadrži niz aktivnosti koje poboljšavaju performanse starog softvera. Slika 1 prikazuje opšti model procesa reinženjeringa softvera. Na ulazu je stari program a na izlazu je poboljšan i restrukturiran isti program.



Proces reinženjeringa sadrži sledeće aktivnosti:

1. Prevođenje izvornog koda: Uz pomoć odgovarajućeg alata, vrši se konverzija sa nekog starog programskog jezika u noviju verziju istog jezika ili u neki potpuno drugi i savremeniji programski jezik. 2. Povratno inženjerstvo: Analizom izvornog koda radi se programska dokumentacija, tj. opisuje se njegova organizacija i funkcionalnost. Često se i ova aktivnost može da automatizuje.

3. Poboljšanje strukture programa: Analizira se struktura programa i vrši se njena promena da bi se program lakše čitao i razumeo. Ovaj proces se može samo delimično automatizovati.

4. Modularizacija programa: Grupišu se međusobno povezani delovi programa i eliminišu se, redundantnosti u sistemu (ponavljanja koda). U nekim slučajevima može se modifikovati i struktura programa. Na primer, umesto nekoliko, koristi se samo jedna baza podataka. Ovo je ručni proces.

5. Reinženjering podataka: U skladu sa promenama u programu, menjaju se i podaci. To obuhvata i promenu šeme baze podataka i konverziju postojećih baza podataka u nove strukture. Tada se i podaci „čiste“, tj. uklanjaju se pogrešni podaci, dupli slogovi podataka i dr. Postoje alati za podršku reinženjeringa podataka.

* 1. Šta je restrukturiranje softvera? Koja je razlika restrukturiranja i reinženjeringa? Koja je specifičnost primene restrukturiranja kod agilnih metoda razvoja softvera? Kada treba primeniti restrukturiranje koda? Šta je i kada se primenjuje restrukturiranje projektnog rešenja?

Restrukturiranje jer proces poboljšanja programa s ciljem da se uspori njegova degradacija (opadanje performansi, povećanje složenosti, otežano snalaženje radi uvođenja promena i dr.) usled promena kojim je vremenom izložen. Restrukturiranje je poboljšanje strukture programa, smanjivanje njegove složenosti i olakšavanje njegovog razumevanja. Restrukturiranje ne dodaje novu funkcionalnost programu, već se sam program poboljšava.

U tom smislu, restrukturiranje je jedan od oblika „preventivnog održavanja“ softvera je smanjuje probleme koji se javljaju usled promena softvera u budućnosti. Koja je onda razlika restrukturiranja i reinženjeringa softvera?

1. Reinženjering se vrši posle izvesnog vremena održavanja softvera i povećanja troškova održavanja. Upotrebom automatskih alata za obradu i reinženjering starih sistema, vi kreirate novi sistem koji se lakše održava.

2. Restrukturiranje koda (refactoring code) je stalni proces poboljšanja za vreme razvoja i evolucije softvera. Sa njim mi izbegavamo degradaciju strukture i koda koji dovode do povećavanja troškova i do drugih teškoća održavanja sistema.

Poboljšanje softvera se vrši restrukturiranjem programskog koda, ali i projektnog rešenja, što je skuplji i složeniji metod poboljšanja, ali se primenjuje kada restrukturiranje koda nije dovoljno. Kada primeniti restrukturiranje? Odgovor: kada se pojavi neka od sledećih situacija:

1. Dupliranje koda: Uočavate isti ili vrlo sličan kod na različitim mestima u programu. Taj kod onda zamenjujete jednim metodom kojega onda pozivate na tim mestima.

2. Dugački metodi: Vidite metod koji je vrlo dugačak (puno linija koda). Odda ga zamenjujete se više manjih metoda.

3. Iskazi SWICH: U programu ima na dosae mesta iskaz „swich“. Kod objektnoorijentisanih sistema se oni mogu zameniti upotrebom polimorfizma 4. Cirkulacija podataka: Cirkulacija podataka se javlja kada se ista grupa podataka (polja u klasama, parametri u metodima) javljaju na više mesta u programu. To se otklanja njihovim učaurenjem u okviru jednog objekta. 5. Spekulativna uopštenost: Do ovoga dolazi kada programeri uključuju opštene iskaze u program

* 1. Šta raditi sa starim softverom? Koje su varijante? Kada koju primeniti?Kako postaviti strategiju evolucije softvera u vašoj firmi? Kakvu klasterizaciju možete primeniti kod starih softverskih sistema? Šta raditi sa svakim od karakterističnih klastera?

Starost programa i njegova struktura: S promenama u softveru, postepeno se kvari njegova struktura, te on postaje teži za razumevanje i održavanje. Stari sistemi nisu ni razvijeni primenom modernih tehnika razvoja softvera. Pri njihovom razvoju, njihova struktura nije optimizovana za razumevanje koda, već za njegovu efikasnost. Dokumentacija sistema je ili izgubljena ili je nekonzistentna. Stari sistemi ne koriste upravljanje konfiguracijama, te je često otežano nalaženje prave verzije njegovih komponenti.

Proces reeinženjeringa starog softvera sadrži niz aktivnosti koje poboljšavaju performanse starog softvera.

* 1. Kako utvrditi poslovnu vrednost nekog softverskog sistema? Koje faktore uzimate u obzir? Objasnite svaki od faktora.

Poslovna vrednost softvera u nekoj organizaciji se određuje analizom frekvencije njegove upotrebe, brojem poslovnih procesa koje podržava, stepenom pouzdanosti i važnosti rezultata.

Kako utvrditi poslovnu vrednost nekog softverskog sistema? Potrebno je prvo da identifikujete zainteresovane aktere tih sistema i onda da im postavite niz pitanja o sistemu. U tim razgovorima, možete diskutujete o sledećim temama:

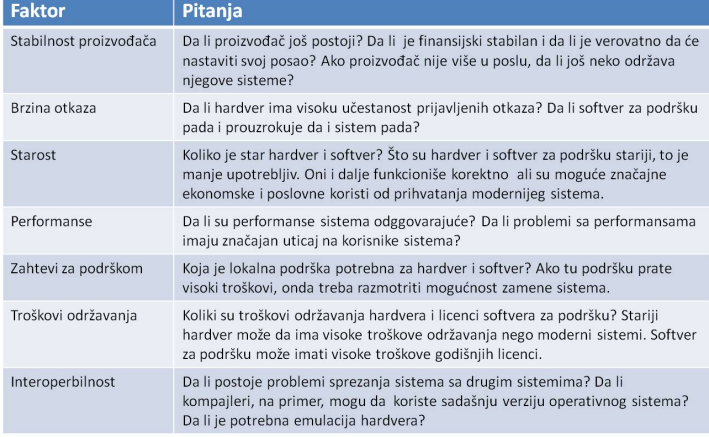
1. Upotreba sistema: Ako se sistemi samo povremeno koriste, ili ih koristi mali broj ljudi, onda oni imaju nisku poslovnu vrednost. To se često dešava ako je prvobitna uloga sistema prevaziđena promenama u poslovanju ili se ona sada realizuje na neki drugi, efektniji način. Međutim, ako je i povremeno korišćenje vrlo važno za organizaciju, onda se takav sistem ne odbacuje.

2. Poslovni procesi koje sistem podržava: Svaki sistem podržava jedan ili više poslovnih procesa. Ako je sistem nefleksibilan, i ovi procesi se ne mogu da menjaju. Međutim, zbog promene okruženja i poslovnih zahteva, vrlo često se poslovni procesi moraju da menjaju. Sistem koji ne može da podrži takve promene onda treba izbaciti iz upotrebe.

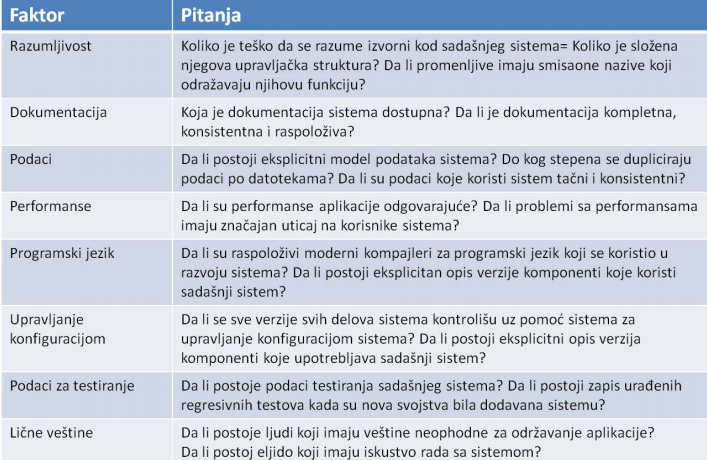
3. Pouzdanost sistema: Pored tehničke, postoji i poslovna pouzdanost. Ako sistem nije pouzdan, te direktno time ometa poslovne partnere, ili zahteva ulaganje resursa u rešavanje problema pouzdanosti, onda sistem ima nisku poslovnu vrednost.

4. Rezultati sistema: Treba utvrditi važnost rezultata koje proizvodi sistem na uspešno funkcionisanje poslovanja. Ako posao zavisi od ovih rezultata, onda sistem ima visoku poslovnu vrednost. S druge strane, ako se rezultati sistema retko koriste, ili se mogu dobiti i na drugi način, onda je poslovna vrednost sistema niska.

* 1. Treba da donesete odluku da li da zamenite stari softver sa novim. Jadan od parametara odlučivanja je i ocena okruženja u kome softver sada radi. Koje faktore koristite za ocenu okruženja? Koja tipična pitanja postavljate za svaki od faktora?



* 1. Treba da donesete odluku da li da zamenite stari softver sa novim. Jadan od parametara odlučivanja je i ocena kvaliteta softvera, tj. aplikacije koju ste do sada koristili. Koje faktore koristite da bi ocenili kvaliteta aplikacije? Koja tipična pitanja postavljate za svaki od faktora?



* 1. Koji su vam podaci potrebni da bi dali ocenu dosadašnjeg softverskog sistema. Ukratko objasnite zašto vam je potreban svaki od navedenih podataka. Koji su ostali uticajni faktori koji dovode do promene softverskog sistema ili do zadržavanja starog softvera.

Posle ocene kvaliteta samog softvera, tj. kvaliteta aplikacije, kao i uzimanja u obzir i faktora okruženja u kome softver radi, donosi se ocena softverski sistem s tehničkog stanovišta. Konačna odluka o zadržavanju ili menjanja postojećeg softverskog sistema zavisi od ocene njegove poslovne vrednosti i njegovog kvaliteta. U idealnom slučaju, pri donošenu odluke o sudbini postojećih sistema, trebalo bi se služiti isključivo objektivnim ocenama svih navedenih faktora. Međutim, često se odluke donose i na osnovu manje objektivnih ocena, jer se zasnivaju da organizacionim ili političkim faktorima. Navešćemo nekoliko primera:

• Pri spajanju dve kompanije, najčešće se zadržavaju sistemi jačeg partnera, a napuštaju drugi sistemi.

• Ako organizacija odluči da pređe na novu hardversku platformu, onda to zahteva promenu aplikacija.

• Ako nema potrebnog budžeta za transformaciju sistema, onda se nastavlja sa održavanjem starog sistema, iako to dovodi do, dugoročno gledano, do povećanih troškova

1. **Lekcija 14: AGILNI RAZVOJ SOFTVERA**
   1. Šta je to agilni metod razvoja? Zašto je došlo da razvoja agilnih metoda razvoja softvera? Koja su svojstva agilnih metoda? Šta je to specifično kod agilnih metoda razvoja softvera? Zašto je brzi razvoj i isporuka novih sistema često važnije za poslovanje nego detaljna funkcionalnost ovih sistema?

Poslovanje firmi se sve više odvija u okruženju koje se brzo menja. Tržište postaje sve zahtevnije, konkurencija sve veća, svi traže nešto bolje, a za manje novca, a naročito, svi žele da se nešto uradi što brže. Firme moraju da reaguju na te promene u što kraćem vremenu. Ko ne može brzo da reaguje, vrlo često gubi trku i nestaje sa globalnog tržišta, pa i iz poslovanja.

Ta trka za vremenom zahvatila je i industriju razvoja softvera. Klasičan postupak razvoja softvera obuhvata specifikaciju zahteve, projektovanje, pa razvoj softvera, a na kraju testiranje i isporuku softvera. Sve to je praćeno odgovarajućom dokumentacijom. Takva metodologija razvoja softvera je odgovarajuća za velika, a naročiti tzv. kritične sisteme, koji moraju da rade vrlo pouzdano i rade u stabilnim uslovima poslovanja. Međutim, ova metodologija, zato što zahteve dosta vremena za razvoj, pa i dokumentovanje softvera, nije odgovarajuće za razvoj softvera namenjen poslovanju firmi, tj. pri razvoju tzv. poslovnih aplikacija. Kao što je rečeno, poslovno okruženje se brzo menja, te se i zahtevi koje softver treba da ispuni, a kojj su definisani danas, već za nekoliko meseci možda neće biti ispravni, jer su se uslovi u poslovnom okruženju promenile. Pa kako onda razviti softver, kada kupac stalno menja zahteve?

Krajem 90-tih godina, postavljene su osnove tzv. brzog razvoja softvera s ciljem da se brzo dobije koristan softver. Softver se ne dobija kao jedna jedinstvena jedinica, već se dobija u seriji inkremenata, pri čemu svako softverski inkrement sadrži neku novu funkcionalnost sistema. Na tom principu, razvijeno je više metoda razvoja softvera koje dele sledeće zajednička svojstva:

1. Procesi pripreme specifikacije, projektovanja i implementacije su izmešani. Ne postoji detaljna specifikacija, a projektna dokumentacija je minimizirana ili je automatski generisana softverskim alatima za razvoj softvera. Dokument o zahtevima korisnika definiše samo najvažnije karakteristike sistema.

2. Sistem se razvija u vidu serije verzija. Korisnici i druge zainteresovane strane definišu zahteve za svaku verziju. Oni predlažu promene softvera i nove zahteve, koje se onda primenjuju u kasnijim verzijama softvera. 3. Korisnički interfejsi sistema se razvijaju primenom interaktivnih sistema za njihov razvoj koji omogućavaju brzo kreiranje grafičkog interfejsa i raspoređivanje ikona i drugih elemenata interfejsa. Ti sistemi generišu interfejse za prikazivanje na veb pretraživačima ili za posebne platforme kao što su MS Windows ili Mac.

* 1. Šta je karakteristično za agilne metode razvoja softvera? Za koje vrednosne stavove sa zalaže tzv. Agilni manifest? Koji su principi agilnih metoda? Šta je to Sprint ciklus?

Agilne metode primenjuju inkrementalni razvoj specifikacije, projektnog rešenja (dizajna), koda, testiranja i isporuke softvera. One eliminišu procesnu birokratiju, jer se usmeravaju ka razvoju softvera, a ne dokumentacije. Agilne metode se filozofski oslanjaju na tzv. „agilni manifest“ koji je dokument oko koga su se složili mnogi proizvođači softvera. Manifest definiše sledeće: “Mi otkrivamo bolje načine razvoja softvera primenjujući ih i pomažući drugima da to isto urade. Radeći, mi smo došli do sledećih vrednosnih stavova:

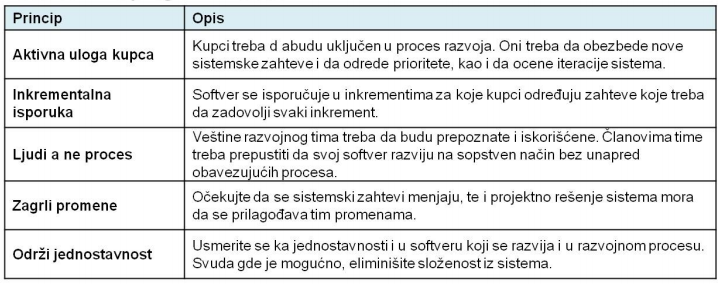
• Pojedinci i interakcije u odnosu na procese i alate

• Softver u radu u odnosu na odgovarajuću dokumentacija

• Saradnja sa kupcem umesto pregovori oko ugovora

• Reakcija na promene umesto realizacije plana

Ovde, u odnosu na vrednosti na desnoj strani, mi vrednujemo više vrednosti na levoj strani ovih iskaza.” Na osnovu ovih principa, razvijeno više različitih agilnih metoda (ekstremno programiranje, Scrum, Crystal, i dr.). Došlo je i do njihove integracije sa tradicionalnim metodama koje se oslanjaju na modelovanje (agilno modelovanje, agilni RUP). Svi ovi metodi dele iste principe prikazane u prikazanoj tabeli (slik 1).:



Scrum sprint ciklus je jedinica planiranja u kojoj se radi neki posao koje se ocenjuje, sa svojstvima koji su odabrani za razvoj, i koji se softverski primenjuje

Glavna inovacija Scrum postupka je u njegovoj centralnoj fazi – spint ciklusima. Scrum sprint ciklus je jedinica planiranja u kojoj se radi neki posao koje se ocenjuje, sa svojstvima koji su odabrani za razvoj, i koji se softverski primenjuje. Na kraju sprinta, isporučuje se završena nova funkcionalnost. Proces ima sledeća svojstva:

1. Sprint ciklus je fiksirane dužine trajanja, najčešće 2 – 4 nedelje. Oni odgovaraju razvoju nove verzije softvera u XP. 2. Početna tačka planiranja je spisak poslova koji treba da se uradi u projektu (tzv programski log). U fazi ocenjivanja, ovo se analizira i dodaju se prioriteti i rizici svakom planiranom poslu. Kupac je uključen u ovaj proces i može da postavi nove zahteve ili zadatke na početku svakog sprinta.

3. Faza izbora obuhvata ceo tim koji radi sa kupcem na izboru svojstava i funkcionosti koje treba da se razviju za vreme jednog ciklusa printa.

4. U ovoj fazi se vrši razvoj dogovorenih svojstava i funkcionalnosti. Drže se kratki dnevni sastanci razvojnog tima na kojima se razmatra napredak posla, i vrši eventualna promena prioriteta. U ovoj fazi je tim izolovan od kupca i organizacije, a sva komunikacija sa njim ide preko člana tima koji ima ulogu “Scrum mastera“. Njegova uloga je da zaštiti tim od spoljnih uticaja i smetnji. Scrum ne određuje način specifikacije zahteva, pisanja testova i dr.

5. Urađen posao se analizira i predstavlja zainteresovanim stranama. Posle toga, novi sprint ciklus može da počne.

* 1. Kada se primenjuju agilne metode? Koje su poteškoće u primeni agilnih metoda? Kako se mogu primeniti agilne metode u održavanju i koje su prisutne teškoće?

Zbog odsustva jasnog i kompletnog dokumenta koji definiše zahteve, što je obično deo ugovore, softverske kuće imaju poteškoće da angažuju druge firme kao podizvođače. Agilne metode uspešno se primenjuju pri razvoju softverskih sistema sledećeg tipa:

1. Razvoj proizvoda koji softverska kompanija razvija za prodaju, a primenom projekta male ili srednje veličine.

2. Razvoj sistema za kupca, pri čemu je kupac jasno opredeljen da aktivno učestvuje u razvoju i gde nema mnogo spoljnih pravila i ograničenja koji se odnose na softver.

Nije uvek lako u praksi sprovesti principe koje koriste agilne metode:

1. Zbog svojih drugih obaveza, predstavnici kupca softvera nisu uvek u mogućnosti da posvete dovoljno vremena radu sa razvojnim timom.

2. Ponekad, članovi tima nemaju kapacitet da mogu intenzivno da rade, što je uobičajeno kod primene agilnih metoda, te ne ostvaruju dobru saradnju sa drugim članovima tima.

3. Određivanje prioriteta u uvođenju promera nije jednostavno, jer različite zainteresovane strane imaju svoje, međusobno suprotstavljene prioritete.

4. Ostvarivanje i održavanje jednostavnosti sistema zahteva dodatan napor i rad. Pod pritiskom rokova, razvojni tim nema uvek dovoljno vremena da traži najednostavnija rešenja.

5. Mnoge organizacije, a naročito vremena, uložile su velike napore i godine da bi navikli zaposlene da poštuju postavljenje procese. Sada im je teško da ih usmere da ne slede formalne, već neformalne procese definisane od strane samog razvojnog tima.

Zbog odsustva jasnog i kompletnog dokumenta koji definiše zahteve, što je obično deo ugovore, softverske kuće imaju poteškoće da angažuju druge firme kao podizvođače, u slučajevima kada primenjuju agilne metode. Zbog toga se u takvim slučajevima ugovara vremensko angažovanje ocenjeno da je neophodno za završetak nekog posla. Međutim, ako se u očekivanim rokovima ne završi taj posao, teško je onda proceniti ko je za to kriv, i na čiji račun treba da ide dodatan rad za završetak posla.

Ukjlučivanje korisnika sistema i česte promene u sastavu razvojnog tima – su dva najveća problema u primeni agilnih metoda u održavanju sistema.

Pitanje je logično, jer se razvoj odvija sa minimumom formalne dokumentacije, a ona je osnov za ljude koji dobijaju zadatak da promene nešto u sistemu, a nisu na njemu radili. Međutim, poznato je da u praksi, i kada se koristi formalna dokumentacija, ona najčešće nije ažurna, tj. osvežena promenama koje su u međuvremenu vršene. Promoteri agilnih metoda zato tvrde da nije važna formalna dokumentacija i da na njoj ne treba gubiti vreme, već je važno napraviti visoko kvalitetan, dobro strukturisan kod, koji je čitljiv i jednostavan za održavanje. Oni smatraju da se napor treba usmeriti ka dobijanju takvog koda, a ne ka dokumentaciji. Smatraju da nedostatak dokumentacije, ne bi trebalo da bude problem u održavanju koda, ako je on dobro napravljen. Međutim, ipak se smatra da bez dobrog dokumenta koji definiše zahteve, teško je oceniti posledice predloženih promena u sistemu. Zato, primena agilnih metoda u održavanju softvera ipak otežava njegovo kasnije održavanje i čini ga skupljim

Praksa je ipak pokazala, da se agilni metodi mogu uspešno primeniti i u održavanju, jer inkrementalne isporuke popravki postojećeg sistema je prirodan način evolucije softvera. Glavni je problem uključiti korisnika sistema u rad na popravkama sistema, i u fazi evolucije softvera.

Drugi problem kod primene agilnih metoda u održavanju je u promenama razvojnog tima. Ako tih ljudi više nema, sa njima je otišlo i njihovo iskustveno znanje i poznavanje sistema, a novi ljudi, u odsustvu dokumentacije, imaju problem da se snađu. Ima predloga da se koriste hibridne metoda, koje spajaju najbolje strane agilnih metoda i planom-vođenih metoda razvoja softvera.

* 1. Objasnite kako principi na kojima se zasnivaju agilni metodi ubrzavaju razvoj softvera?
  2. Kada nećete preporučiti primenu agilnog metoda razvoja softverskog sistema i zašto?

Najveći broj projekata razvoja softvera koriste praksu i planom vođenim i agilnog pristupa razvoju softvera. Ustvari, najveći broj projekata razvoja softvera koriste praksu i planom vođenim i agilnog pristupa razvoju softvera. Pri određivanja ravnoteže u primeni metoda i jednog i drugog pristupa, imaju se u vidu i sledeći tehnički, ljudski i organizacioni aspekti:

1. Da li važno da se ima detaljna specifikacija i projketno rešenje pre nego što se počne sa implementacijom? Ako je to tako, onda treba primeniti planom vođeni pristup razvoju.

2. Da li je ostvarljivo da primenite inkremenalnu isporuku softvera kupcu, i da li od njega očekujete brz odgovor? Ako je to tako, onda možete koristiti agilne metode.

3. Koliko je veliki sistem koji se razvija? Agilne metode su najefektivnije kada se sistem razvija sa malim timom koji mogu da neformalno komuniciraju. To nije moguće u slučaju velikih sistema koji zahtevaju veliki razvojni tim, tako da je u tom slučaju plom vođen pristup izgledniji.

4. Koji se tip sistema razvija? Sistemi koji zahtevaju puno analiza pre implementacije (npr. sistemi u realnom vremenu sa složenim vremenskim zahtevima) obično imaju potrebu za vrlo detaljnim projektovanjem radi izvršenja ove analize. U takvom slučaju, planom vođen pristup je najbolji.

5. Šta je očekivani životni ciklus sistema? Dugoživeći sistemi obično zahtevaju više projektne dokumentacije radi komunikacije sa originalnim namerama tima koji je razvio sistem i tima koji ga održava. Međutim, tim za održavanje često ima primedbe da je ta dokumentacija neažurna i da zbog toga nije od velike koristi za održavanje sistema.

6. Koje su raspoložive tehnologij za podršku razvoja sistema? Agilne metode često zavise od dobrih alata za održavanje verzija projektnog rešenja softvera. Ako se ne koristi dobar alat (IDE) za vizualizaciju programa i njegovu analizu, onda je potrebna obimnija projektna dokumentacija.

7. Kako je projektni tim organizovan? Ako su članovi tima na različitim lokacijama, ili ako u razvoju učestvuje i neka druga organizacija, onda je najčešće neophodno izraditi projektnu dokumentaciju radi komunikacije između udaljenih razvojnih grupa i pojedinaca. To se mora unapred planirati. 8. Da li postoje kulturološki aspekti koji mogu da utiču na razvoj sistema? Tradicionalne inženjerske organizacije obično imaju kulturu negovanja planom vođenog razvoja, jer je to norma u inženjerstvu. To obično zahteva obimnu dokumentaciju, što nije slučaj kod agilnih procesa kada se upotrebljava neformalno znanje.

9. Koliko su dobri projektanti i programeri razvojnog tima? Obično primena agilnih metoda zahteva ljude većih sposobnosti nego primena plnom vođenih pristupa. Ako to nije slučaj, onda se najbolji rudi koriste za projektovanje,a ostali , za programiranje.

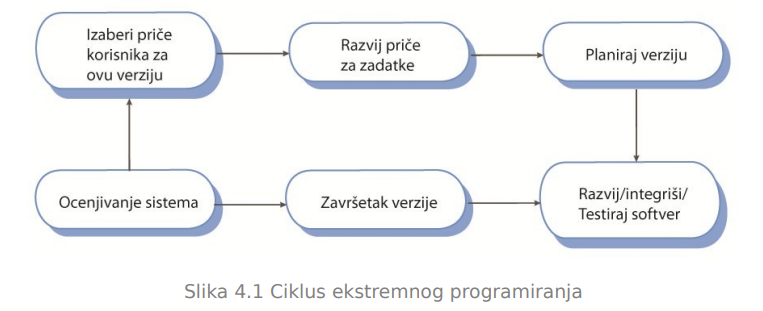
10. Da li je sistem zavistan od spoljne regulacije (zakona, propisa, standarda? Ako sistem mora da bude prihvaćen od neke regulatorne agencije, onda je najčešće potrebno da se podnese detaljna dokumentacija sistema, radi provere njegove sigurnosti, odnosno, zadovoljenja pravila.

U praksi, mnoge firme koje tvrde da primenjuju agilne metode u razvoju, primenjuju samo neke elemente agilne prakse koju integrišu u njihove planom vođene procese

* 1. Šta je ekstremno programiranje? Nacrtajte i objasnite ciklus ekstremnog programiranja.

Kod ekstremnog programiranja, zahtevi se izražavaju u vidu scenarija a programeri rade u parovima i razvijaju testove za svaki zadatak pre nego što napišu kod.

Ekstremno programiranje (XP) je najčešće primenjivan i najbolje poznat agilni metod. Kod ekstremnog programiranja, zahtevi se izražavaju u vidu scenarija (koji se nazivaju korisničkim pričama) koji se direktno primenjuju u vidu serije zadataka. Programeri rade u parovima i razvijaju testove za svaki zadatak pre nego što napišu kod. Ovi testovi moraju da se uspešno izvrše kada je novi kod integrisan sa sistemom. Postoji vrlo kratko vreme između susednih verzija sistema. Slika 1 ilustruje XP proces koji proizvodi jedan inkrement sistema koji je u razvoju.



Ekstremno programiranje odražava sledeće principe agilnih metoda:

1. Inkrementalni razvoje je podržan preko malih, ali čestih verzija sistema. Zahtevi nastaju iz jednostavnih priča korisnika ili scenarija koji se koristi kao osnova za odlučivanje o tome koja funkcionalnost treba da se uključi u inkrement sistema

2. Korisnik sistema je uključen njegovim stalnim angažovanjem u razvojnom timu. On je u timu odgovoran za definisanje testova prihvatanja sistema.

3. Ljudi, a ne procesi, su podržani programiranjem u paru, kolektivnom pripadanju koda sistema, i održivom razvojnim procesom koji ne zahteva vrlo dugo vreme rada.

4. Promene su podržane regularnim izdavanjem novih verzija sistema kupcima, razvojem u kome su testovi na prvom mestu, restrukturiranjem sistema da bi se izbegla njegova degeneracija, i stalna integracija novih funkcionalnosti (funkcija).

5. Održavanje jednostavnosti koje poboljšava kvalitet koda i u kome se upotrebljava jednostavna projektna rešenja koja ne ograničavaju buduće promene sistema.

* 1. Koji principa agilnog programiranja se primenjuju pri ekstremnom programiranju?
  2. Navedite i objasnite principe ekstremnog programiranja. Otkud naziv „ekstremno“ programiranje? Šta je tu „ekstremno“?

Kod XP procesa, korisnici si vrlo uključeni u specifikaciju zahteva sistema i u određivanje prioriteta. Zahtevi nisu specificirani u obliku lista zahtevanih funkcija sistema., već u vidu scenarija koji nastaju kao rezultat diskusije predstavnika korisnika i razvojnog tima. Oni zajedno razvijaju „kartu sa pričom” (engl. story card) koja sadrži potrebe korisnika sistema. Razvojni tim onda ima zadatak da primeni ovaj scenario u sledećoj verziji softvera. Sledeća tabela ilustruje kako XP proces proizvodi jedan inkrement sistema u razvoju.



* 1. Šta su i čemu služe karte sa pričama? Objasnite njihovu primenu.

Kada se karte sa pričama razviju, razvojni tim svaki kartu podeli na zadatke i onda procenjuje potreban rad i resurse za implementaciju.

Karte sa pričama su glavni ulaz u XP proces planiranja. Kada se one razviju, razvojni tim svaki kartu podeli na zadatke i onda procenjuje potreban rad i resurse za implementaciju (realizaciju) svakog zadatka

Ovo obično podrazumeva razgovor sa korisnicima radi poboljšanja zahteva. Korisnik onda određuje prioritete implementacije, izabirajući one koje se mogu odmah primeniti i koje obezbeđuju korisnu podršku poslovanju. Namera je da se odredi korisna funkcionalnost koja se može primeniti u roku oko dve nedelje, kada je vreme za izbacivanje nove verzije sistema, i koja se šalje korisniku Ako dođe do promena zahteva, odbacuju se ne primenjene krte.

Ako se zahtevane promene odnose na već primenjen sistem, razvijaju se nove karte, i opet, kupac odlučuje o prioritetima uvođenja novih funkcionalnosti.

U slučajevima kada razvojni time ne može lako da dođe do rešenja, određuje se dodatno vreme za traženje rešenja. Za to vreme, moguće je i razvoj prototipa rešenja da bi se vršile provere mogućih rešenja.

Ekstremno programiranje se naziva „ekstremnim“ pristupom inkrementalnom razvoju jer se nove verzije softvera mogu izrađivati i nekoliko puta u toku dana, a konačna verzija, tj. Izdanje koja se šalje kupcu, isporučuje s ekupcu svake dve nedelje. Rokovi se nikada ne menjaju, a ako ima problema, uz konsultaciju sa kupcem, sporna funkcionalnost, tj. sporni deo softvera se izostavlja iz planiranog izdanja softvera, a ono što ostaje, šalje se kupcu.

* 1. Šta je problem inkrementalnog programiranja? Kako se on otklanja u slučaju primene ekstremnog programiranja?

Opšti problem kod inkrementalnog programiranja je da on dovodi do degradacije strukture softvera, tako da se buduće promene sve teže i teže implementiraju.

Kada programer razvija novu verziju, on mora da izvrši automatske testove i za sve postojeće softverske jedinice, kao i za nove, koje obezbeđuju novu funkcionalnost. Nova verzija se prihvata samo ako svi testovi daju pozitivne rezultate. Takva verzija postaje onda osnova za sledeći ciklus promena softvera. U ekstremnnom programiranu se ne primenjuje princip po kome se projektovanje vrši tako da se omoguće lake buduće promene, jer se smatra da to zahteva dodatno vreme u razvoju, koje se ne želi potrošiti za tu namenu. Kod XP pristupa, promene sistema se planiraju samo kada se te promene zahtevaju, tj. one se unapred ne planiraju.

Opšti problem kod inkrementalnog programiranja je da on dovodi do degradacije strukture softvera, tako da se buduće promene sve teže i teže implementiraju. To je posledica traženja rešenja za konkretnu promenu, ne vodeći računa o optimizaciji koda i njegovoj strukturi, te se desi da dođe i dupliranja koda , a i do kopiranja delova softvera na ne odgovarajući način, te dolazi do degradacije strukture kode kada se on dodaje u sistem.

Taj problem ekstremno programiranje rešavanja stalnim restrukturiranjem softvera. Razvojni tim razmatra načine da se poboljša struktura softvera i nađeno rešenje odmah primenjuje.

Kada neki član tima primeti da se kod može poboljšati, on to odmah radi, iako ta promena nije povezana za zahtevom za promenama na kojoj radi. Na primer, tako se može promeniti struktura klasa, da bi se uklonio dupliran kod, ili promeniti nazivi atributa i metod i zameniti kod sa pozivima metoda koji su definisani u programskoj biblioteci. Programski alati, kao što je Eclipse, podržavaju restrukturiranje koda jer nalaze zavisnost promena opšteg koda izabranog koda.

Iako bi trebalo da se na opisan način softver uvek održava u dobrom stanju, tako da je lako čitljiv i strukturisan, u praksi nije uvek tako. Pod vremenskim pritiskom da se primene nove funkcionalnosti, inženjeri razvoja ne posvećuju dovoljno vremena usavršavanju samog koda., sem kada nova funkcionalnost zahteva promenu strukture i arhitekture sistema.

U praksi, mnoge kompanije ne primenjuju ekstremno programiranje u skladu sa principima datim u tabeli, već primenjuju samo neke od njih, u skladu sa svojim načinom rada. Ali, ono što najčešće primenjuju, to su česte i malo promenjene verzije sistema, definisanje najpre testova, i stalna integracija.

* 1. Kako se vrši testiranje softvera u uslovima primene ekstremnog programiranja pri razvoju softvera? Kada i kako se definiše test? Dajte primer testa za neki zadatak u razvoju softvera.

Prvo se definiše test, pa se onda razvija kod. To omogućava testiranja programa još kada je u fazi razvoja. Ekstremno programiranje daje značaj testiranju. Osnovna svojstva testiranja u ekstremnom programiranju su sledeća:

1. Prvo se pripremaju testovi.

2. Razvijaju se inkrementalni testovi na osnovu scenarija.

3. Korisnik učestvuje u razvoj i validaciju testova.

4. Koriste se sistemi za automatsko testiranje.

Umesto da se test definiše posle pisanja koda, kod ekstremnog programiranja je obrnuto. Prvo se definiše test, pa se onda razvija kod. To omogućava testiranja koda još kada je u fazi razvoja.

Pisanjem testa se implicitno definišu i interfejs i specifikacija ponašanja za funkcionalnost koja se razvija. Ovaj pristup se može da primenu svuda gde postoji jasan odnos između sistemskih zahteva i koda koji ih primenjuje. Ta veza se kod ekstremnog programiranja jasno vidi jer karta sa pričom (scenarijom) predstavlja zahteve koji se razbijaju na zadatke, a zadaci su glavne jedinice implementacije.

* 1. Zašto se vrši automatizacija testova kod ekstremnog programiranja? Kako tetsovi prate novu funkcionalnost koju donosi svaka nova verzija softvera? Koji se problemi ovde mogu da jave?

Da bi se testiranje ubrzalo, izvršenje testova se automatizuje. Testovi se pišu kao izvršne komponente pre nego što se implementira zadatak. To su samostalne programske komponente koje obrađuju ulaze i proveravaju ulaze u skladu sa definisanim testom. Postoje alati za pisanje automatskih testova, tako da oni izvršavaju grupe testova. (na primer, Junit). Uvek kada se doda nova funkcionalnost, pokreće se izvršenje svih (i starih i novih) testova jedinica, tako da se može uočiti eventualno negativna posledica uvođenja novog koda u sistem. Iako se na ovaj način programi mogu vrlo detaljno i pouzdano testirati, u praksi nije uvek tako.

1. Programeri više vole programiranje nego testiranje, te pri definisanju testova koriste skraćene postupke testiranja, što nije dobro. Nekompletni testovi ne vrše proveru svih izuzetaka koji se mogu javiti u radu sistema

2. Neki testovi se teško mogu inkrementalno napisati. Na primer, složeni korisnički interfejsi se teško proveravaju testovima jedinica. Jedinični testovi nisu pogodni za proveru „logike prikazivanja“ i za radne tokove.

3. Teško je oceniti kompletnost izvršenih skupa testova. Možda i vrlo važan segment sistema nije obuhvaćen nekim testom.

Ako se ne vrši provera napisanih testova, i ako se ne radi na dopuni nedostajućih testova, onda se mogu javiti neotkrivene greške (tzv. bagovi) u sistemu.

* 1. Šta je to programiranje u parovima? Koju povoljnost ta praksa donosi? Kakva je produktivnost takvog rada? Navedite četiri razloga za ubrzavanje produktivnosti programera kada rade u paru za oko 50% u odnosu kada sva programera rade individualno.

Podržava ideju o osećanju kolektivnog autorstva, realizuje neformalna recenzija procesa razvoja i omaže rad na usavršavanu koda, tj. restrukturiranje, tokom razvoja izmena sistema Programiranje u parovima je česta praksa pri primeni ekstremnog programiranja. Programeri rade u paru te zajedno na istoj radnoj stanici razvijaju softver. Međutim, članovi para ne mora uvek da programiraju zajedno. Parovi se dinamički planiraju, te ne mora celo vreme da parovi budu isti. Raspored članova razvojnog tima, po parovima, se može menjati s vremena na vreme. Rad u parovima obezbeđuje sledeće povoljnosti:

1. Podržava ideju o osećanju kolektivnog autorstva nad svakom linijom koda. Tim, a ne pojedinac, preuzima odgovornost za ceo kod i svaki njegov deo, i zajedno radi na rešavanju problema.

2. Na ovaj način se realizuje neformalna recenzija procesa razvoja, jer svaku liniju koda gleda najmanje dva člana tima. Provera i recenzija koda se uspešno primenjuje na ovaj način. Međutim, zbog dodatnih vremenskih zahteva za ovakav način rada, rad u parovima zahteva duže vreme, i može da malo uspori projekat razvoja. Mada rad u paru nije tako efikasan kao formalna inspekcija, on je mnogo jeftiniji za realizaciju od formalne inspekcije.

3. Pomaže rad na usavršavanu koda, tj. restrukturiranje, tokom razvoja izmena sistema, što donosi koriste na duže staze.

Gledano po članu tima, rad u parovima za određeno vreme proizvede za 50% manje koda nego ako oni radi samostalno. Međutim, to ne mora da bude tako. Njihova produktivnost može biti slična produktivnosti ako rade samostalno, ako zajedničkim diskusijama na vreme otkriju moguće greške i koje mogu kasnije nastati kada počnu da kodiraju. To smanjuje broj kasnijih grešaka, a takođe i ubrzava otklanjanje kasnije otkrivanih grešaka.

Praksa ipak pokazuje da rad u parovima smanjuje produktivnost tima, ali povećava kvalitet rada. Takođe, daje stabilnost timu, jer ako ga napusti neki član tima, drugi član može bez problema da radi poslove na kojimaje radio član koji je otišao.

* 1. Uporedite planom vođeni razvoj i agilni razvoj softvera. Kako se pri oba pristupa realizuju iteracije u procesu razvoja? Šta je to hibridni pristup u razvoju softvera? Kada se on primenjuje?

Kod planski vođenog razvoja, iteracije se javljaju unutar aktivnosti sa formalnom dokumentacijom koja se koristi za komunikaciju između faza procesa. Na primer, na osnovu prikupljenih zahteva, kreira se specifakicija zahteva, a na osnovu nje, vrši se projektovanje i implementacija softvera. Kod agilnog pristupa, iteracije se dešavaju između aktivnosti, te se definisanje zahteva i projektovanje realizuju zajedno, a ne posebno, kao kod planom vođenog razvoja.

Međutim, oba pristupa mogu da primenjuju metode koje su osnova drugog pristupa, te tako nastaju hibridne metode. Na primer, planom vođen proces može da primenjuje i inkrementalni razvoj i isporuku. Na taj način, se dodeljivanje zahteva, i planiranje faze projektovanja i razvoja, realizuju preko serije inkremenata. Isto tako, nije nemoguće da agilni proces, koji je po prirodi usmeren na programiranje, proizvodi i neku projektnu dokumentaciju

* 1. Kako se može primeniti agilne metode razvoja softvera i u slučaju razvoja velikih softverskih sistema? Koje su specifičnosti razvoja velikih sistema? Koje su perspektive primene agilnih metoda kod razvoja velikih sistema? Koje su poteškoće u primeni kod velikih sistema?

Agilne metode su najefektivnije kada se sistem razvija sa malim timom koji mogu da neformalno komuniciraju. To nije moguće u slučaju velikih sistema koji zahtevaju veliki razvojni tim, tako da je u tom slučaju plom vođen pristup izgledniji

Veliki softverski sistemi imaju svoje specifičnosti koje ometaju primenu agilnh metoda u njihovom razvoju. Agilne metode su razvijene za upotrebu od strane malih programerskih timova koji mogu da rade u istoj prostoriji i da neformalno komuniciraju. Zbog toga se najčešće koriste za projektima razvoja malih i srednjih sistema. Kako i kod velikih projekata postoji pritisak da se sistem što pre isporuče kupcu, postavilo se pitanje da li, i kako, agilne metode mogu da se primene i u projektima razvoja velikih sistema. Specifičnosti razvoja velikih softverskih sistema:

1. Veliki sistemi su najčešće skupovi posebnih sistema koji međusobno komuniciraju, a koje razvijaju različiti timovi. Ti timovi često rade na različitim, pa i vrlo udaljenim lokacijama. Ti timovi nemaju vidljivost celog sistema. Njihov prioritet, zbog toga, je da završe svoj deo sistema ne obazirajući se na ostale delove.

2. Veliki sistemi komuniciraju i uključuju mnoge druge postojeće sisteme. Mnogi od njih služe za obezbeđivanje integracije i integracije ovih sistema, i ne obezbeđuju direktno neku posebnu funkcionalnost celog sistema, niti obezbeđuju fleksibilnost i inkrementalni razvoj.

3. Kada su više sistema integrisani u novi sistem, značajan deo razvoja se bavi konfiguracijom sistema, a ne razvojem originalnog koda.To nije kompatibilno sa inkrementalnim razvojem i čestim integracijama sistema.

4. Veliki sistemi i njihovi procesi razvoja su često ograničeni spoljnim pravilima i regulacijom koji ograničavaju način njihovog razvoja, jer zahtevaju proizvodnju određenih dokumenata.

5. Veliki sistemi imaju dugo vreme naručivanja i razvoja. Teško je održati isti razvojni tim na okupu celo vreme trajanja razvoja, te postoji nepoznavanje celine sisteme od strane novih članova.

6. Veliki sistemi često imaju različite zainteresovane aktere (stejkholderse). Zato je teško sve njih uključiti u razvojni proces softvera.

Postoji mišljenja da se agilne metode mogu koristiti i kod razvoja velikih sistema,ali uz izvesne modifikacije metoda. Postoje dve perspektive primene agilnih metoda u razvoju velikih sistema:

1. Perspektiva „scaling up“ upotrebe agilnih metoda za razvoj velikih sistema koji se ne mogu razvijati od strane malih timova.

2. Perspektiva „scaling out“ koja se bavi kako agilne metode se mogu koristiti u velikoj organizaciji sa višegodišnjim iskustvom u razvoju softvera.

Sommerville (2009) je mišljenja da se agilni metoda mogu prilagoditi za primenu u razvoju velikih sistema na sledeći način:

1. Kod velikih sistem se ne može tim da usmeri samo na razvoj koda. Mora da se pripremi najpre projektna i sistemska dokumentacija. Mora de projektovati arhitektura sistema i mora da postoji dokumentacija koja opisuje kritičke aspekte sistema, kao što je šema baze podataka, podela poslova članovima tima i dr.

2. Mora se uspostaviti mehanizam komunikacije unutar razvojnog tima . To može da obuhvati regularne telefonske i video konferencije između članova tima, kao i česte, kratke elektronske sastanke na kojima timovi upoznaju jedne druge sa napretkom svog posla. U tu svrhu koriste se komunikacioni kanali kao što su: e-pošta, razmena poruka, wikis, i socijalne mreže.

3. Stalna integracija, pri čemu se ceo sistem konfiguriše uvek kada se unosi neka promena, je nemoguća kada se mora se mora integrisati nekoliko sistema koji čine sistem. Međutim, neophodno je održavati čest razvoj sistema i proizvoditi nove verzije sistema. Zbog ovoga je potrebno koristiti nove alate za upravljanje konfiguracijom sistema a koji podržavaju rad više timova.

* 1. Obično se navodi da je jedan od problema uključivanje korisnika u razvojni tim je što mogu da izgube kontakt sa potrebama svojih kolega, korisnika sistema. Preporučite tri načina za izbegavanje ovog problema i diskutujte prednosti i nedostatke svakog od pristupa.

Ukjlučivanje korisnika sistema i česte promene u sastavu razvojnog tima – su dva najveća problema u primeni agilnih metoda u održavanju sistema.

Pitanje je logično, jer se razvoj odvija sa minimumom formalne dokumentacije, a ona je osnov za ljude koji dobijaju zadatak da promene nešto u sistemu, a nisu na njemu radili. Međutim, poznato je da u praksi, i kada se koristi formalna dokumentacija, ona najčešće nije ažurna, tj. osvežena promenama koje su u međuvremenu vršene. Promoteri agilnih metoda zato tvrde da nije važna formalna dokumentacija i da na njoj ne treba gubiti vreme, već je važno napraviti visoko kvalitetan, dobro strukturisan kod, koji je čitljiv i jednostavan za održavanje. Oni smatraju da se napor treba usmeriti ka dobijanju takvog koda, a ne ka dokumentaciji. Smatraju da nedostatak dokumentacije, ne bi trebalo da bude problem u održavanju koda, ako je on dobro napravljen. Međutim, ipak se smatra da bez dobrog dokumenta koji definiše zahteve, teško je oceniti posledice predloženih promena u sistemu. Zato, primena agilnih metoda u održavanju softvera ipak otežava njegovo kasnije održavanje i čini ga skupljim.

Praksa je ipak pokazala, da se agilni metodi mogu uspešno primeniti i u održavanju, jer inkrementalne isporuke popravki postojećeg sistema je prirodan način evolucije softvera. Glavni je problem uključiti korisnika sistema u rad na popravkama sistema, i u fazi evolucije softvera.

Drugi problem kod primene agilnih metoda u održavanju je u promenama razvojnog tima. Ako tih ljudi više nema, sa njima je otišlo i njihovo iskustveno znanje i poznavanje sistema, a novi ljudi, u odsustvu dokumentacije, imaju problem da se snađu.

Ima predloga da se koriste hibridne metoda, koje spajaju najbolje strane agilnih metoda i planom-vođenih metoda razvoja softvera

1. **Lekcija 15: Upravljanje projektovanja razvojem softvera**
   1. Koja su tri glavna ograničenja pri razvoju softvera? Dajte neki primer.
   2. Šta je cilj menadžera projekta? Navedite njogove odgovornosti.

Softver treba razviti u okviru određenog budžeta i u definisanom roku, a pri tom, obezbediti njegov kvalitet, tj. da ispunjava sva očekivanja kupca.

Upravljanje projektom razvoja softvera je bitan deo softverskog inženjerstva. Softver treba razviti u okviru određenog ubudžeta i u definisanom roku. Menadžer projekta treba da obezbedi da se sa takvim ograničenima razvije kvalitetan softver. Dobo upravljanje nije dovoljno za garanciju uspeha, ali loše uprvljanje projektom sigurno dovodi do neuspeha, tj. do zakašnjenja u isporuci, probijanja budžeta, ili do isporuke softvera koji ne zadovoljava zahteve korisnika. Menadžer projekta najčešće ima definisane sledeće ciljeve:

1. Isporuka softvera korisniku u dogovorenom roku.

2. Troškovi projekta da budu u okviru planiranog budžeta.

3. Isporuka softvera koji zadovoljavaju očekivanje kupaca.

4. Održavanje zadovoljstva i dobro funkcionisanje razvojnog tima.

Ovo su ciljevi koji su slični za sve projekte, i van softverskog inženjerstva, i u drugim disciplinama. Međutim, upravljanje razvojem softvera nosi i svoje specifičnosti koje se ogledaju u sledećem:

1. Proizvod je nedodirljiv. U drigim tehničkim disciplinama, menadžer projekta vidi kako napreduje razvoj proizvoda. U slučaju softvera to nije slučaj. On zavisi od drugih koji mora da mu daju potrebne informacije da bi mogao da prati stanje razvoja softvera.

2. Veliki projekti razvoja softvera nisu “jedan od” projekata. Šta to znači? Veliki projekti se najčešće razlikuju, te prethodno iskustvo nije uvek relevantno. Dolazi do napretka i promene tehnologija, te to zahteva i od iskusnih menadžera stalno usavršavanje i učenje.

3. Softverski procesi su promenljivi i organizaciono specifični. Procesi razvoja softvera se dosta razlikuju od organizacije do organizacije. Iako je sve veža standardizacija ovih procesa, ipak je teško predvideti da li on vodi sigurno do uspeha. Softverski procesi dosta zavise i od načina realizacije širek projekta čiji su oni samo jedan deo (na primer, razvoj hardverskog proizvoda kojim upravlja softver).

Zbog ovih teškoća, značajan broj projekata razvoja softvera nisu uspešni, te i ako se završe, dovode do kažnjenja u isporuci, probijanju budžeta, a i manjkovastima u kvalitetu softvera.

* 1. Koje kategorije rizika postoje pri razvoju softvera? Navedite primere.

Potrebno je da menadžer projekta predvidi rizike koji mogu da utiču na njegov projekat, tj. na rokove, troškove i kvalitet softvera. Tri kategorije rizika: rizici projekta, proizvoda i poslovanja. Upravljanje rizikom je jedan od najvažnijih poslova menadžera projekta. Potrebnoj e da predvidi rizike koji mogu da utiču na njegov projekat, tj. na rokove, troškove i kvalitet softvera. Postoje tri kategorije rizika:

1. Rizici projekta: To su rizici koji utiču na plan projekta i na njegove resurse. Na primer, odlazak projektanta softvera.

2. Rizici proizvoda: To su rizici koji utiču na kvalitet softverskog proizvoda. Na primer, nabavka komponente koja utiče na kvalitet softvera.

3. Poslovni rizici: To su rizici koji utiču na organizaciju razvoja i prodaju softvera. Na primer, konkurent se pojavljuje na tržištu sa sličnim proizvodom.

Efektivno upravljanje rizicima olakšava rad sa problemima i obezbeđuje da oni ne dovedu do probijanja budžeta, rokova ili ugrožavanj akvaliteta softvera. Tabela 1 prikazuje neke od čestih i zajedničkih rizika koje prate sve projekta razvoja softvera.

* 1. Nacrtajte proces upravljanja rizikom pri razvoju softvera.

Rizike treba identifikovati, analizirati, ali i planirati delovanje u slučaju njihovog javljanja, kao i stalno pratiti i ocenjivati rizike. Na slici 1 prikazan je proces upravljanja rizicima. On obuhvata sledeće aktivnosti:

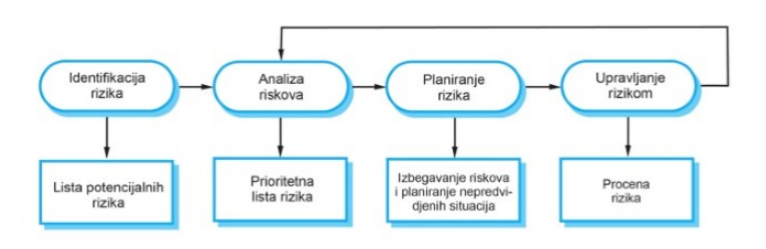
1. Identifikaciji rizika: Trebalo bi da utvrdite moguće rizike projekta, proizvoda i poslovanja.

2. Analiza rizika: Trebalo bi da procenite verovatnoću javljanja rizika i posledice njihovog javljanja.

3. Planiranje rizika: Morate da pripreminte planove za delovanje u slučaju dejstva rizika, d bi ili izbegli javljanje rizika ili smanjili negativne efekte njihovog javljanja.

4. Monitoring rizika: Trebalo bi regularno da ocenjujete rizike i važe planove u slučaju njihovog dejstva, kao i njihovu promenu kada steknete više znanja o njima.

Kao izlaz iz procesa upravljanja rizicima, trebalo bi da pripremite dokument u kome bi izložili vaš plan upravljanja rizicima. On bi trebalo da sadrži informaciju o mogućim rizicima, analizu identifikovanih rizika, kao i informaciju kako vi mislite da delujete u slučaju njihovog javljanja



* 1. Navedite vrste rizika i navedite po jedan primer za svaki od njih.

1. Rizici tehnologije: Rizici koji nastaju iz tehnologije hardvera i softvera koja se koristi pri razvoju softverskog sistema.

2. Rizici ljudskog faktora: Rizici povezani sa ljudima koji su angažovona i razvoju softvera.

3. Organizacioni rizici: Rizici koji potiči iz organizacionog okruženja u kome se softver razvija.

4. Rizici alata: Rizici koji potiču od razvojnih alata koji se koriste u projektu.

5. Rizici zahteva: Rizici koji su vezani za promene zahteva kupaca i procesa upravljanja promena zahteva.

6. Rizici procenjivanja: Rizici koji nastaju iz upravljanja resursima neophodnih za razvoj sistema.

* 1. Kako se mogu oceniti verovatnoža rizika. Kako se mogu oceniti efekti rizika.

Pri analizi rizika, potrebno je oceniti verovatnoću javljanja i mogući efekat na projekat svakog identifikovanog rizika. Ovo dosta zavisi o vaše procene i prethodnog iskustva. Pri tome se ocenjuje rizik koristeći se sledećim kategorizacijama:

1. Verovatnoća rizika je veoma niska (75%).

2. Efekti rizika se mogu oceniti kao katastrofični (ugrožavaju opstanak projekta), ozbiljni (dovešće do značajnih kašnjenja), tolerantna (zakašnjenja su u okviru dozvoljenih odstupanja) i beznačajni.

* 1. Šta je planiranje rizika. Koje se strategija u planiranju rizika najčešće koriste? Dajte primere.

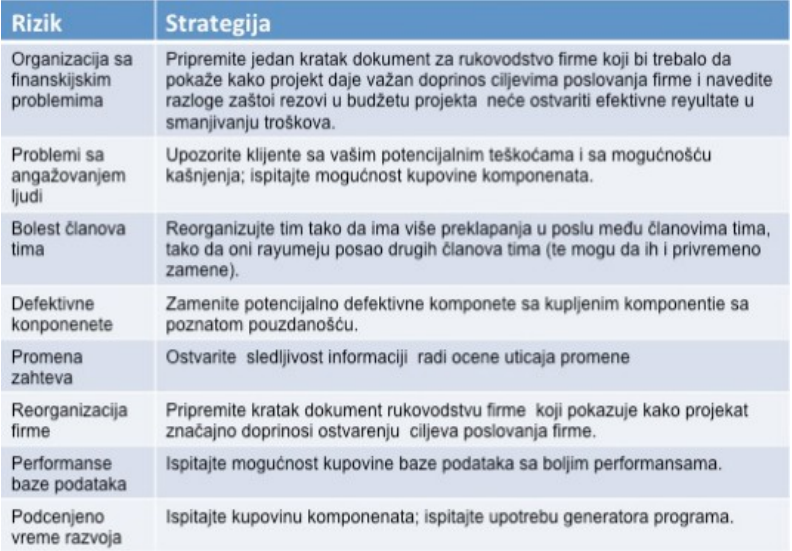
Proces planiranja rizika razmatra svaki od identifikovanih rizika i razvija strategije za upravljanje ovim rizicima. To znači da se pripreme akcije koje bi se spovele ako se neki od tih rizika obistine, a sa ciljem da se njihov remetilački efekat po projekt minimizira. Takođe, mora da se utvrdi informacija koja je potrebna u toku monitoringa projekta, kako bi se na vreme uočio problem. Tabela 5 prikazue moguće strategije koje se obično primenjuju kod ključnih rizika, tj. onih koji su ozbiljni i neprihvatljivi, a prikazani na slici 4. Te strategije se razvrstavaju u tri kategorije:

1. Strategije izbegavanja: Ona rade na smanjivanju verovatnoće javljanja rizika. Na primer, strategija za defektne komponente.

2. Strategije minimizacije: One rade na smanjivanju efekta rizika na projekat. Na primer, startegija za bolest članova tima.

3. Planovi za slučaj javljanja nepredviđenih situacija (contingency plans): Ovde imate jasan plan šta raditi ako dođe do najgoreg scenarija. Na primer, strategije delovanja u slualju organiyacija sa finansijskim problemima

Najpre bi trebalo da pokušate da primenite strategije za smanjivanje verovatnoće javljanja rizika, pa onda strategije za minimizaciju negativnih efekata ako dođe do javljanja rizika, i konačno, tek na kraju, primenite strategije sa planovima akcija u slučajvime javljanja nepredviđenih situacija. Ovo ima za cilj da se smanji uticaj rizika na projekat ili proizvod.

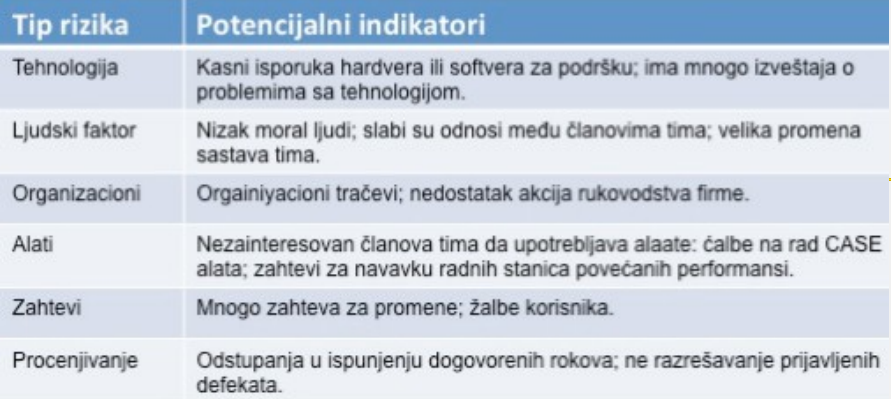


* 1. Šta je monitoring rizika. Koji su indikatori razika. Navedite ih po tipovima rizika.

Monitoring rizika je proces provere da li se vaše predpostavke vezane za proizvod, proces i poslovne rizike nisu promenile. Regularno treba da proveravate svaki od utvrđenih rizika, kako bi procenili da li se menja verovatnoća njegoog javljanja. Takođe, treba da ponovo procenite da li se menjaju njihovi potencijalni efekti na realizaciju projekta.

Trebalo bi da uzmete u razmatranje i druge faktore, kao što je broj predloga za promenu zahteva koji daje ideju o verovatnoći rizika i o efektima. Na slici 6 su navedeni neki faktori koji mogu da doprinesu boljem sagladavanju verovatnoće javljanja pojedinih tipova rizika i njihovih efekata.

U svim fazama projekta trebalo bi redovno da pratite rizike. Na svakom sastanku, na kome se analizira stanje na projektu, trebalo bi da prodiskutujete svaki od ključnih rizika da bi ponovo procenili da li se promenila verovatnoća njegovog javljanja i da li je došlo do promene njegovih mogućih efekata.



* 1. Koji su kritični faktori u upravljanu ljudima, tj. članovima razvojnog tima projekta? Objasnite svaki o kritičnih faktora.

Najveću vrednost u organizacijama koje razvijaju softver su ljudi koji u njoj rade. Važno je da menadžeri projekata razumeju tehnička pitanja koja utiču na rad na razvoju softvera. Međutim, to nije dovoljno. Vrlo je važno da menadžeri projekta imaju dara za rad sa ljudima. Dobar softver inženjer često nema tzv. neke veštine koje mu omogućavaju da motiviše ljude i da vodi projekat. U radu sa ljudima, postoje čeitiri kritička faktora:

1. Konsistentnost. Svi ljudi u projektnom timu bi trebalo da se tretiraju na isti način.

2. Respekt. Menadžeri treba da uvažavaju razlike u sposobnostima ljudi. Svakom treba dati šansu da se pokaže i da prikaže šta zna.

3. Inkluzija: Ljudi efektivno doprinose kada vide da se njihova reč sluša i kada se njihovi predlozi uzimaju u obzir.

4. Poštenje. Kao menadžer projekta uvek bi trebalo da pošteno ocenite šta ide dobro, a šta loše u projektnom timu. Pošteno je i priznati nekom da ima više znanja od vas. Ako pokušavate da ignorišete ili prikrijete probleme, izgubićete respekt grupe.

* 1. Koje potrebe ljudi utiču na njihovu motivaciju?

Motivacija znači organizovanje rada i radnog okruženja koji podstiče ljude da daju svoj najveći mogući doprinos.

Kao menadžer projekta, trebalo bi da motivišete ljude da rade sa vama tako da doprinose najbolje što mogu. Motivacija znači organizovanje rada i radnog okruženja koji podstiče ljude da daju svoj najveći mogući doprinos. Jedan od načina da se ljudi motivišu je i da se zadovolje njihove potrebe. Na slici 7 su prikazani nivoi potreba. Na nižim hijerarhijskim nivoima su osnovne potrebe ljudi, Te potrebe ljudi prvo žele da zadovolje. Međutim, u softverskim organizacijama, ljudi imaju potrebe na višim nivoima, te njih treba zadovoljiti.

1. Socijalne potrebe: I kada ljudi rade na udaljenim lokacijama zajedno u timu, važno je da se na početku projekta okupe zajedno i lično upoznaju i uspostave personalni kontakt. Direktna interakcija je vrlo bitna za prihvatanje ljudi da postaju deo jedne zajedničke društvene grupe.

2. Poštovanje: Da bi stekli poštovanje, morate pokazati ljudima da ih organizacija uvažava i vrednuje. Davanje javnih priznanja je jednostavan, ali efektivan način za to. Naravno, ljudi treba da osećaju da su i adekvatno plaćeni u skladu sa svojim doprinosom organizaciji.

3. Odgovornost: Dajte ljudima odgovarajuću odgovornost i obuku da dostignu neophodne veštine.

* 1. Zašto je za razvoj projekta potrebno primeniti timski rad? Koje su koristi od kohezije tima? Koja tri faktora utiču na rad tima?

Profesionalni softver najčešće razvijaju timovi čiji broj članova se kreće od dva pa do nekoliko stotina. Velike grupe se dele na manje. Svaka od grupa je odgovorna za razvoj dela softverskog sistema. Opšte je pravilo da jedna grupa ne bi trebalo da ima više od 10 članova. Kada je grupa mala (do 10 članova), smanjeni su komunikacijoni problemi članova grupe, oni se dobro znaju, mogu da sednu za isti sto i da diskutuje sve probleme i dr.

Pri sastavljanju grupe, treba voditi računa da njihova znanja i veštine, koje su različite, ostvare potrebnu ravnotežu tehičkih veština, iskustva i ličnih svojstava. Pored toga uspešna grupa treba da bude kohezivna (“svi za jednog, jedn za sve”) i da u njoj vlada timski duh. Članovi grupe su motivisani uspehom grupe, kao da je to njihov lični uspeh.

U kohezivnoj grupi, njeni članovi smatraju da je grupa važnija od pojedinaca u njoj. Oni su svi lojalni grupi. Svi se identifikuju sa grupom i sa drugim članovima grupe. Kada zatreba, svi štite grupu od spoljnih “napadača”. To čini grupu robusnom i sposobnom da se suoči sa neočekivanim situacijama. Postižu se sledeće koristi od kreiranje kohezivne grupe:

1. Grupa može da postavi sopstvene standarde kvaliteta. Kako su oni usvojeni jednoglasno, veća je verovatnoća da će biti sprovedeni nego nametnuti standardi spolja.

2. Pojedinci uče i dobijaju pomoć od drugih članova: Članovi grupe uče jedan od drugog i pomažu jedan drugoga.

3. Znanje se međusobno deli: To osigurava kontinualnost rada grupe i kada ih neki član napusti, jer neki drugi član grupe može preuzeti njegov zadatak.

4. Podstiče se refaktorisanje i stalno usavršavanje softvera. Članovi grupe kolektivno rade na postizanju što boljih rezultata i rešavanju problema, nezavisno od toga koji od njih jradi na tom problemu.

Dobar menadžer projekta uvek podstiče stvaranje kohezije u grupi. S tim ciljem može organizovati i socijalna okupljanja van posla, uključujući i članove njihovih porodica. Stvaranju kohezivne grupe pomaže inkluzija, tj. da se svi članovi uvažavaju i da ima se poverava odgovornost, a i čine dostupnim sve informacije.

Sastav grupe, njena organizacija i interna i eksterna komunikacija grupe – to su faktori koji utiču na uspešan rad grupe. Tri faktora utiču na timski rad grupe:

1. Ljudi u grupi. Morate da izmešate ljude tako da se pokriju različite potrebe koje grupa ima (pregovaranje, programiranje, testiranje, dokumentovanje i dr.).

2. Organizacija grupe Grupa treba da bude tako organizovana tako da svaki njen član može da pruži maksimalan doprinos i da se svi zadaci izvrše prema očekivanjima.

3. Tehnička i menadžerska komunikacija. Bitna je dobra komunikacija i unutar grupe, a i sa spoljnjim okruženjem, kao što je komunikacija sa zainteresovanim licima.

Kreiranje pravog tima, kao i mnoge druge stvari, ne garantuje uspeh projekta. Postoje puno drugih faktora koji utiču na uspeh projekta, kap što su promene u poslovnom okruženju. Međutim, ako ne obratite pažnju pri formiranju kohezivne grupe sa potrebnim timskih duhom, veća je verovatnoća da će vaš projekat imati problema

* 1. Navedite pitanja čiji odgovori određuju organizaciju grupe.

Organizacija grupe utiče na način donošenja odluka, na informisanost članova grupe, i na interakciju grupe sa spoljnim okruženjem. Pri organizovanju grupe, javljaju se sledeća važna organizaciona pitanja na koje treba naći odgovarajuće odgovore:

1. Da li menadžer projekta treba da bude i tehnički lider grupe? Tehnički lider ili arhitekt sistema je odgovoran za kritičke tehničke odluke u toku razvoja softvera. Međutim, u slučaju velikih projekata, najbolje je odrediti iskusnog inženjera da bude arhitekta sistema u projektu koji bi preuzeo odgovornost za tehničko vođenje projekta.

2. Ko bi trebalo da bude uključen u donošenje važnih tehničkih odluka i kako će se one donositi? Da li će odluku doneti arhitekta sistema, menadžer projekta ili će se odluka doneti konsenzusom većeg dela tima?

3. Kako će se odvijati komunikacija grupe se rukovodstvom firme i spoljnim okruženjem? Najčešće, za to je zadužen menadžer projekta, potpomognut arhitektom sistema, ako ga ima. Međutim, postoji i mogućnost imenovanja posebnog člana grupe za vezu sa spoljnim okruženjem, imajući u vidu njegove dobre komunikacijone veštine.

4. Kako integrisati grupu sa članovima koji rade na različitim i udaljenim lokacijama? O ovome treba posebno voditi računa kod grupnog donošenja odluka.

5. Kako ostvariti deljenje znanja među članovima grupe? Način organizacije grupe utiče na deljenje informacija. Pri tome treba voditi računa i da se ne pretera, jer isuviše visok nivo deljenja znanja može da ometa rad članova grupe.

Neformalno organizovane grupe, koje donose odluke dogovorima, mogu biti efekanse ako je čine iskusni članovi. Međutim, kod heterogenih grupa, u kojim i dominiraju neiskusni članovi, to bi dovelo do nedostatka dobre koordinacije rada. Zato, u takvim slučajevime, menadžer grupe treba da preuzme odgovornost za donošenje odluka i koordinaciju zadataka članova grupe. Taj pristup se koristi i kod hijerarhijski organizovanih grupa

* 1. Koja su tri nivoa planiranja projekt? Koj tri glvna prametra utiču na troškobe projekt razvija.

Planiranje projekta je jedan od najvažnijih poslova menadžera projekta. Razvojne poslove treba da razbijete na manje celine, tj. na zadatke koje treba da podelite članovima razvojnog tima, da pratite njihov rad, i da rešavate probleme koji se mogu javiti. Planiranje projekta se vrši tokom:

1. pripreme projekta, kada se priprema predlog projekta,

2. u početnoj fazi projekta, kada treba da planirate ko treba da učestvuje u projektu, kako će projekat da se razbije na inkremente, kako će se rasporediti resursi i dr.

3. u toku realizacije projekta, kada se periodično vrši modifikacija plana u skladu sa stečenim iskustvom i na osnovu prikupljenih informacija o stanju projekta.

Kada se priprema predlog projekta, vaša firma mora da ponudi cenu softvera koji treba da razvije. Najpre je potrebno izračunati očekivane troškove, a na bazi početnog plana projekta koji ne može da bude vrlo precizan, jer se zasniva na nedovoljno detaljnim informacijama o samom softveru i o poslu koji treba da se radi. Najteže je proceniti potrebni obim rada ljudi. Pored toga, treba da dodate opšte troškove koja firma ima (zakup prostora,administracija, grejanje i dr.). Direktne troškove planiranog projekta određujete uzimanjem u obzir:

• troškove rada (bruto plate softverskih inženjera i menadžera),

• troškovi neophodnog hardvera i softvera, uključujući i njihovo održavanje.

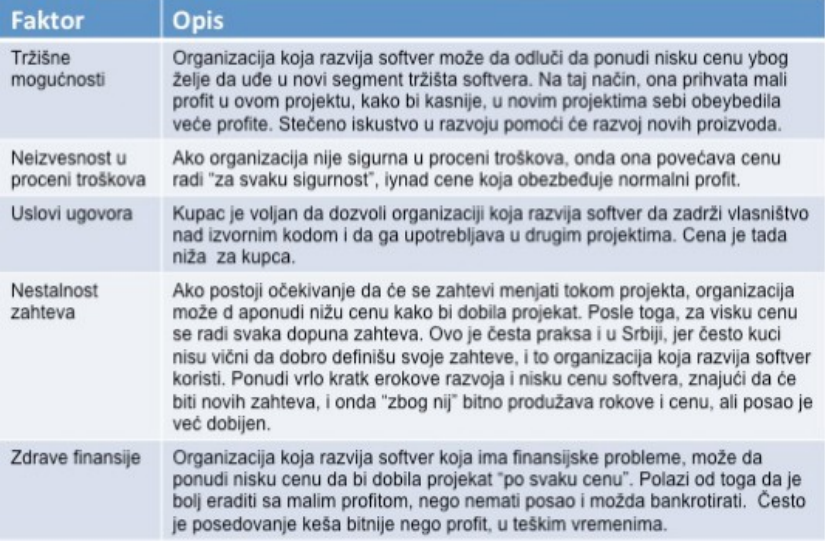
• troškovi putovanja i obuke. Najveći udeo u troškovima nose bruto plate članova razvojnog tima (inženjeri softvera, menadžeri). Zato, treba napraviti što realniju procenu očekivanog posla koji se izražava u jedinicama čovek-meseci.

* 1. Koji su uticajnu faktorna cenu softvera?

Pri pripremi cene za ponudu, treba uzeti u obzir i šire organizacione, ekonomske, političke, i poslovne faktore. Često se cena ponude podešava kako bi se posao dobio.

U osnovi, cena softvera je zbir troškova razvoja softvera i profita. Međutim, odnos troškova projekta i cene softvera koja se daje u ponudi, nije tako jednostavan. Pri pripremi cene za ponudu, treba uzeti u obzir i šire organizacione, ekonomske, političke, i poslovne faktore., kao što je pokazano u Tabeli 1 . Treba da uzmete u obzir interese organizacije, rizik posla, i tip ugovora koji treba da se sklopi. Zato, u donošenju odluke o ceni učestvuju ljudi iz marketinga, prodaje, rukovodstvo firme i menadžeri projekta.

Često se cena ponude podešava kako bi se posao dobio, pri čemu se nudi cena za koju se očekuje da je kupac spreman da je plati. Po dobjanu posla, počinju detaljniji pregovori sa kupcem, kako bi se dobile detaljnije informacije o poslu, i na osnovu toga, sačinio tačniji i detaljni plan projekta. Imajući u vidu ugovrenu cenu projekta, vodi se računa o tom ograničenju prilikom izrade plana realizacije projekta. Vrši se usaglašavanje zahteva tako da se ne probije ugovorena cena softvera.



* 1. Navedite tipičan plan projekta razvoja softvera? Nacrtajte dijagrm aktivnosti procesa planiranja projekta razvoja softvera.

Plan projekta pri planski vođenom projektu definiše potrebne resurse, raspodelu zadataka, i termin plan realizacije. Trebalo bi da identifikuje rizike projekta i softvera koji se razvija, i postupak koji će se primeniti u upravljanju projektom. Plan projekta najčešće sadrži sledeće:

1. Uvod: Kratak opis ciljeva projekta i postavljenih ograničenja (npr. budžet, vreme, i dr.).

2. Organizacija projekta: Orgnizacija tima i uloge (odgovornosti) članova tima.

3. Analiza rizika: Opisuje moguće rizike, verovatnoću njhovog javljanja i strategije za njihov smanjivanje.

4. Hardver i softver: Definiše se potrebna hardverska i softverska oprema neophodna za realizaciju projekta.

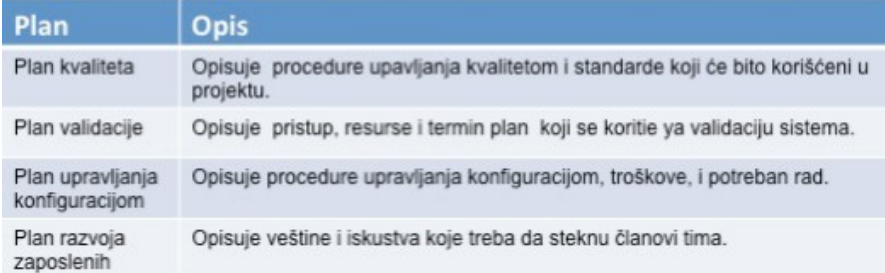
5. Definisanje zadataka (aktivnosti): Ukupan posao se deli na aktivnosti, i definšu se njihovi rezultati i rokovi

6. Termin plan projekta: Pokazuje zavisnost među aktivnostima, procenjeno vreme njihovog trajanja, vreme početka i kraja aktivnsoti, i određene ljude za njihovu realizaciju.

7. Monitoring i mehanizmi izveštavanja: Definiše izveštaje koji se moraju pripremati i vreme izveštavanja, kao i način monitoringa napretka projekt

* 1. Koja je dodatna dokumnetacij potrebna pri planiranju projekta?

Glavni plan projekta ima više priloga koji sadrže dokumenta koja bliže analiziraju pojedine aspekte plana projekta. Pored glavnog plana projekta, koristi se i izvestan broj drugih planova i dokumenata koji prate glavni plan projekta. Primeri ovih planova i dokumeata navedeni su u Tabeli 1.



* 1. Šta je termin plan projekta? Šta on definiše? Dajte dijagram izrade termin plana.

Planski vođen razvoj softvera je pristup u softverskom inženjerstvu u kome se proces razvoja detaljno planira. On definiže sve aktivnosti projekta, ko i kada treba da ih radi, kao i rezultate svake aktivnosti. Na osnovu takvog plana, menadžeri donose odluke i prate izvršenje projekta. Na ovaj “tradicionalan” način se realizuju najčešće veći projekti.

Promoteri planski vođenih projekta ističu da tako vođen projekat omogućava precizno predviđanje neophodnih resursa, tačnije određivanje rokova i troškova. Smatraju da se na taj način potencijalni problemi otkrivaju i pre početka projekta, umesto kasnije, u toku realizacije.

Oponenti planski vođenog razvoja, s druge strane, smatraju da se zbog stalnh promena zahteva i drugih faktora, plan mora često da menja, te se uzalud gubi vreme (i novac) za detaljnije planiranje projekta.

Planski vođen razvoj treba primeniti kod velikih i bezbednonosno-kritičkih sistema, sistema koji se rade po standardima te ih neko mora odobriti pre početka, kao i kod projekata u kojima učestvuje više organizacija. U slučaju malih i srednjih poslovnih aplikacija, koji treba da rade u vrlo promenljivim okruženjima, najčešće treba raditi metodim agilnog razvoja softvera

* 1. Koja dva osnovna tipa planiranja se koriste pri planiranju agilnom razvoja softvera. Objasnite ih.

Dva najčešće primenjivana metoda agilnog razvoja softvera su Strum i ekstremno programiranje.. Oni primenjuju dvofazno planiranje razvoja softvera:

1. Planiranje izdanja (release planning) gleda unapred nekoliko meseci i odlučuje o svojstvima koja treba da budu uključena u izdanje softvera na kraju tog periodu.

2. Iterativno planiranje (iteration planning) koje je usmereno na planiranje sledećeg inkrementa sistema. To definiše rad tima u naredne 2-4 nedelje.

* 1. Kako se procenjuje potreban rad na razvoju softvera?

Već pri pripremi ponude, sučeljavate se sa problemom procene potrebnog rada za razvoj softvera o kome ne znate mnogo, jer imate samo opšte zahteve kupca. Potrebno je da odredite potreban budžet projekta da bi dali ponudu, a kasnije ćete morati prema tom budžetu da ostvarujete moguće funkcije sistema. Koriste se dve tehnike procenjivanja potrebnog rada:

1. Tehnike zasnovane na iskustvu: Procena potrebnog rada se zasniva na iskustvu menadžera iz prethodnih projekata.

2. Algoritamsko modeliranje troškova: Ovde se primenjuju formule za obračun troškova, a na osnovu procenjenih atributa softvera, kao što su: veličina i karakteristike procesa, kao što je iskustvo članova tima.

U oba slčaja je potrebna vaša procena, ili za direktnu procenu potrebnog rada (efforts) ili indirektno, preko procene karakteristika softverskog proizvoda. U početnoj fazi projekta ove procene imaju veliku marginu greške. Tokom razvoja softvera, ove procene postaju tačnije, te se margina odstupanja smanjuje (slika 1 )

Procena na osnovu iskustva se vrši tako što se identifikuju svi moduli i komponente sistema, i onda se (na primer, u Excel-u) za svaki od njih procenjuje, na osnovu iskustva, vreme potrebno za njihov razvoj. Ako u tome učestvuje grupa ljudi, onda se mogu očekivati veća preciznost, tj. tačnija procena.

Problem sa ovakvom iskustvenom procenom je u tome što novi projekt može da se dosta razlikuje od prethodnih, jer se tehnologije i metode softverskog inženjerstva i alati brzo menjaju i razvijaju, te se i projekti razlikuju.

* 1. Navedite dijagram akcija procesa planiranja agilnog nacrtajte COCOMO modelII azvoja

Pri primeni algoritamskih modela, preporučljivo je raditi sa opsezima vrednosti parametara (sa najgorom, srednjom i najboljom vrednošću) umesto samo sa jednom vrednosti.

Procene postaju tačnije kada razumenete dobro tip softvera koji razvijate, imate kalibrisan model na bazi lokalnih podataka, ili kada su unapred definisani programski jezik i hardver.

Radi pomoći u proceni empirijskih modela, razvijeno je više modela. Ovde će se samo navesti ukratko COCOMO II model.

COCOMO II model podržava spiralni model razvoja softvera i koristi ugrađene podmodele koji proizvode detaljnije procene (slika 2 ). Na slici su prikazani modeli u okviru COCOMO II modela koji služe za procenjivanje vrednosti parametara modela. Za velike sisteme različiti delovi sistema se razvijaju različitim tehnologijama i vi ne možete sa istim stepenom tačnosti da procenjujete sve delove sistema.

