SE311 - Projektovanje i arhitektura softvera

1 Osnove projektovanja softvera

Q: Šta omogućava apstrakciju unutar projektovanja softvera?
A: Apstrakcija omogućava uklanjanje detalja iz opisa problema uz zadržavanje suštinskih osobina svoje strukture. Projektant softvera treba da razume i razmišlja o sistemu na apstraktan način kroz događaje, entitete, objekte ili neke druge predmete koji su bitni za sistem. Proces projektovanja retko može biti konvergentan u smislu da bude u stanju da projektanta usmeri ka jednom poželjnom rešenju. Efikasna upotreba apstrakcije predstavlja ključnu veštinu koju svaki projektant softvera treba da nauči i primeni. Apstrakcija se koristi u procesu rešavanju problema kao i za razdvajanje logičkih od fizičkih aspekata procesa projektovanja.

Q: Na koji način se vrši podela procesa projektovanja na projektne aktivnosti?
A: Na izgradnju i evaluaciju. Aktivnosti su usmerene na postizanje određenog definisanog efekta (cilja) i dešavaju se tako što se predpostavlja kranji rezultat a zatim se prelazi na definisanje načina za dolazak do predpostavljenog krajnjeg rezultata.

Q: Kako faza održavanja ima uticaj na proces projektovanja softvera?
A: Održavanje je neophodno isplanirati zato što se vremenom potrebe korsnika sistema menjaju kao i okolona u kojoj se softver koristi. Neophodno je u startu isplanirati i fazu održavanja.

Q: Kako testiranje softvera može ispuniti ciljeve verifikacije i validacije?
A: Testiranjem softvera omogućava nam da rano detektujemo neusaglašenosti u fazama razvoja procesa.

Q: Kako unaprediti proces testiranja softvera tako da odgovara ciljevima verifikacije i validacije procesa projektovanja softvera?
A: Možemo ga unaprediti tako što ćemo ga automatizovati.

Q: Šta se podrazumeva pod razmenom znanja projektanta sa drugim učesnicima u procesu projektovanja softvera?
A: Projektant softvera treba da poseduje tri značajne karakteristie:

1) Poznavanje domena aplikacije omogućiće jednostavno mapiranje strukture problema i strukture potencijalnog rešenja
2) Veštine u proenošenju tehničke vizije projektnog rešenja drugim učesnicima procesa projektovanja

3) Identifikovanje tehničkog napretka projekta

Projektan je neophodno da ima sposobnos da prenese znanje i tehničku viziju projektnog rešenja. Tokom predstavljanja vrši se identifikovanje osobina objekata u procesu projektovanja.

Q: Šta su kanali komunikacije projektanta softvera?

A: Projektant softvera kroz kanale komunikacije dobija sve potrebne informacije relevantne za proces projektovanja softvera. U procesu projektovanja softvera potrebno je da projektan softvera ima određeni stepn znanja o domenu (domain knowledge) kao početni vid informacija koji je potreban za obavljanje bilo kog konkretnog zadataka u procesu projektovanja. Projektant mora da zna ograničenja i specifikaciju zahteva.

Q: Koja je uloga ograničenja u procesu projektovanja softvera?

A: Ograničenja mogu biti nametnuta shodno određenom zadatku a neka ograničenja mogu biti prisutna shodno potrebi za usklađivanjem sa drugim softverskim proizvodima U procesu projektovanja ograničenja se teško identifikuju. Veliki broj ograničenja identifikuje se u specifikaciji zahteva za softver iako nije uvek obavezno i podrazumevano da se tamo nalaze. Uglavnom, ograničenja služe da ograniče ukupni prostor mogućeg rešenja projektantu softvera. Ograničenja su specifična za svaki novi identifikovani problem i potrebno ih je razmotriti u svakoj fazi projektovanja softvera. Praćenje ograničenja moguće je kroz redovne preglede i revizije faza procesa projektovanja.

Q: Šta je metoda prepoznavanja?

A: Metoda u okviru koje projektant softvera prepoznaje rešenje problema koje je bilo u samom identifikovanom problemu.

Q: Šta je dugoročni plan izmena? U kojim slučajevima se koristi?

A: Kao što je već navedeno, često se u toku korišćenja softvera javi potreba za određenim modifikacijama i unapređenjima softvera. Planiranje dugoročnih izmena u procesu projektovanja zavisi od nekoliko faktora. Jedan od faktora je budžet. Ukoliko održavanje nije unapred planirano često se dešava da programeri nemaju podsticaj za izvršavanje izmena u toku procesa održavanja. Drugi faktor je nemogućnost projektanata softvera da stvore plan razvoja i održavanja softvera u budućnosti. Takođe, neadekvatna projektna dokumentacija može onemogućiti modifikovanje softvera shodno početnim planovima. Na osnovu Lehmanovog istraživanja (Lehman i Ramil, 2002) softverski proizvodi koji automatizuju ljudske ili društvene aktivnosti moraju da se menjaju i dodatno unepređuju kako bi zadovoljili potrebe korisnika.

Q: Objasniti da li proces projektovanja usložnjava razvoj softvera ili ga čini jednostavnijim?

A:

Q: Šta je model rešenja?

A: Model rešenja predstavlja UML diagram na osnovu prethodno definisanih funkcionalnih i nefunkcionalnih zahteva. Model rešenja ne mora da biti konačan ali treba da sadrži sve zahteve koji su dobijeni od strane naručioca softvera.

Q: Kako može doći do neusaglašenosti u procesu projektovanja softvera?
A: Neusaglašenosti u procesu projektovanja moraju biti otkrivene u što ranijoj fazi kako bi projektant doneo odluku u načinu ispravke uočene greške. Uoačavanje grešaka u kasnijim fazama procesa projektovanja zahteva dodatnu analizu ispravljanja grešaka i kako ispravljanje neusaglašenosti utiče na druge delove softvera. Takođe, potrebno je planirati i kasnije održavanje softvera jer se vremenom potrebe korisnika sistema menjaju kao i okolina u kojoj se softver koristi (operativni sistemi računara, performanse hardvera). Tri tipa održavanja softvera su:

1) održavanje koje se bavi proširivanjem i unapređenjem operativnog softvera uz dodavanje novih funkcionalnosti (Perfective maintenance)
2) održavanje kojim se vrše potrebne promene nametnute van specifikacije zahteva (primer može biti izmene u zakonodavstvu određene države ili promena operativnog sistema) (Adaptive maintenance)
3) održavanje koje za cilj ima ispravku uočenih grešaka u toku operativnog rada sistema (Corrective maintenance)

2 Arhitektonske strukture, pogledi i stilovi

Q: Kada je potrebno ostvariti visoke performanse softvera, na koji način je to moguće izvršiti kroz primenu softverske arhitekture?
A: Visoke performanse je potrebno ostvariti kada je neophodno odogovoriti na zadatke korisnika. Sposobnost softverskih sistema da prozvedu tačan rezultat nije od pomoći korisniku ukoliko sistemu treba dugo vremena da prozvede rezultat. Da bi se ostvarile visoke performanse softvera potrebno je:

1) ispitati potencijalni paralelizam kroz dekompoziciju sistema i sinhronizaciju sistema
2) identifikovati potencijalna uska grla performansi

Q: Šta predstavlja cilj pisanja softverske dokumentacije?
A: Dokumentovanje softverske arhitekture omogućava projektantu softvera da iz različitih uglova gledanja izvrši analizu budućeg softvera. Dokumentacija softverske arhitekture ima tri obaveze gledajući iz ugla atributa kvaliteta. Prvo potrebno je naznačiti koji zahtevi atributa kvaliteta definišu projektovanje. Drugo, potrebno je označiti odabrano rešenje da se ispune zahtevi atributa kvaliteta. I kao poslednje, potrebno je označiti argument zašto predloženo rešenje ispunjava potrebne atribute kvaliteta. Cilj je obezbediti dovoljno informacija tako da arhitektura sistema može bili analizirana i da se iz navedene analize vidljive mogućnosti sistema da ispuni zahteve atributa kvaliteta.

Q: Na koji način se određuju korisnici dokumentacije softverske arhitekture?
A: Glavni korisnik dokumentacije arhitektura je projektant. Ukoliko projektant softvera bude ista osoba koja je pisala dokumentaciju softverske arhitekture ili na to mesto dođe novi projektant uvek će imati dokumentaciju koje može koristiti. Novi projektanti softvera koji su zainteresovani

za razumevanje softvera, analizu eventualnih nedostataka i donešenih odluka u određenom trenutku procesa projektovanja mogu to jednostavno izvršiti. Čak i ako se radi o dokumentovanju arhitekture u kratkim crtama i takav način omogućava i unapređuje proces projektovanja softverske arhitekture.

Q: Šta omogućavaju različiti pogledi na softversku arhitekturu?
A: Različiti pogledi ističu različite sistemske elemente ili veze između elemenata.

Q: Šta se postiže dokumentovanjem različitih pogleda na arhitekturu softvera?
A: Različiti pogledi ističu različite sistemske elemente ili veze između elemenata.

Q: Za koje aktivnosti u toku procesa razvoja softvera koristimo dokumentaciju softverske arhitekture?
A: Tu najčešće spadaju aktivnosti kao što su kodiranje, reinženjering ili testiranje u kojima će ušteda troškova biti značajna.

Q: Koji koraci su potrebni da bi se izvršilo kombinovanje različitih stilova softverske arhitekture?
A:

Q: Šta je element softverske arhitekture?
A: Svaki element softverske arhitekture predstavlja rezultat pažljivog i preciznog projektovanja u cilju ispunjavanja zahteve za kvalitet softvera a i zahteve nametnute od strane logike poslovanja organizacije za koju se softver projektuje. Softverska arhitektura omogućava da setovi delova sistema rade spojeno i uspešno u jednoj funkcionalnoj celini.

Q: Zašto je potrebno pisati softversku dokumentaciju iz ugla čitaoca?
A: Pravilo podrazumeva da projektant softvera u svakom trenutku razmišlja o krajnjem cilju dokumentacije softvera koji projektuje a to je da dokumentacija treba da bude od koristi određenim korisnicima. Često se zbog kratkih rokova za projektovanje softvera zaboravlja na ovo pravilo i dokumentacija piše vrlo brzo bez detaljnog objašnjenja sistema. Čitalac softverske dokumentacije koji primeti da je dokument pisan tako da njemu bude razumljiv, čitaće dokument svaki put kada mu bude potrebna pomoć u razumevanju softvera, dok ukoliko čitalac primeti da je dokument nerazumljiv neće ga koristiti naredni put. Neophodno je pisati softversku dokumentaciju iz ugla čitaoca zato što je nophodno da dokumentacija bude prilagođena za njega kako bi mogao da je razume. Dokumentacija je neophodno da bude koncizno i precizno tako da prosečan čitaoc razume celokupnu dokumentaciju. Dokumentacija napisana za čitaoca biće čitana dok dokumentacija napisana tako da bude pogodna za pisca neće.

Q: Kako primena pravila za pisanje softverske dokumentacije utiče na proces pisanja softverske dokumentacije?
A:

Q: Na koji način se vrši odabir elemenata koji se predstavljaju u softverskoj dokumentaciji?
A: Jedan od zadataka u dokumentovanju pogleda na arihtekturu softvera je odlučivanje koje osobine elemenata treba dokumentovati. Svojstva gotovo uvek uključuju ime elementa, kratak opis i ulogu ili odgovornost u arhitekturi softvera. Recimo element slojevitog stila (koji spada u stil modula) treba da ima ime sloja, jedinicu softvera koju sloj sadrži i sposobnost koju sloj pruža. Kroz prikaz sloja moguće je navesti ime za svaki sloj, jedinicu softvera i mogućnosti koje svaka jedinica omogućava. Pored ovih osnovnih osobina, postoje svojstva koja se koriste za analizu arhitekture. Ukoliko je potrebno analizirati performanse dokumentovane arhitekture onda svojstva treba da imaju vreme odgovora najboljeg i najlošijeg slučaja, maksimalan broj događaja koje određeni elemenat sistema može da obradi u jedinici vremena. Analiza bezbednosti arhitekture podrazumeva svojstva nivoa šifrovanja, pravila autorizacije za različite elemente i odnose. Često se može desiti da i kombinacija stilova korišćenih u određenoj softverskoj arhitekturi može omogućiti različit pogled. Svaki stil ima sopstveni rečnik (tipova elemenata i odnosa). Zajedno predstavljeni stilovi treba da budu dosledno opisani i spremni za upoređivanje i selekciju. Ukoliko su stilovi predstavljeni na identičan način sa definisanim prednostima i manama projektant softvera može izvršiti analizu stilova i pronaći pravi stil za projektovanje softverske arhitekture.

Q: Koje su prednosti pisanja dokumentacije u toku procesa razvoja softvera?
A: Prednosti su te što u toku procesa projektovanja donese se odluke i preispituju se sa velikom učestalošću. Revizija dokumenta koji ne predstavlja realnu sliku procesa projektovanja je nepotreban trošak.

Q: U kojim slučajevima se koristi formalna notacija?
A: U slučajevima kada su pogledi imaju preciznu (matematička zanosovana) semantika. Formalna analiza je u tom slučaju moguća. Opisane notacije obično pružaju grafički prikaz i osnovnu semantiku predstavljanja arhitekture. U nekim slučajevima dešava se da su formalne notacije specijalizovane za određene stilove. Određivanje vrste notacije za određenu upotrebu zahteva i nekoliko ustupaka u procesu projektovanja. Formalne notacije zahtevaju dosta više vremena i napora projektanta softvera ali doprinose smanjenju dvosmislenosti i većoj mogućnosti kasnije analize.

Q: Da li možete identifikovati neke nedostatke pisanja softverske dokumentacije?
A: Jedini nedostatak potencijalno može da bude ukupna cena izrade softvera ali cena izrade i održavanja dokumentacije arhitekture, na osnovu pretpostavki, trebao bi da bude pokriven izbegavanjem mogućih grešaka u procesu implementacije softvera. Tu najčešće spadaju aktivnosti kao što su kodiranje, reinženjering ili testiranje u kojima će ušteda troškova biti značajna.

3 Stilovi softverskih modula

Q: Kada se koristi izraz modul softvera?
A: Projektanti softvera koriste izraz modul za objašnjavanje softverske strukture uključujući jedinice programskog jezika kao što su: C programi, Java ili C# klase, PL/SQL procedure ili opšte grupacije programskog koda kao što su Java paketi. Modulima, kao što je već navedeno, moguće je dodeliti svojstva, definisati odgovornost u softveru i analizirati karakteristike. Moduli se mogu dodatno razložiti na nove module ili spojiti u jedan modul. Slojeviti stil identifikuje module i povezuje ih na osnovu relacije koja dozvoljava takav način korišćenja (allowed-to-use-relation) dok stil generalizacije identifikuje module na osnovu zajedničkih osobina.

Q: Šta omogućavaju različiti pogledi na softversku arhitekturu?
A: Dokumentacija softverske arhitekture bez bar jednog pogleda modula ne smatra se kompletnom dokumentacijom i kao takva je često neupotrebljiva. Razumevanje i primena stilova softverskih modula, pregled elemenata, relacija i svojstva pogleda modula. Kombinovanje pogleda modula sa drugim pogledima na softversku arhitekturu, notacije unutar pogleda modula.

Q: Koja je osnovna relacija stila razlaganja?
A: Osnovna relacija ovog stila je "je deo od" relacija koja prikazuje da element može biti deo drugog elementa. Razlaganje modula može omogućiti vidljivost podmogula samo u okviru modula u kome se nalaze rodidelj modula ili vidljivost svim ostalim modulima sistema.

Q: Na koji način je stil razlaganja pogodan za nove članove razvojnog tima?
A: Pregled stila razlaganja je pogodan za proces učenja o sistemu za nove članove razvojnog tima. Novi članovi se mogu fokusirati samo na određeni deo sistema od koga žele da krenu i zatim pratiti veze sa ostalim identifikovanim modulima sistema. Ne moraju znati kompletan sistem ili sve funkcionalnosti sistema da bi mogli da rade na implementaciji ili modifikaciji određenog modula. Takođe, iz ugla projektanta softvera pogodan je za proces projektovanja i navigacije kroz kompletan projekat. Projektant softvera može izvšiti podelu radnih zadataka članovima tima pomodulima dobijenim u stilu razlaganja.

Q: Po čemu se stil upotrebe razlikuje od stila razlaganja?
A: Stil razlaganja pokazuje samo strukturu implementacionih jedinica u smislu modula i podmodula, stil upotrebe ide korak dalje i otkriva koji modul može koristiti drugi modul unutar sistema. Kroz ovaj stil moguće je prikazati programerima koji vrše implementaciju softvera koji drug modul mora postojati u sistemu kako bi modul na kome rade pravilno funkcionisao. Stil upotrebe omogućava inkrementalni razvoj i postavljanje podskupova celih sistema.

Q: Šta podrazumevaju inkrementalni podskupovi modula?
A: Podrazumevaj da su pogodniji za analizu izmena u unutar sistema i postavljanje podskupova celih sistema.

Q: Koje su prednosti stila generalizacije na raspolaganju projektantu softvera?
A: Elementi stila generalizacije su moduli. Stil generalizacije može biti primenjen na različite tipove softverske arhitekture. Stil generalizacije koristi "je" relaciju. Kada su moduli predstavljeni kroz stil generalizacije to znači da "roditelj" modul predstavlja generalnu verziju modula "dete". Unapređenja modula mogu biti izvršena dodavanjem, brisanjem ili izmenom modula "deteta" dok se unapređenje roditelja automatski primenjuju na modul "dete". Generalizacija može biti korišćenja za:

1) Objektno-orijentisano projektovanje. Stil generalizacije omogućava izražavanje nasleđivanja u objektno-orijentisanom projektovanju sistema.
2) Produžavanje. Često je lakše razumeti modul koji je drugačiji od drugog poznatog modula nego razumeti nov modul od nule. Generalizacija predstavlja mehanizam za proizvodnju inkrementalnih opisa u cilju formiranja potpunog opisa modula.
3) Lokalna promena ili varijacija. Svrha arhitekture je da obezbedi stabilnu strukturu koja odgovara mogućim promenama ili varijacijama u procesu projektovanja softvera.
4) Ponovna upotreba. Odgovarajuće apstrakcije mogu se ponovo koristiti samo na nivou
5) interfejsa ili kroz proces implementacije. Definisanje apstraktnih modula omogućava njihovu ponovnu upotrebu.

Izražavanje generalizacije moguće je kroz UML. Moduli se prikazuju kao klase ili interfejsi. Na slici 2 prikazuje se osnovna notacija koja je dostupna u UML-u za klasu ili interfejs.

Q: Na koji način je omogućeno "produžavanje" modula primenom stila generalizacije?
A: Često je lakše razumeti modul koji je drugačiji od drugog poznatog modula nego razumeti nov modul od nule. Generalizacija predstavlja mehanizam za proizvodnju inkrementalnih opisa u cilju formiranja potpunog opisa modula.

Q: Kako se prikazuje softverska arhitektura kroz stil slojeva?
A: Slojeviti stil kao i svi stilovi modula vrši podelu softvera na jedinice. Jedinice u ovom stilu predstavljaju slojeve. Svaki sloj predstavlja grupu modula koji su povezani i omogućavaju povezani skup usluga. Odnosi između slojeva moraju biti jednosmerni. Slojeviti prikaz arhitekture softvera jedan je od često korišćenih prikazana u dokumentaciji. Korišćena relacija ovog sloja je dozvoljeno da koristi (" allowed to use"). Takođe, često se dešava da projektanti softvera pogrešno koriste prikaz softverske arhitekture prikazujuću slojeve sistema čak i ako oni nisu slojeviti. Softverska arhitektura se kroz slojeviti stil prikazuje uglavnom odozgo nadole. U softverskoj arhitekturi često može biti prisutno pravilo da moduli na veoma visokom nivou zahtevaju korišćenje modula na veoma niskom nivou. Dijagram slojevitog stila mora prikazati ovako projektovanu softversku arhitekturu. Slojevi predstavljaju logične grupe koje omogućavaju pomoć u kreiranju i komuniciranju arhitekture ali često nisu eksplicitno ograničene kroz programski kod. Izvorni kod može otkriti gde se koji modul koristi i šta određeni modul koristiti unutar sistema. Kriterijumi za definisanje slojeva sistema mogu biti očekivanja projektanta softvera da će određeni sloj biti nezavisno razvijan od drugih slojeva ili zaduženje određenog člana tima da razvija određeni sloj sistema. Takođe, podela slojeva u razvojnom timu može biti i

na osnovu specifičnog znanja nekog od članova razvojnog tima kome će biti dodeljen određeni sloj.

Slojevi omogućavaju postavljanje atributa kvaliteta za modifikaciju i prenosivost softverskog sistema. Elementi slojevitog stila su slojevi. Sloj predstavlja kolekciju modula a moduli mogu biti bilo šta kao na primer veb servisi ili deljivi podaci. U slojevitom stilu dozvoljena je komunikacija između slojeva relacijom "dozvoljeno da koristi". Za dva sloja koja koriste navedenu relaciju svakom modulu koji je u prvom sloju je dozvoljeno da koristi drugi modul drugog sloja. Slojevi mogu imati određena svojstva koja moraju biti dokumentovana kroz katalog elemenata unutar grafičkog prikaza:

1) Sadržaj. Opisuje sloj i sadrži uputstva kako moduli funkcionišu u sloju i na koji način se implementiraju. Svaki modul treba dodeliti tačno jednom sloju. Slojevi mogu imati oznake koje su opisne ili nejasne kao na primer "mrežni komunikacioni sloj" ili "sloj poslovnih pravila" pa je potreban dodatni opis koji identifikuje kompletan sadržaj svakog sloja.
2) Softveru je dozvoljeno korišćenje sloja. Ovaj deo dokumentacije objašnjava izuzetke, ukoliko ih ima kao i pravila korišćenja određenog sloja sistema.

Slojevi predstavljaju primenu principa prikrivanja informacija. Promena na donjem sloju softverske arhitekture može biti skrivena iza njegovor interfejsa i neće uticati na slojeve iznad u hijerarhiji. Promene u okviru određenog sloja mogu uticati na pretpostavku u prformansama softvera. Pogrešna pretpostavka je da slojevi uvode dodatne troškove izvršenja, jer u složenim sistemima često se dešava da se troškovi razvoja i implementacije softvera primenom slojevitog stila na softversku arhitekturu dodatno smanjuju.

Q: Koji stilovi softverske arhitekture su pogodni za korišćenje uz stil slojeva?
A:

Q: Šta su aspekti?
A: Aspekti su slični klasama, mogu sadržati atribute i operacije a mogu izvršiti i nasleđivanje. Aspekti omogućavaju nasleđivanje i stil aspekta može biti kombinovan sa stilom generalizacije ukoliko je potrebno prikazati hijerarhiju aspekata.

Q: Koje informacije sadrži prikaz aspekta u softverskoj dokumentaciji?
A: Za prikaz aspekata UML nema definisan simbol tako da se najčešće koriste klase unutar klasnog dijagrama. Prikaz unakrsnog koncepta može biti predstavljena kao stereotipna zavisnost koja ide sa aspekta na svaki ukršteni modul. Takođe, može biti dodat komentar modulu aspekta kako bi veza između aspekta i drugog modula bila opisana (kroz formalnu sintaksu ili na prirodnom jeziku).

Q: Kada se koristi model podataka?
A: Koristi se za opisivanje strukture podataka korišćenih u sistemu. Omogućuje analize uticaja izmena na model podataka. Sprovođenje kvaliteta podataka izbegavanje redudantnosti i nedoslednosti i vođenje implementacije modula koji pristupaju podacima.

Q: Koji su tipovi modela podataka?

A: Tipovi modela podataka mogu biti:

- Konceptualni. Konceptualni model podataka se fokusira na entitete i njihove veze.
- Logički. Logički model podataka predstavlja evoluciju konceptualnog modela podataka kroz tehnologiju upravljanja podacima (kao što su relacione baze podataka).
- Fizički. Fizički model podataka vrši implementaciju entiteta podataka. Omogućava optimizaciju, kreiranje identifikacionih ključeva i indeksa i vrši optimizaciju performansi.

Q: Da li primena stilova arhitekture usložnjava ili olakšava proces projektovanja softvera projektantu softvera?
A: Olakšava.

4 Stilovi povezivanja

Q: Na koji način se u neformalnoj notaciji predstavljaju komponente i konektori?
A: Pogled komponenti i konektora je sveprisutan u predstavljanju softverske arhitekture i uglavnom se sastoji iz kutije i linije koja prikazuje povezanost između kutija. Pristup modelovanju softverske arhitekture korišćenjem simbola kutije i linije predstavlja neformalni pogled i kao takav cesto može biti nejasan čitaocima softverske dokumentacije.

Q: Koja svojstva ima elemenat komponenta u pogledu komponente i konektora softverske arhitekture?
A: Komponente predstavljaju glavne računske elemente i skladišta podataka koji su prisutni u toku izvršavanja programa. Bitno je naglasiti da svaka komponenta ima ime na osnovu koje je moguće odrediti funkciju komponente. Takođe, ime može omogućiti jednostavnije dokumentovanje grafičkog prikaza. Komponente imaju interfejse koji se nazivaju portovi. Port predstavlja specifičnu tačku potencijalne interakcije komponente sa okruženjem. Najčešće ima eksplicitan tip koji definiše različite načine interakcije. Komponenta može imati više portova istog tipa.

Q: Kada se koristi stil toka podataka?
A: Primena ovih stilova je u okviru domena gde se obavlja obrada podataka kroz nekoliko transformacionih koraka. Stilovi toka podataka omogućavaju model u kome se komponente prikazuju kao transformatori podataka a gde konektori prenose podatke iz izlaza jedne komponente na ulaz druge komponente. Svaka vrsta komponente u stilu toka podataka ima određeni broj ulaznih i izlaznih portova. Posao komponente je da obrađuje podatke dobijene na ulazne portove i prosleđuje ih na izlazne portove. Najpoznatiji stil toka podataka je stil cevi i filtera.

Q: Koja je osnovna razlika između klijent-server stila i stila peer-to-peer?
A: Razlika između klijent server stila i ovog stila je da interakcija može biti inicirana od bilo koje komponente, jer svaka komponenta može delovati kao klijent ili server.

Q: Šta podrazumeva grafičko predstavljanje pogleda?
A: Predstavljanje dijagrama kroz softversku dokumentaciju podrazumeva tekstualni opis svih identifikovanih elemenata. Navedeni konektori predstavljaju složen oblik interakcije i zahtevaju kompleksne mehanizme u toku implementacije. Tip konektora klijent-server predstavlja protokol interakcije koji propisuje kako klijent započinju sesiju između klijenta i servera i opisuju način preusmeravanja ili prekidanja sesije. Implementacija klijent-server konektora uključuje mehanizme izvršavanja koji otkrivaju trenutni status servera, postavljaju zahteve klijenata, upravljaju dodavanjem i odvajanjem klijenata sa serverom. Konektori ne moraju biti binarni.

Q: Koja je svrha elementa konektor u pogledu komponente i konektora?
A: Konektori su druga vrsta elemenata u pogledu komponenta i konektor softverske arhitekture. Primeri konektora su: poziv za servisiranje, asinhrone poruke, tokovi podataka. Konektori često predstavljaju složene oblike interakcije kao što je recimo kanal komunikacije orijentisan ka transakcijama između servera baze podataka i klijenta ili magistrala servisnih usluga koja posreduje u interakcijama između usluga korisnika i provajdera. Konektori imaju uloge koji su njihovi interfejsi koji definišu načine na koje konektor može koristiti komponente za obavljanje interakcije. Primer može biti konektor klijent-server koji može imati ulogu poziva usluge (" invokes-services") i ulogu omogućavanja usluge (" provides-services"). Cev (u okviru" pipes and filter" stil softverske arhitekture) predstavlja uloge pisanja i čitanja. Uloge konektora se razlikuju od interfejsa modula po tome što se mogu replicirati ukazujući time koliko komponenti može biti uključeno u njihovu interakciju. Uloga obično definiše očekivanja od učesnika unutar interakcije.

Q: Koja su svojstva koja se koriste u pogledu komponente i konektora?
A: Svojstva komponente i konektora su različita. Svaki identifikovani element softverske arhitekture treba da ima ime i tip. Dodatna svojstva zavise od specifičnosti komponente ili konektora (projektant može dodati svojstva za detaljniji prikaz elementa). Identifikovana svojsta su potreba za vođenje implementacije softverske arhitekture i konfiguracije komponente i konektora. Ukoliko je potrebno izvršiti analizu performansi sistema na osnovu pogleda kompomente i konektora, projektant može dodati svojstva koja se tiču kapaciteta, latence ili prioriteta. Najčešća svojstva koja se mogu koristiti su:

1) Pouzdanost. Koristi se za utvrđivanje ukupne pouzdanosti sistema. Primer pitanja u okviru analize pouzdanosti može biti: "Koja je verovatnoća prestanka rada komponente ili konektora?"
2) Performanse. Koristi se za analizu sistemskih mogućnosti, testiranje potrebnog vremena za odgovor na određeni zadatak ili za utvrđivanje propusnog opsega određenog elementa softverske arhitekture.
3) Potrebni resursi. Koristi se za utvrđivanje potrebnog hardvera za određeni sistem. Primer analize potrebnih resursa može se bazirati na pitanju: " Koje su potrebe obrade i skladištenja podataka komponente ili konektora?".
4) Funkcionalnost. Koristi se za razumevanje kompletnih funkcija i mogućnosti unutar sistema. Primer analize funkcionalnosti može biti: "Koje funkcije obavlja element (bilo da se radi o komponenti ili konektoru)?"
5) Bezbednost. Koristi se za određivanje potencijalnih ranjivih tačaka sistema u okviru pogleda komponenta i konektor.

6) Konzistentnost. Koristi se u procesu analize ili simulacije performansi komponenti i za identifikovanje mogućih blokada u sistemu.
7) Nivo. Koristi se za definisanje procedure izgradnje sistema i raspoređivanje komponenti ili konektora u softverskoj arhitekturi.

Q: Kada se koristi stil komponente i konektora za predstavljanje softverske arhitekture?
A: Pogled komponente i konektora omogućava prikaz softverske arhitekture shodno definisanim pravilima stila.

Q: Koji pogledi na softversku arhteikturu mogu biti kombinovani sa pogledom komponente i konektora?
A:

Q: Koji stilovi softverske arhitekture spadaju u pogled komponente i konektora?
A: Spadaju sledeći stilevi:
- Stil poziv - povratak (Primeri stilova poziv-povratak su klijent-server, peer to peer i REST stilovi.)
- Stil peer - to - peer (Stil peer-to-peer omogućava da komponente direktno komuniciraju kao komponente na istom nivou uz razmenu usluga.)
- Stil zasnovan na događaju (Stilovi zasnovani na događajima omogućavaju komunikaciju između komponenata korišćenjem asinhronih poruka)

Q: Šta je pružalac usluga? Koja su njegova svojstva?
A: Komponente koje pružaju usluge su serveri. Serveri mogu pružati skup usluga preko jednog ili više portova. Neke komponente mogu delovati kao klijenti i serveri u softverskog arhitekturi. Takođe, može biti jedan ili više servera. Primeri klijent-server arhitekture su:
1) informacioni sistemi koji se pokreću na lokalnim mrežama gde su klijenti GUI aplikacije a sever sistem za upravljanje bazom podataka
2) veb aplikacije na kojima klijenti rade na veb pretraživačima a komponente se pokreću na veb serveru (primer može biti Tomcat)

Q: Koji su elementi sistema koji koristi klijent-server softversku arhitekturu?
A: Stil klijent-server predstavlja sistemski prikaz koji razdvaja klijentske aplikacije od usluga koje koriste. Stil podržava ponovnu upotrebu komponenti i zajedničkih usluga. Serverima može pristupiti bilo koji broj klijenata, relativno je lako dodati nove klijente u softversku arhitekturu. Takođe, serveri mogu biti replicirani da podrže skalabilnost ili dostupnost. Klijenti i serveri su često grupisani i raspoređeni na različitim mašinama u distribuiranom okruženju u svrhu formiranja višestruke hijerarhije, ukoliko je to potrebno.

5 Stilovi alokacije i hibridni stilovi

Q: Šta se dobija specijalizacijom stilova alokacije?
A: Specijalizacija stila alokacije se vrši u slučaju potrebe za ponovnim korišćenjem stila u različitim delovima određenog sistema. Drugi stilovi alokacije su takođe mogući, moguće je definisati zahteve za stilove alokacije koji povezuju sistemske zahteve sa softverskim elementima arhitekture. Primer za tako nešto može biti i specijalizacija određenog stila alokacije:

1) Specijalizacija stila raspoređivanja
2) Specijalizacija stila dodeljivanja radnih zadataka

Q: U kojim stilovima alokacije je potrebno prikazivati hardverske elemente okruženja softvera?
A: Stil raspoređivanja (deployment style) koji opisuje povezanost softverskih komponenti i konektora sa hardverom kompijuterske platforme na kojoj se softver izvravaša.

Q: Koja je osnovna relacija stila alokacije?
A: Relacija koja se koristi u stilu alokacije je dodeljen prema ("allocated to").

Q: Na koji način se vrši specijalizacija stila raspoređivanja?
A: Microsoft je razvio šablon nazvan "Tiered Distribution" koji propisuje raspodelu softverskih komponenti u višeslojnoj arhitekturi na hardverske elemente koji se koriste za pokretanje. Opisani šablon omogućava generički stil primene. Takođe, kompanija IBM ima svoju verzije šablona kao što je: topologija jedne mašine (stand- alone server), topologija vertikalnog skaliranja (vertical scaling topology), topologija horizontalnog skaliranja (horizontal scaling topology) itd.

Q: Zašto je neophodno kombinovati različite poglede na softversku arhitekturu?
A: Osnovni principi dokumentovanja softverske arhitekture predstavljaju grupu različitih pogleda koji omogućavaju detaljan prikaz softverske arhitekture. Često se dešava da tako odabrani pogledi nemaju zajedničke elemente ili relacije sa drugim pogledima pa tako čitaoci projektne dokumentacije nemaju uvid u ono što je projektant kroz poglede hteo da prikaže.
Prilikom razmatranja kombinovanog pogleda, potrebno je proveriti da li je asocijacija između elemenata jasna. U suprotnom, pogledi verovatno nisu dobri za kombinovanje jer će krajnji rezultat biti kompleksan i zbunjujući pogled. U tom slučaju bi bilo bolje upravljati asocijacom pojedinačno kroz poglede. Za različite članove razvojnog tima potrebne su različite vrste informacija. Izbor pogleda na softversku arhitekturu direktno zavirsi od potreba članova tima koji se bavi razvojem softvera.

Q: Da li je moguće kombinovati bilo koji pogled softverske arhitekture sa nekim drugim pogledom?
A: ?
Q: Kada se primenjuje stil dodeljivanja radnih zadataka? Na koji način se vrši primena stila dodeljivanja radnih zadataka?
A: Stil dodeljivanja radnog zadatka se koristi za podelu sistema na module i dodeljivanje određenih modula timovima ili članovima tima koji je odgovoran za realizaciju sistema. Stil

definiše odgovornost za implementaciju i integraciju modula određenom razvojnom timu. Navedeni stil se uglavnom koristi za povezivanje aktivnosti sa resursima kako bi projektant softvera osiguran da svaki modul bude dodeljen pojedincu ili timu. Arhitektura u kombinaciji sa procesom određuje alokaciju. Radni zadaci predstavljaju mapiranje softverske arhitekture na grupe ljudi kroz stil dodeljivanja radnog zadatka. Timovi a saimim tim i radni zadaci nisu povezani sa pisanjem koda koji će se pokrenuti u finalnom sistemu. Postoji mnogo više zadataka koji ljudi moraju da obavljaju: upravljanje konfiguracijom, testiranje, evaluaciju potencijalnih komercijalnih proizvoda ili kontinuirano održavanje proizvoda. Stil dodeljivanja radnog zadatka vezan je za stil dekompozicije jer se na taj način najlakše vrši mapiranje alokacija. Prikaz stila dodeljivanja radnog zadatka može proširiti dekompoziciju modula na module koji odgovaraju razvojnim alatima, alatima za testiranje, sistemima za upravljanje konfiguracijama i slično čija nabavka ili svakodnevno korišćenje takođe može biti dodeljeno određenom članu ili timu. Primena stila dodeljivanja radnog zadatka omogućava projektantu da razmisli na koji način je moguće podeliti posao u delove kojima može upravljati.

Q: Koja je razlika između predstavljanja softverske arhitekture kroz stilove alokacije i kroz 4+1 Kručtenove poglede?

A: 4+1 Kručtova softverska arhitektura se najčešće koristi u procesu projektovanja i implementacije kao detaljan prikaz strukture softvera. Elementi softverske arhitekture su definisani kako bi rešili osnovne funkcionalnosti i korisničke zahteve. Takođe, elementi softverske arhitekture treba da omoguće i ispunjavanje zahteva za performansama i ostale nefunkcionalne zahteve ko što su skalabilnost, portabilnost i dostupnost. Upotrebom Kručtenovih pogleda softversku arhitekturu je potrebno predstaviti kroz:

1) Logički pogled, koji predstavlja projektovanje objektnog modela (u slučaju korišćenja objektno-orijentisane metode projektovanja)
2) Procesni pogled, koji beleži procese, način komunikacije i ponašanja sistema tokom izvršavanja određene funkcionalnosti
3) Fizički pogled, koji opisuje komponente softverske arhitekture iz ugla projektanta softvera
4) Pogled raspoređivanja, koji opisuje povezivanje softverskih komponenti sa hardverskim elementima softverskog okruženja

Q: Šta omogućava primena 4+1 Kručtenovih pogleda na softversku arhitekturu?

A: Primenom Kručtenovih pogleda omogućeno je sagledavanje arhitekture softvera iz logičkog, procesnog, fizičkog i pogleda raspoređivanja softverskih komponenti. Opisani pogledi na softversku arhitekturu zahtevaju modelovanje sledećih dijagrama:

1) Logički pogled: modelovanje klasnog dijagrama i dijagrama stanja
2) Procesni pogled: dijagram aktivnosti
3) Fizički pogled: dijagram komponenti
4) Pogled raspoređivanja: dijagram komponenti

Navedene dijagrame moguće je modelovati upotrebom alata koji podržava UML jezik i predviđenu grafičku notaciju. Pored modelovanja potrebno je detaljno dokumentovati dijagrame i opisati način predstavljanja softverske arhitekture kroz odabrani pogled.

Q: Zašto je potrebno kombinovati stil raspoređivanja i stil instalacije?
A: Stil instalacije je najčešće usko povezan sa stilom raspoređivanja i može se smatrati nastavkom prikaza softverske arhitekture nakon stila raspoređivanja. Pogled stila instalacije može biti projektovan da bi koristio različite varijacije jer se često može desiti da zahtevi instalacije podrazumevaju instalaciju na različitim hardverskim platformama. Kao i kod stila raspoređivanja, važni elementi softvera i elementi okruženja softvera utiču na raspodelu softverskih komponenti.

Q: Šta je specijalizacija stilova alokacije?
A: Specijalizacija stila alokacije se vrši u slučaju potrebe za ponovnim korišćenjem stila u različitim delovima određenog sistema.

Q: Šta su kombinovani pogledi na softversku arhitekturu? Koji su načini za kreiranje kombinovanje pogleda?
A: Najjednostavniji način za prikaz asocijacije između dva pogleda je da se pogledi grupišu u jedan kombinovani pogled. Kombinovani pogled najčešće smanjuje broj pogleda na softversku arhitekturu u dokumentu jer zamenjuje više različitih pogleda. Postoje dva načina za kreiranje kombinovanog pogleda:

1) razvoj dodatnog sloja (overlay) koji kombinuje informacije iz dva različita pogleda na softversku arhitekturu. Ovakav pristup dobro funkcioniše ako je asocijacija između dva pogleda jaka u smislu da postoji između dva softverska elemenata dva različita pogleda.
2) kreiranjem hibridnog stila,koji kombinuje dva postojeća stila u svrhu kreiranja vodiča za stil koji ukazuje na kombinovane stilove i opisuje sve nove dobijene elemente i tipove odnosa, osobine i ograničenja. Hibridni stilovi zahtevaju definisanje novih elemenata i novih tipova odnosa shodno kombinovanju dva različita stila. Hibridni stil je korisno razviti ako se stil koristi po nekoliko puta u istom sistemu ili na istim vrstama sistema razvijenim u jednoj organizaciji i ako postoji mnogo poznatih korisnika.

6 Upotreba šablona projektovanja softvera

Q: Navedite primer kada je moguće koristiti šablon Abstraction-Occurrence
A: Primer: Moguće je kreirati klasu <Apstrakcija> pod nazivom "TvSerija" a zatim je potrebno kreirati klasu <pojava> pod imenom "Epizoda". Takođe, moguće je napraviti <apstrakcija> klasu pod imenom "Publikacija" koja će sadržati ime, autora, isbn i datum objavljivanja publikacije. Klasa <pojava> koja odgovara takvoj klasi bila bi "BibliotečkaStavka" koja bi sadržala bar kod publikacije (knjige). (primer na slici 1).
Q: Koja su ograničenja u primeni šablona Abstraction-Occurrence?

A: Potrebno je prikazati članove svakog skupa pojava bez dupliranja zajedničkih informacija. Dupliranje informacija bi iskoristilo prostor i zahtevalo bi mejanje svih pojava u slučaju menjanja zajedničkih informacija. Takođe, potrebno je izbeći rizik od nekonzistentnosti koja bi rezultovana promenu zajedničkih informacija samo u određenim objektima. Potrebno je modelovati rešenje koje omogućava maksimalnu fleksibilnost sistema.

Q: Da li šablon General Hierarchy koristi nasleđivanje?
A: Ne to predstavlja anti-šablon.

Q: Koja asocijacija se koristi u šablonu General Hierarchy?
A: Koristi agregaciju i to može da bude opciono na više (optional-to-many) ili više na više (many-to-many)

Q: U kojim situacijama je potrebno koristiti šablon Player-Role?
A: Primena ovog šablona može rešiti problem predstavljanja klasa različitog tipa u okviru klasnog dijagrama. Uloga ("role") predstavlja skup mogućnosti povezanih sa objektom u određenom kontekstu. Objekat može "igrati" ("play") određene uloge u određenom kontekstu. Primer može biti, student koji je na Univerzitetu i može biti diplomirani ili ne diplomirani student u određenom trenutku i on može promeniti ulogu iz jedne u drugu. Takođe, student može biti registrovan na kurs u punom vremenu ili u određenom vremenskom intervalu. Student po potrebi može menjati tip vremena koji provodi na kursu.
Example: Potrebno je kreirati klasu <<Igrač>> koja prezentuje objekte koji "igraju" ("play") različite uloge. Pored toga, kreirati asocijaciju od ove klase ka apstraktnoj klasi <<ApstraktnaRola>> koja je superklasa seta svih mogućih uloga. Podklasa ove klase <<Rola>> obuhvata sve funkcije povezane sa različitim ulogama. Ako "igrač" može "igrati" samo jednu ulogu u isto vreme, veza između "igrača" i "uloge" može biti jedan na jedan ili će biti jedan na više. Umesto da bude apstraktna klasa, klasa "Rola" može biti interfejs. Jedini nedostatak ovakvog pristupa je da klasa "Rola" obično sadrži mehanizam povezan sa podklasama koji im omogućava da pristupe informacijama iz klase < >. U tom slučaju potrebno je jedino napraviti < > kao interfejs ukoliko navedeni mehanizam nije potreban.

Q: Šta predstavlja "uloga" (role) u primeni šablona Player-Role?
A: Uloga ("role") predstavlja skup mogućnosti povezanih sa objektom u određenom kontekstu. Objekat može "igrati" ("play") određene uloge u određenom kontekstu.

Q: Da li primena šablona Delegation omogućava uštedu resursa potrebnih za razvoj softvera?
A: Da zato što omogućava reupotrebu operacija u klasi, odnosno ako operacija već postoji implementirana u okviru druge klase onda je možemo iskorisiti. Nije prihvatljivo od klase napraviti podklasu i naslediti operaciju ukoliko se ne koriste sve metode druge klase.

Q: Da si primenom šablona Delegation duplira programski kod?
A: Ne duplira se.
Q: Šta predstavlja nepromenljiv objekat?

A: Nepromenljiv objekat koji ima stanje se nikada ne menja nakog njegovog kreiranja. Bitan razlog za korišćenje nepromenljivog objekta je što drugi objekti mogu verovati svom sadržaju bez neočekivanih izmena. Potrebno je proveriti da li je konstruktor nepromenljive klase jedino mesto gde se vrednosti varijabli postavljaju ili modifikuju. Uveriti se da metode bilo koje instance nemaju neželjene efekte promenom varijabli instance ili metoda pozivanja. Ako metoda koja bi inače izmenila varijablu instance mora biti prisutna, onda ona mora vratiti novu instancu klase.

Q: Koji su srodni šabloni šablonu Immutable?
A: Šablon "Read-only Interface" omogućava iste mogućnosti kao i šablon Immutable.

Q: Kada se koristi šablon Read-Only Interface?
A: Šablon Read-Only Interface može biti korišćen da šalje podatke ka objektima u grafičkom korisničkom interfejsu. Ovaj šablon je usko povezan sa šablonom Immutable. Ukoliko je ponekad potrebno kreirati određene privilegovane klase koje će moći da modifikuju atribute objekata su inače nepromenljivi.

Q: Koje su prednosti korišćenja šablona Read-Only Interface?
A: Ovaj šablon omogućava da se ne vrše neautorizovane izmene podataka.

Q: Šta je klasa "teške kategorije"? (heavyweight class)?
A: Često je potrebno dosta vremena kako bi se pristupilo instanci klase. Takve klase nazivaju se klase teške kategorije ("heavyweight classes"). Instance od klasa teške kategorije mogu uvek biti u bazi podataka. Da bi se instance koristile u programu potrebno je da ih konstruktor učita sa podacima iz baze podataka. Slično tome objekti teške kategorije mogu postojati samo na severu, pre korišćenja objekta klijent treba da pošalje zahtev a nakon toga sačeka da objekat stigne. U obe situacije postoji vremensko kašnjenje i kompleksan mehanizam koji uključuje stvaranje objekata u memoriji. Mnogi drugi objekti u sistemu možda imaju potrebu da pozovu ili koriste instance klasa teške kategorije.

Q: Koji su srodni šabloni šablonu Proxy?
A: Šablon Delegation je sličan šablonu Proksi i može se koristiti zajedno sa ovim šablonom.

Q: Šta predstavlja primetan sloj?
A: Drugo proširenje OCSF (Objektni klijent-server okvir) okvira je dodavanje primetnog sloja. Poruku koju primi klijent obrađuje podklasa "ApstraktniKlijent" koju implementira metoda "obradiPorukuOdServera". Svaki put kada se razvije nova aplikacija, klasa "ApstraktniKlijent" mora biti podklasa.

Q: Koji su glavni nedostaci primene šablona projektovanja?
A: ?

Q: Na koji način primena šablona omogućava povećanje fleksibilnosti projektovanog sistema.

A: Primenom šablona projektovanja omogućeno je kreiranje preciznijih i detaljnih modela klasa. Šabloni projektovanja pomažu u izbegavanju grešaka i čine sistem fleksibilnim i jednostavnijim.

7 Strategije i metodi projektovanja softvera

Q: Na koji način se dobija deklarativno znanje o korišćenju metoda?
A: Deklartivno znanje opisuje zadatke koje treba izvršiti u svakom koraku procesa projektovanja. Deklarativno znanje predstavlja kroz opis: “uradi to, a onda uradti to” a proceduralno znanje se stiče iskustvom projektanata koji koristi metodu.

Q: Kako skup heuristika utiče na aktivnosti u procesu projektovanja softvera?
A: Omogućava smernice o načinu definisanja određenih aktivnosti unutar procesnog dela i organizacije određenih klasa problema. Bazirani su na iskustvu i prethodnoj upotrebi metode sa specifičnim domenom problema. Skup heuristika se sastoji od nekoliko različitih tehnika koje se preporučuju za upotrebu u različitim situacijama i za rešavanje različitih problema. Heuristike se generišu u određenom vremenskom periodu a iskustvo primene heuristika se stiče korišćenjem metode u širem problemskom domenu.

Q: Navedite primer tehničkih problema koji mogu nastati primenom metoda projektovanja softvera.
A: Tehnički problemi se sastoje od problema vezanih za projektovanje. Određeni broj tehničkih problema treba uzeti u obzir sa relativnom važnošću jer su zavisni od spoljnih faktora kao što je struktura razvojne organizacije i priroda samog problema projektovanja.

Q: Šta je dizajn virtuelna mašina i u kojim slučajevima se koristi?
A: Svaka metoda projektovanja pruža poseban oblik nazvan "Dizajn virtuelne mašine" (DVM) koji predstavlja okvir koji projektant softvera treba da razvije i pridržava se u toku projektovanja. Navedene tačke upotrebe metoda projektovanja su većinom povezane sa velikim sistemima ali su primenljive i na malim sistemima. Jedan od razloga zašto projektanti softvera mogu da projektuju sistem na osnovu svog iskustva je taj što su razvili svoje "dizajn virtuelne mašine" za određene klase problema sa kojima su se susretali. Jedna od glavnih komponenti DVM- a je stil softverske arhitekture koji određuje oblik setova elemenata projektovanja i definiše okruženje koje se koristi za kasniju implementaciju. Primena koncepta virtuelne mašine nije novost, ali se ne koristi tako često u metodama projektovanja. U primeni u računarstvu, virtuelna mašina, predstavlja sloj apstrakcije iznad sloja fizičke mašine. Operativni sistem obezbeđuje određeni broj virtuelnih računara kako bi programerima bio omogućen pristup resursima računara na različitim nivoima apstrakcije. Programski jezici takođe omogućavaju virtuelne mašine na kojima se izvršava programski kod implementiran od strane programera. Na slici 1 prikazana je osnovna ideja opisanog pristupa.

Q: Šta su dekompozicione metode? Kako se primenjuju?
A: Strategija dekompozicije je podela glavnog zadatka programa na manje zadatke koje je moguće realizovati kao podprograme. Većina programskih jezika koja je postala široko dostupna (asembler i Fortran) omogućavaju prilično moćne mehanizme za opisivanje strukturiranja programa korišćenjem podprograma ali nisu imali sredstva za kreiranje i korišćenje složenih struktura podataka. Strategija je primenjena (Niklaus Wirth (1971)) i omogućava praktične načine implementacije projektovanog rešenja. U najosnovnijem obliku uspeh ovog pristupa zavisi od načina na koji se opisuje izvorni problem koji predstavlja model koji je osnova za početni izbor projektanta softvera. Na slici 1 dat je primer dva načina razgradnje datog problema. Odluke u toku dekompozicije "odozgo na dole" moraju biti donete na početku procesa projektovanja ili će efekti loše odluke biti vidljivi kroz kompletan proces projektovanja. Loša odluka može dovesti do ponovnog projektovanja sistema.

Q: Na koji način je moguće primeniti proceduralni model procesa projektovanja softvera?
A: ?

Q: Na koji način se vrši projektovanje po kompoziciji?
A: Drugačiji pristup (suprotan od dekompozicije) predstavlja kompoziciona strategija za izgradnju modela projektovanja. Primenom strategije "odozgo na dole" uglavnom se identifikuju operacije koje sistem mora izvršiti. Dobijeni model problema predstavlja razmatranje funkcionalnosti koje se kasnije mogu razmatrati i dodatno objasniti. Model kompozicije je zasnovan na razvoju opisa za određeni skup entiteta objekata koji mogu biti prepoznati kao mogući problemi. Pored opisa entiteta, opisuju se i veze između tih entiteta. Takođe, entiteti u okviru modela variraju shodno problemu koji je potrebno predstaviti.

Q: Koji su nedostaci u primeni metoda projektovanja softvera?
A: Metod projektovana softvera ne omogućava automatsko uklanjanje svih problema. Metoda omogućava projektantu softvera okvir u kome može organizovati proces. Predstavlja skup preporučenih šablona koji mogu dati uvid u probleme koji se mogu identifikovati u nekoj fazi projektovanja. Takođe, omogućava uputstva o koracima koje je potrebno izvršiti u procesu projektovanja, savete koje je potrebno uzeti u obzir u određenim koracima kao i o kriterijumima koje je potrebno zadovoljiti prilikom projektovanja. Bitno je naglasiti da nijedna od navedenih smernica ne može biti specifična za probleme, već je potrebno akcenat postaviti na dobijanje i primenu ideje koja može uz metode projektovanja softvera rešiti određeni problem.

Q: Kako se izvršava transformacija u procesu projektovanja softvera?
A: ?

Q: Kako faktori organizacije utiču na proces projektovanja softvera?
A: Faktori organizacije najčešće ne utiču direktno na deo reprezentacije i njihov uticaj može
Primer mogu biti međunarodne agencije kao i organi centralne i lokalne samouprave koji su glavni kupci softverskih sistema. Takvi sistemi su uglavnom specijalizovani sistemi za obradu podataka u toku radnog vremena organizacije , sistemi za kontrolu zaliha ili oporezivanje.

Većina ovih sistema je veoma velika i jako teško je precizirati zahteve (mogu se menjati primenom tehnologije, zakonodavstva ili unutrašnjom reorganizacijom).

Q: Kako faza elaboracije i transformacije utiče na proces projektovanja softvera?
A: Faza elaboracije ne podrazumeva interpretaciju kroz različite poglede i uglavnom se bavi restrukturiranjem ili reorganizacijom modela projektovanja u okviru trenutnog pogleda. Svrha koraka elaboracije je da omogući dodatne informacije u modelu i prikaz trenutnog stanja planiranja projektovanja kroz restrukturiranje u cilju moguće transformacije.

Q: Koje informacije projektant dobija iz faza procesa projektovanja?
A: ?

8 Tradicionalni metodi projektovanja softvera

Q: Šta čitalac dokumentacije softverske arhitekture može da vidi iz "P-Spec" specifikacije procesa?
A: Ovakav tip specifikacije procesa, "P-Spec", omogućava tekstualni opis i funkcionalni pogled osnovnog procesa koji je predstavljen krugom u dijagramu toka podataka. Specifikacija će obuhvatiti opis procesa kroz:

- Naziv procesa
- Opis procesa
- Ulazno/izlazno podatke iz procesa
- Proceduralne zadatke

Q: Šta je rezultat strukturnog projektovanja softvera?
A: ?

Q: ???????

9 Ponovna upotreba softvera

Q: Zašto se pri razvoju softvera koristi i stari softver (ranije razvijene softverske komponente pripremljene za ponovnu upotrebu i dr.)?
A: Ponovna upotreba postojećeg softvera povećava pouzdanost sistema, smanjuje rizike procene troškova, ubrzava razvoj, povećava upotrebu standarda i omogućava efektivno korišćenje ljudi. Softversko inženjerstvo bazirano na ponovnoj upotrebi postojećeg softvera (software reuse), je strategija razvoja čiji proces razvoja uključuje i ponovnu upotrebu postojećeg softvera. Ona je posledica težnji da se smanje troškovi razvoja i održavanja softvera,

da se ubrza isporuka novog softvera i da se poveća kvalitet softvera. Pokret promocije upotrebe softvera otvorenog koda je doprineo dramatičnom povećanju upotrebe postojećeg softvera u delovima novog softvera. Primenom komponenti za ponovnu upotrebu softvera, mogu se "krojiti" novi i prilagođeni softverski sistemi potrebama njihovih korisnika. Novi standardi, kao što su standardi za veb servise, doprineli su lakšem razvoju servisno- orijentisanih softverskih sistema preko veba, i to u mnogim domenima primene.

Q: Koje su koristi od ponovne upotrebe ranije razvijenih softverskih jedinica?
A: Primena postojećeg softvera u delovima novog softvera smanjuje ukupni trošak razvoja i održavanja softvera, jer se manji broj komponenata mora razvijati. Sledeća tabela prikazuje i druge koristi.

Q: Zašto ponovna upotreba softvera povećava pouzdanost softvera?
A: Softvare koji je prethodno korišćen i testiran je poudaniji od novog softvera. Njegove greške su utvrđene i otklonjene.

Q: Zašto ponovna upotreba softverskih jedinica smanjuje rizik razvoja novog softvera?
A: Za razliku od novog softvera, cena postojećeg softvera je poznata. Ovo smanjuje marginu greške pri proceni troškova projekta. Ovo je naručito važno pri razvoju velikih sistema sa velikim komponentama.

Q: Zašto ponovna upotreba softverskih jedinica povećava efektivnost upotrebe softverskih specijalista?
A: Umesto da ponovo razvijaju već razvijeno, softver specijalisti se usmeravaju da razviju komponente za ponovnu upotrebu

Q: Zašto ponovna upotreba softverskih jedinica povećava usaglašenost sa standardima?
A: Neki standardni, npr. Za grafičke korisničke interfejse se mogu korisiti komponente za ponovnu upotrebu. Aplikacije koje koriste standardne interfejse su pouzdaniji jer ih korisnici poznaju i prave manje grešaka u korišćenju.

Q: Zašto ponovna upotreba softverskih jedinica ubrzava razvoj softvera?
A: Često važnije lansirati softver što pre na tržištu, nego troškovi njegovog razoja. Softver sa ponovno upotrebljenim komponentama se brže razvija, jer je vreme razvoja i ispitivanja smanjeno.

Q: Koji se problemi mogu javiti pri razvoju softvera uz upotrebu ranije razvijenih softverskih jedinica? Objasni svaki od ovih problema.
A: Ponovna upotreba softvera može da poveća troškove: održavanja-ako nema izvornog koda i ako nedostaju alati za podršku, održavanja biblioteka, nalaženja i prilagođavanja komponenti. Primena postojećeg softvera, pored prednosti, donosi i određene probleme. Utvrđivanje da li se neka komponenta može ponovo iskoristiti i njeno testiranje u okviru novog softvera čine određene troškove. Ti dodatni troškovi smanjuju ukupan troškovni efekat primene postojećeg

softvera pri razvoju novog softvera. Ponovnoa upotreba softvera je naefektivnija kada se ona planira kao deo organizacijske politike upotrebe postojećeg softvera.

Q: Koje bi faktore imali u vidu pri izboru softvera za ponovnu upotrebu?
A: Da li postoji održavanje, da li postoji podrška za te komponente, održavanje biblioteka, nalaženje i prilagođavanje komponenti.

Q: Šta je radni okvir (framework) za razvoj softvera? Koje su tri kategorije radnog okvira?
A: Radni okvir (framework) je opšta (generička) struktura koja se proširuje radi kreiranja specifičnog podsistema ili aplikacije. Praksa je pokazala da su softverski objekti često isuviše mali i isuviše specijalizovani za primenu u razvoju određene aplikacije. Pokazalo se da je potrebno duže vreme za njihovo razumevanje i prilagođavanje objekta novoj aplikaciji, nego što je potrebno da se razvije novi objekat kakav je potreban aplikaciji. Sada znamo da je kod razvoja objektno-orijentisanih softverskih sistema da je ponovna upotreba objekata izglednija kada se koriste krupnije apstrakcije koje nazivamo radnim okvirima (frameworks). Radni okvir (framework) je opšta (generička) struktura koja se proširuje radi kreiranja specifičnog podsistema ili aplikacije. Schimdt (2004) je definisao radni okvir na sledeći način: "...integrisan skup artifakata (kao što su klase, objekti i komponente) koje međusobno sarađuju da bi obezbedile ponovljenu arhitekturu za familiju sličnih aplikacija." Radni okviri obezbeđuju opšte funcije koje se najčešće koriste u svim aplikacijama sličnog tipa.

Q: Objasni radni okvir za razvoj veb aplikacija (MVC šablon). Koju funkcionalnost obezbeđuje MVC šablon?
A: Radni okviri su često implementacije određenih šablona projektovanja ili arhitektonskih šablona. Najčešće koriste MVC šablon, ali i druge šablone. MVC šablon odvaja stanje aplikacije od korisničkog interfejsa. MVC radni okvir podržava predstavljanje podataka na različite načine i dozvoljava međusobnu interakciju ovih različitih predstavljanja. Kada dođe do promene podatka u jednom predstavljanju, dolazi do promene modela sistema, te i do promene tog podatka u svim drugim predstavljanjima.

Q: Šta su povratni poyivi (callbacks)?
A: Pri proširenju radnog okvira vi ne menjate njegov kod. Vi dodajete konkretne klase koje nasleđuju operacije iz apstraktnih klasa radnog okvira. Možete i definisati povratne pozive. Povratni pozivi (callbacks) su metodi koji se pozivaju kao reakcija na događaje koje prepoznaje radni okvir. Scmiidt (2004) ih naziva "inverznom kontrolom". Objekti radnog okvira su odgovorni za kontrolu rada sistema. Kao odgovor na događaje generisani sa korisničkog interfejsa ili iz baze podataka, ovi objekti radnog okvira pozivaju metode veze (hook methods) koji su povezani sa korisničkim funkcijama.

Q: Šta je linija proizvoda (software product line)? Kako se projektuje jezgro softverskog sistema?
A: Jedan od najefetivnijih pristupa u primeni postojećeg softvera je kreiranje linija softverskog proizvoda ili familija aplikacija. Linija softverskog proizvoda (software product line) je skup aplikacija sa zajedničkom arhitekturom i deljivim komponentama, pri čemu je svaka aplikacija

specijalizovana prema zadovoljenju različitih zahteva. Jezgro sistema je projektovano tako da može da se konfiguriše i prilagodi potrebama različitih korisnika sistema. To može da obuhvati konfigurisanje nekih od komponenata, uz primenu dodatnih komponenata i uz modifikaciju nekih od komponenata u skladu sa novim zahtevima.Razvoj aplikacija prilagođavanjem jedne uopštene verzije aplikacije znači da se jedan veliki deo koda ove opšte aplikacije ponovo koristi. Kako se i iskustvo u primeni prenosi iz projekta u projekat, ova praksa skraćuje vreme uhodavanja novih članova tima za razvoj.

Q: Koja je razlika između aplikacionog radnog okvira i linje softverskog proizvoda?
A: Aplikacioni radni okviri i linije softverskog proizvoda imaju dosta zajedničkog. Oba pristupa koriste zajedničku arhitekturu i zajedničke komponente, i zahtevaju novi razvoj pri kreiranju nove verzije sistema. Ovde sa navode njihove međusobne razlike:

1) Aplikacioni radni okviri zavise od OO svojstava, kao što je nasleđivanje i polomorfizam radi proširenja radnog okvira. Programski kod radnog okvira se ne menja. Kod linija softverskih proizvoda, njihov kod se menja.
2) Aplikacioni okviri primarno obezbeđuju tehničku podršku razvoju, umesto domenske podrške. Linije softverskog proizvoda obično sadrže i domenske i platformske informacije.
3) Linije softverskih proizvoda obično kontrolišu neku opremu. Aplikacioni radni okviri retko daju podršku vezivanja hardvera.
4) Liniju softevrskog proizvoda čini familija sličnih aplikacija organizacije. Ne koriste opšte jezgro neke aplikacije, već menjaju jednu od aplikacija linije, koja se najmanje razlikuje.

Q: Kako se može razviti specijalizovani tip softverskog proizvoda primenom lije softverskog proizvoda.
A: Ako razvijate liniju softverskog proizvoda upotrebom nekog objektno-orijentisanog programskog jezika, onda možete koristiti aplikacioni okvir kao osnovu za razvoj sistema. Kreirajte jezgro linije softverskog proizvoda proširenjem radnog okvira sa domenskim specifičnostima određene kompanije i ugradnjom njenih mehanizama. U drugoj fazi razvoja, kreiraju se različite verzije sistema za različite kupce. Na ovaj način, mogu da se razviju različiti specijalizovani tipovi softverskog proizvoda.

Q: Kako se vrši konfigurisanje softverskog sistema tokom procesa njegovog razvoja?
A: Konfigurisanjem u vreme projektovanja se postiže veća fleksibilnost nego pri konfigurisanju u vreme raspoređivanja. Linije softverskih proizvoda se projektuju tako da mogu da se rekonfigurišu dodavanjem ili uklanjanjem komponenata iz sistema, definišući parametre i ograničenja komponenata sistema, i uključivanjem znanja o poslovnim procesima. ada se sistem konfiguriše u vreme projektovanja, isporučilac počinje sa generičkim sistemom ili sa postojećim proizvodom. Modifikacijom i proširenjem modela prvog sistema, oni kreiraju specifičan sistem koji obezbeđuje zadovoljenje specifičnih zahteva kupca. To često zahteva promenu i proširenje izvornog koda sistema, tako da se postiže veća fleksibilnost nego pri konfigurisanju u vreme raspoređivanja. Konfigurisanje u vreme raspoređivanja zahteva upotrebu alata za konfigurisanje radi variranja specifične konfiguracije sistema koja se zapisuje u bazu konfiguracija ili u skup konfiguracionih fajlova.

Q: Kako se vrši konfigurisanje softverskog sistema u vreme njegovog rapoređivanja 'stavljanja u upotrebu)? Koji su nivoi konfigurisanja sistema?
A: ?

Q: Šta je komercijalni gotov proizvod (a commercial-off-the-shelf ili COTS)? Koje su prednosti njihove primene?
A: Komercijalni gotov proizvod (a commercial-off-the-shelf ili COTS) je softverski sistem koji se može dalje prilagođavati potrebama različitih kupaca bez promene njegovog izvornog koda. Skoro svi desktop sistemi i deo serverskih sistema pripadaju kategoriji COTS softvera. Kako je COTS softver namenjen opštoj upotrebi, on obično sadrži puno funkcija. Zbog toga, može da se koristi u različitim radnim okruženjima i može biti deo različitih aplikacija. COTS proizvodi se prilagođavaju potrebama kupaca primenom ugrađenih mehanizama konfigurisanja koji dozvoljavaju krojenje funkcionalnosti sistema u skladu sa potrebama specifičnog kupca. Ovaj pristup ponovne upotrebe softvera je danas široko prihvaćen i primenjen od strane velikih kompanija, jer obezbeđuje značajne koristi u odnosu na razvoj novog proizvoda u skladu sa zahtevima kupca. Prednosti:

1. Omogućeno je brzo raspoređivanje pouzdanih sistema.
2. Moguće je videti funkcionalnost koja je obezbeđena, što olakšava procenu da li nam proizvod odgovara ili ne.
3. Izbegnuti su neki rizici razvoja upotrebom postojećeg softvera.
4. Poslovanje se može usresrediti na ključnu aktivnost bez odvajanja mnogo resursa na razvoj IT sistema.
5. Kako se izvršne platforme usavršavaju, ugradnja novih tehnologija postaje uprošćena jer to obezbeđuje proizvođač COTS proizvoda, umesto da to radi kupac.

Q: Koji se problemi mogu javiti pri primeni gotovih komercijalnih softverskih proizvoda?
A: Problemi koji mogu da se jave:
1. Zahtevi se često moraju prilagođavati funkcionalnostima i načinu rada COTS proizvoda. To može da dovede do naglih promena u postojećim poslovnim procesima.
2. COTS proizvod se može bazirati na pretpostavkama koje je kasnije praktično nemoguće promeniti.
3. Izbor odgovarajućeg COTS sistema za neko preduzeće može biti težak proces, naročito ako postoji mnogo COTS proizvoda koji nisu dobro dokumentovani. Izbor pogrešnog COTS proizvoda može dovesti do toga i da novi sistem ne radi, kako je zahtevano.
4. Može da postoji nedostatak lokalnog znanja za razvoj sistema, te kupac onda postaje zavistan od proizvođača i spoljnih konsultanata radi saveta u vezi razvoja. Ovi saveti mogu biti pristrasni i podređeni cilju prodaje proizvoda i servisa, umesto da pomognu kupcu da zadovolji svoje zahteve.
5. Proizvođač COTS proizvoda kontroliše podršku sistema i nejgovu evoluciju. Mogu da nestanu iz biznisa, ili prodati, što sve može da dovede do teškoća kod kupaca.

Q: Objasni razliku između COTS-sistemskih rešenja (COTS-solution systems) i COTS-integrisanih sistema (COTS-integrated systems)?
A: COTS sistemsko rešenje predstavlja jedan COTS sistem. COTS integrisani sistemi predstavlja više integrisanih COTS sistema koji zajedno ostvaruju zahteve nove aplikacije. Ponovno korišćenje softvera primenom COTS proizvoda je postalo vrlo rašireno. Naveći broj novih poslovnih informacionih sistema se sada nude kao COTS proizvodi, umesto da se razvija objektno-orijentisan softver. Praksa je pokazala da ponovna upotreba softvera primenom COTS proizvoda dovodi do smanjivanja napora i vremena pripreme i raspoređivanja raspoređivanja sistema. Postoji dva tipa upotrebe COTS proizvoda:
• COTS sistemska rešenja (COTS-solution systems), koji sadrže generičku aplikaciju razvijenu od jenog proizvođača a koja je ekonfigurisana u skladu sa zahtevima kupca.
• COTS integrisani sistemi (COTS-integrated systems), koji integrišu dva ili više COTS sistema)moguće proivedenih od strane različitih proizvođača), a da bi kreirali željenu aplikaciju.

Q: Opiši tipičnu arhitekturu ERP sistema. Koja su ključna svojstva ove arhitekture?
A: ERP sistemi se koriste kod skoro svih velikih kompanija radi podrške nekim ili svim poslovnim funkcijama. Glavno ograničenje ovog načina ponovne upotrebe softevera je funkcionalnost koja je ugrađena u generičko jezgro sistema, jer neka kompanija ne može da se tome prilagodi. Drugo ograničenje je zahtev da se poslovni procesi i operacije izraze u jeziku konfigurisanja sistema, a to može da dovede do neslaganja između koncepata poslovanja kompanije i koncepata podržanih jezikom konfigurisanja.

Q: Navedi uobičajene aktivnosti konfigurisanja COTS-sistema, kao što su ERP sistemi.
A: ?

Q: Kako bi izabrao COTS sistem? Koje faktore bi uzeo u obzir?
A: ?

Q: Koji se problemi mogu javiti pri razvoju softvera integracijom više COTS proizvoda?
A:

10 Projektovanje softvera primenom komponenata

Q: Šta je softverska komponenta? Nevedite pet karakteristika softverskih komponenata i objasnite ih.
A: Komponenta je softverska jedinica koja se može spojiti sa drugim komponentama radi kreiranja jednog softverskog sistema.Komponente su apstrakcije višeg nivoa u odnosu na klase i definišu se svojim interfejsima. Komponente su najčešće veće od pojedinačnih klasa i njihovi implementacioni detalji (izvorni kod) su sakriveni u odnosu na druge komponente. Proces projektovanja pomoću komponenata obihvata aktivnosti definisanja, implementacije i integracije ili slaganja slabim vezama, nezavisnih komponenata u sisteme. Kako softverski sistemi postaju sve veći i sve složeniji, to se razvoj softvera zasnovan na primeni komponenata sve više koristi i

postao je vrlo značajan u softverskom inženjerstvu. Da bi se do novih softverskih sistema došlo brže, sa što manje troškova, a da se pri tom dobije kvalitetniji i pouzdaniji softver, potrebno ga je razvijati uz pomoć komponenti. ali u najvećoj meri, ukoliko je moguće, već ranije razvijeninih i korišćenih (te proverenih i testiranih). Ponovna upotreba komponenti je ključ za razvoj složenih, a pouzdanih sistem, za kraće vreme i sa nižim troškovima. Komponente su međusobno nezavisne, komuniciraju preko svojih interfejsa koji sakrivaju njihovu implementaciju i obezbeđuju standardne servise, čime smanjuju obim koda koji treba razviti. Karaketristike:

1) Standardizovane
2) Nezavisne
3) Kompozitni
4) Raspodeljiva
5) Dokumentovani

Q: Objasnite dve kritične komponente softverske komponente: nezavisnost i nuđene servisa posredstvom interfejsa komponente? Koja je uloga interfejsa u slučaju primene softverskih kompšonenata.

A: Posmatranjem komponente kao provajdera servisa naglašava dve kritičke karakteristike komponente za ponovnu upotrebu:

1. Komponenta je nezavisni izvršni entitet definisan svojim interfejsima. Ne morate da znate njen izvorni kod, niti da li se koristi kao servis ili je uključena direktno u program, tj. u sistem koji se razvija.
2. Servisi koje nudi komponenta su raspoložive preko interfejsa i sve interakcije se odvijaju samo preko tog interfejsa. Interfejs komponente se izražava pomoću parametrizovanih operacija i ne prikazuje svoje unutrašnje stanje.

Q: Postoje dve vrste interfejsa komponenata. Koje su to vrste i objasnite razliku između njih.

A: Komponente imaju dve vrste interfejsa, vezanih za servise koje pruža i koje potražuje:

1) Interfejs "obezbeđuje" ("provides") definiše servise koje obezbeđuje komponenta. Ovaj interfejs je API komponente. On definiše metode koji se mogu pozivati od strane korisnika komponente. U UML dijagramu komponenata, intervejsi koje komponenta obezbeđuje označavaju se punim krugom, povezan linijom sa komponentom.

2) Interfejs "potražuje" ("requires") specificira servise koje treba da joj obezbede druge komponente da bi komponenta ispravno radila. U UML-u, interfejs koji potražuje servise označava se otvorenim polukrugom i linjom spajanja sa interfejsom.

Q: Šta je model komponente? Koji su osnovni elementi modela komponenata? Šta oni sadrže?

A: Model komponente je standard za implementaciju, dokumentaciju i raspoređivanje komponente. Ovi standardi obezbeđuju interoperabilnost komponenata. Oni su značajni i za proizvođače posredničkog softvera (middleware) koji treba da podržava rad komponenata.
Osnovni elementi:

1. Interfejsi komponenata: se definišu specificiranjem njihovih interfejsa. Model komponente specificira kako se definišu interfejsi i njihovi elementi, kao što su imena operacija, parametri i

izuzeci, koje takođe treba uključiti u definicuju interfejsa. Model takođe definiše jezik koji se koristi za defnisanje interfejsa komponenata. Za veb servise, to je WSDL, za EJB je Java, a za .NET interfejse se definišu u Common Itermediate Language (CIL).
2. Potreba: Komponente moraju da imaju svoja jedinstvena imena. Meta podaci daju informaciju interfejsima i atributima. Njih koriste korisnici da bi utvrdili koje servise komponenta može da im pruži. Pri raspoređivanju, komponenta mora da se konfiguriše u skladu sa određenim okruženjem u kojme se koristi.
3. Raspoređivanje: Model sadrži specifikaciju kako bi trebalo da se komponenta upakuje radi raspoređivanja. Mora da sadrži podržavajući softver koju ne sadrži infrastrikutura komponente ili koji nije definisan u interfejsima zahteva. Informacija raspoređivanja sadrži informaciju o sadržaju paketa i njene binarne organizacije. Komponente mogu da sadrže i pravila za njihovu zamenu. Model komponente može da definiše i dokumentaciju komponente koja mora da se pripremi i doda.

Q: Koja je razlika između servisa platforme i servisa podrške?
A:

1) Servisi platforme, koji omogućavaju komponentama da komuniciraju i međusobno usaglašeno rade u distribuiranom okruženju (instalirani su u različitim serverima i čvorovima na mreži). Ovo su osnovni servisi koji moraju biti ponuđeni svim sistemima koji koriste komponente.
2) Servisi podrške, koji se nude kao zajednički servisi koji eventualno mogu biti korišćeni od strane nekih različitih komponenata.

Q: Procesi softverskog inženjerstva zasnovanim na komponenatama omogućavaju projektovanje softvera primenom komponenata (PSPK) ili, na engleskom, CBSE procesi (Component-Based Software Engineering). Postije dva tipa PSPK: Razvoj za ponovnu upotrebu (development for reuse) i Razvoj sa ponovnom upotrebom (development with reuse). Koja je razlika između ova dva tipa projektovanja softvera primenom komponenata?
A:
1. Razvoj za ponovnu upotrebu (development for reuse): Ovaj proces se bavi razvojem komponenata ili servisa koji će biti ponovo korišćeni u drugim aplikacijama. To obično podrazumeva primenu i uopštavanje postojećih komponenata.
2. Razvoj sa ponovnom upotrebom (development with reuse): To je proces razvoja novih aplikacija koji upotrebljava postojeće komponente i servise.

Q: Razvili ste softver i razmišljate da li da na osnovu njega napravite nekoliko kompoenata koje bi kasnije mogli da koristiti pri razvoju softvera. Kako ćete odlučiti? Koje faktore ćete uzeti u obzir prilikom odlučivanja?
A: ?

Q: Pri korišćenju softverskih komponenata, postavlja se pitanja rada sa izuzecima.
A: ?

Q: Šta se podrazumeva pod upravljanjem komponentama?
A: ?

Q: Navedite aktivnosti procesa projektovanja softvera primenom komponenti (PSPK). U čemu je specifičnost ovog procesa u odnosu na projektovanje softvrea bey primene komponenti?
A: ?

Q: Postoje tri vrste spajanja (sastavljanja) komponenata pri razvoju softvera. Nevedite te tri vrste i ukratko opišite.
A:

1) Sekvencijalno spajanje kreira novu komponentu spajanjem 2 postojeće komponente, njihovim sekevencijalnim pozivanjem. Kako obe komponente imaju interfejs kojim nude svoje servise, neophodno je ubaciti međusoftver koji preuzima informacije i sa jednog i sa drugog interfejsa. Ovaj tip spajanja se može primeniti i kod komponenata koje su element programa i koje su servisi.
2) Hijerarhijsko spajanje se realizuje kada jedna komponenta poziva servise drugog. Pozvana komponenta onda daje tražene servise drugoj komponenti. Ako dođe do neslaganja dva interfejsa, potrebna je konverzija koda. Ovaj tip spajanja se ne koristi kod veb servisa.
3) Aditivno spajanje je slučaj kada se dve ili više komponente sastavljaju radi kreiranja nove komponente koja kombinuje nihovu funkcionalnost. Komponente se nezavisno pozivaju preko spoljnjeg interfejsa složene komponente. A i B nisu međusobno zavisne i ne pozivaju jedna drugu. Ovaj tip spajanje se može primeniti i kod komponenata koje su element programa i koje su servisi.

Q: Pri povezivanju komponenata, često se suočavamo sa nekompatibilnim interfejsima komponenata. Koje su to nekompatibilnosti interfejsa?
A: U svakom slučaju spajanja, neophodno je pisanje "koda lepljenja", da bi se "uparili" interfejsi tražnje i nuđenja komponenata na pravi način. Pri pisanju interfejsa komponenata koje će se kasnije spajati radi kreiranja novog sistema ili složene komponente, potrebno je interfejse pisati tako da su oni kompatibilni, da bi se jednostavnije kreirala nova komponenta ili sistem. Međutim, kada su komponente nezavisno razvijane, njihovi interfejsi najčešće nisu kompatibilni, te se javljaju tri slučaja nekompatibilnosti:
1. Nekompatibilnost parametara: Operacije na svakoj strani interfejsa imaju isto ime, ali njihovi tipovi parametara ili njihov broj je rezličit.
2. Nekompatibilnost operacija: Imena operacija i kod interfejsa koji obezbeđuje ("provides") i koji zahteva ("requires") su različite.
3. Nekompletnost operacija: Interfejs komponente koji obezbeđuje servis ("provides") je podskup interfejsa koji zahteva servise ("requires") druge komponente ili suprotno. Problemi nekompatibilnosti se rešavaju razvojem adaptora koji usklađuje interfejse dve komponente. Komponenta adapter konvertuje jedan interfejs u drugi. Oblik adaptora zavisi od tipa spajanja.

Q: Šta su adaptori i čemu oni služe pri spajanju komponenata?
A: Problemi nekompatibilnosti se rešavaju razvojem adaptora koji usklađuje interfejse dve komponente. Komponenta adapter konvertuje jedan interfejs u drugi. Oblik adaptora zavisi od tipa spajanja

Q: Navedite najčešće odluke koje projektanti treba da donesu kada koriste komponente pri rayvoju softvera.
A: Kada treba da kreirate sistem sastavljan od komponenata, možete uvideti konflikt između funkcionalnih i nefunkcionalnih zahteva, potrebe da se sistem isporuči što pre, i potrebe koji se može menjati sa novim zahtevima. Odluke koje morate doneti su kompromisi koje moraju uzeti u obzir sledeće:
1. Koji sastav komponenata daje najveće efekte na zadovoljenje funkcionalnih i nefunkcionalnih zahteva sistema?
2. Koji sastav komponenata čini sistem lakši za prilagođavanje komponenata kada dođe do promena u zahtevima?
3. Koje ća svojstva nastati u sistemu? Ova svojstva se odnose na performanse i pouzdanost. Ovo možete utvrditi tek kada je sistem kompletiran i primenjen.

11 Projektovanje distribuiranih softverskih sistema

Q: Šta su distribuisani softverski sistemi? Zašto se oni koriste? Koje su prednosti koje njihova primena dovodi?
A: Distribuirani sistem je onaj koji se realizuje na nekoliko kompjutera, za razliku od centralizovanih sistema u kome su sve komponente softvera izvršavaju u jednom kompjuteru. Veliki softverski sitemi se projektuju i realizuju kao distribuirani sistemi. Distribuirani sistem je onaj koji se realizuje na nekoliko kompjutera, za razliku od centralizovanih sistema u kome su sve komponente softvera izvršavaju u jednom kompjuteru. Sledeće su prednosti primene distribuiranih softverskih sistema:
1. Deljenje resursa: Distribuirani sistem dozvoljava deljenje, tj. zajedničko korišćenje hardverskih i softverskih resursa umreženih računara, što racionalizuje potreban njihov kapacitet.
2. Otvrorenost: DIstribuirani sistemi su najčešće otvoreni sistemi, što znači da su projektovani da primenjuju standardne protokole, što mogućava kombinovanje opreme i softverskih komponenti različitih proizvođača.
3. Konkurentnost (concurrency): U jednom distribuiranom sistemu, nekoliko procesa mogu da se odvijaju u isto vreme na različitim kompjuterima u isto vreme. Ove procesi mogu (ali i ne moraju) da međusobno komuniciraju.
4. Proširivost (scalability): Mogućnosti distribuirani sistema se mogu povećavati dodavanjem novih računarskih resursa (npr. više procesora, više diskova i dr.) ova i u skladu sa novim zahtevima. To u praksi može biti ograničeno mogućnostima računarske mreže koja ih povezuje.
5. Otpornost u slučaju otkaza (fault tollerance): Zbog korišćenja nekoliko računara i dupliciranja informacija, distribuirani sistemi su otporni na neke otkaye hardvera i softvera. U slučaju nekog

otkaza, distribuirani sistemi najčešće reaguju tako što im se umanjuje brzina pružanja servisa, ali se retko događa da se oni u potpunosti obustave.

Q: Šta utiče na performanse distribuiranih softverskih sistema?
A: Performanse distribuiranih sistema najviše zavise od propusnog kapaciteta računarske mreže, od njenog trenutnog opterećenja, kao i od brzina svih računara distribuiranog sistema. Distribuirani sistemi su složeniji od centralizovanih softverskih sisetma. Zbog toga, oni su teži za projektovanje, implementaciju i testiranje. Teže je i razumete njihova nastala svojstva zbog složenosti interakcija između komponenata sistema i infrastrukture sistema. Najčešće njihove performanse ne zavise odlučujuće od brzine korišćenih procesora, već od propusnog kapaciteta računarske mreže koja ih povezuje, od njenog trenutnog opterećenja (radom drugih sistema), kao i od brzina svih računara koji čine delove jednog distribuiranog sistema. Prebacivanje resursa sa jednog dela sistema na drugi, može značajno uticati na performanse sistema.

Q: Projektan softvera treba da bude svestan problema upravljivosti distribuiranim sistemom. Na koja pitanja projektan treba da obrati pažnju pri projektovanju sistema, kako bi oni bolje upravljiv? Objasnite zašto su ta pitanja relevantna.
A: Postoji više pitanja povezana sa svojstvima kao što su: transparetnost, otvorenost, proširivost, bezbednost, kvalitet servisa i upravljanje otkazima na koja projektant mora da odgovori. Svaki čvor distribuiranih sistema daje svoju specifičnu funkcionalnost sistema, i sam za sebe, predstavlja jedan sistem. Zbog ove složenosti, postavlja se pitanje njegove kontrole. Nemoguće je imati jedan centralizovani sistem upravljanja u slučaju distribuiranih sistema. Zbog toga se mora računati na nepredvidljivost ponašanja distribuiranih sistema i toga mora da bude svestan njihov projektant. Pri projektovanju distribuiranih sistema, moraju se uzeti u obzir sledeća pitanja:
1. Transparetnost: Do koje mere bi trebalo da se distribuirani sistem predstavi korisniku kao jedan sistem?
2.Otvorenost: Da li treba projektovati sistem koji koristi standardne protokole interoperabilnosti, ili bi trebalo da koristi specijalizovane protokole, i time da ograniče slobodu projektanta?
3.Proširivost: Kako treba postaviti arhitekturu sistema tako da on bude proširiv? Kako projektovati ceo sistem tako da se njegov kapacitet može povećavati u skladu sa povećanjem zahteva za korišćenjem sistema (npr. povćanjem broja njegovih korisnika)?
4.Bezbednost: Kako definisati i primeniti bezbedonosna pravila koriščenja sistema na svim podsistemima koje čine jedan distribuirani sistem?
5. Kvalitet servisa: Kako specificirati kvalitet servisa sistema koji se isporučuje kupcima i kako ga realizovati (implementirati) da bi bio isporučen sa prihvatljivim kvalitetom servisa svim korisnicima?
6. Upravljanje otkazima: Kako se mogu otkriti otkazi u sistemu, šta učiniti da se minimizira njihov negativni efekat na druge komponente sistema i kako se mogu izvršiti popravke sistema?

Q: Šta je posrednički softver, tj. middlver (middleware)?
A: Posrednički softver (midlver, middleware) se koristi za mapiranje logičkih resursa koje program poziva u stvarne fizičke resurse. On se koristi i za upravljanje interakcijama između ovih resursa sistema.

Q: Šta su otvoreni distribuirani sistem? Šta podrazumeva njihova «Otvorenost»?
A: Otvoreni distribuirani sistemi su sistemi koji su sagrađeni u skladu s generalno prihvaćenim standardima. To znači da se komponente različitih dobavljača integrišu sistematski da one mogu da rade sa ostalim sistemskim komponentama. Na nivou mreže, otvorenost se podrazumeva jer se koriste Internet protokoli. Međutim, na nivou komponenti, otvorenost nije univerzalno primenjena. Otvorenost podrazumeva da su sistemske komponente nezavisno razvijene u bilo kom programskom jeziku i, ako su razvijene primenom standarda, one mogu da međusobno komuniciraju i rade.

Q: Šta je proširivost (scalability) distribuiranih sistema? Koje su tr i dimenzije proširivosti sistem? Objasnite ih.
A: Proširljivost (scalability) je svojstvo softvera da obezbeđuje visoki kvalitet servisa i kada se zahtevi za korišćenjem sistema povećavaju. Postoje tri dimenzije proširivosti sistema:
1. Veličina: Trebalo bi da bude mogućno da se dodaju resursi sistemu da bi on mogao da odgovori na povećan broj svojih korisnika.
2. Distribucija: Trebalo bi da bude mogućno da se komponente prostorno raštrkaju a da to ne umanji njegove performanse.
3. Upravljivost: Neohodna je uspešno upravljanje sistemom i kada se povećava njegova veličina, čak i u slučajevima kada su delovi sistema locirani u nezavisnim organizacijama.

Q: Šta je razlika između prišireња sistema (scaling-up) i povećanja sistema (scaling-out)?
A: Prišireње sistema (scaling up) podrazumeva zamenu komponenata sa moćnijim komponentama, a povećanje sistema (scaling out) podrazumeva povećanja povećanje broja istih komponenata, što je najčešće isplativije.

Q: Distribuirani sistem su ranjive na napade, te su manje bezbedni od centralizovanih sistema. Zašto? Nevdite i objasnite vreste napada kojima su distribuirani sistemi izloženi.
A: Distribuirani sistem su ranjiviji na napade i manje bezbedniji od centralizovanih sistema. Napadač, upadom u neki deo sistema "ulazi na mala vrata" u sistem, i onda može da pristupi i drugim delovima sistema. Postoje sledeće vrste napada na koje neki distribuirani sistem mora da se odbrani:
1. Presretanje (interception) kada se komunikacije između delova sistema presreću od strane napadača, tako da dolazi do pada poverenja i bezbednost sistema.
2. Presecanje (interception) kada se napadaju servisi sistema te oni ne mogu da se ostvare, onako kako se očekuje. To se postiže bombradovanjem nekog čvora sistema sa nelegitimnih zahtevima za servisima, da bi se sprečili njegove odgovore na legitimne zahteve.
3. Promena (modification) kada se menjaju podaci ili servisi u sistemu od strane napadača.
4. Fabrikacija (fabrication) kada napadač generiše informaciju koja ne bi trebalo da postoji i koju onda upotrebljava da bi stekao neke privilegije, tj. prava pristupa. Na primer, napadač može da generiše lažnu lozinku i da na osnovu nje uđe u sistem.

Q: Šta je kvalitet servisa? Koji problemi prate zahteve za većim kvalitetom servisa? Navedite neki primer.
A: Kvalitet usluga (quality of service, ili QoS) koje nudi distribuirani sistem odražava sposobnost sistema da isporući traženi servis pouzdano, sa prihvatljivim vremenom odziva, i prihvatljivim protokom informacija. U idealnom slučaju, zahtevi za kvalitetom servisa bi trebalo da budu specificirani unapred, i u skladu sa tim, bi sistem trebalo da se projektuje i da se konfiguriše. Na žalost, to nije uvek praktično iz sledećih razloga:
1. Nije prihvatljiva cena takvog sistema prilikom maksimalnog opterećenja. Zbog toga, najveći deo resursa stoji neiskorišćen. Najveći razlog za primenu klaud računarstva je upravo u ovome, jer se njegovim primenom optimalno koriišćenja raspoloživih resursa. Upotreba Računarstva oblaka (cloud computing), postiže se dodavanjem resursa kada poraste broj zahteva.
2. Parametri QoS mogu međusobno da budu kontradiktorni. Na primer, porast pouzdanosti može da smanji protok informacija, zbog procedura proveravanja radi obezbeđenja ispravnosti svih ulaza u sistem. QoS je posebno kritičan kada sistem radi sa kritičnim vremenskim podacima, kao što su audio i video strimovi.

Q: Kako se upravlja otkazima u slučaju distriburanih softverskih sistema?
A: Upravljanje otkazima obuhvata primenu tehnika za obezbeđenje otpornosti na otkaze. To znači da sistem treba da pronađe komponentu u kojoj je došlo do otkaza i da sistem nastavi da pruža što više servisa, bez obzira na otkaz u nekoj komponenti. Sistem bi trebalo da primeni mere za automatski opravak od otkaza u nekoj komponenti.

Q: Postoje dva osnovna modela interakcija između čvorova distribuiranog sistema. Koja su to dva modela i objasnite ih.
A: Postoje dva osnovna tipa interakcije među računarima distriburanog računarskog sistema:
• proceduralna interakcija, i
• interakcija sa porukama.
Proceduralna interakcija uključuje računar koji poziva poznati servis koji nudi neku drugi računar i obično čeka isporuku tog servisa. Interakcija porukama uključuje računar koji definiše informaciju o onome što bi trabalo da sadrži poruka, koja se onda šalje drugom računaru. Poruke obično prenose više informacija u jednoj interakciji nego proceduralna interakcija.

Q: Šta je proceduralni poziv (RPC)? Šta je tačka povezivanja (stub)? Gde se nalazi udaljena procedura i koja je njena funkcija?
A: Proceduralna komunikacija u distribuiranom sistemu se obično primenjuje sa proceduralnim pozivima (procedural calls, ili RPC). Proceduralni poziv (RPC) koristi komponenta koja poziva drugu komponentu kao u slučaju poziva lokalne operacije, tj. metoda. Midlver sistem presreće ovaj poziv i prosleđuje ga udaljenoj komponenti. On obavlja zahtevanu obradu podataka i preko midlvera, vraća rezultat komponenti koja je poslala poziv. U slučaja Jave, ovo se realizuje primenom pozivanja udaljene procedure (remote procedural calls ili RMI). RMI okvir obrađuje pozivanje udaljenih metoda u Java programu.

Q: Šta je interakcija sa porukeama? Oja je učaga midlvera u tome?
A: Interakcija porukama normalno uključuje komponentu koja kreira poruku koja sadrži detalje zahtevanog servisa koji nudi neka druga komponenta. Preko midlver sistema (posredničkog softvera), poruka se šalje komponenti koja treba da je primi. Komponenta koja prima poruka, vrši konverziju (čitanje) poruke, vrši odgovarajuću obradu podataka, i kreira poruku za komponentu koja je poslala poruku, a koja sadrži rezultat obrade podataka (tj. rezultat servisa koji je zahtevan). Poruka se daje midlveru radi transmisije komponenti koja je poslala poruku.

`Q: **Koja je razlika proceduralnog poziva (RPC) i razmene poruka? Objasnite..**
A: ?

Q: Koje su najčešće funkcije midlvera? Kako midlver podržava interakciju čvorova?
A: On je odgovaran za obezbeđenje prenosa poruke odgovarajućem u daljem sistemu. Midlver se nalazi u sloju između komponenata sistema i operativnog sistema. Obezbeđuje podršku interakcijama komponenti, kao i više zajedničkih servisa svojim komponentama. Komponente distribuiranog sistema mogu biti implementirane pomoću različitih programskih jezika i mogu se izvršavati u različitim tipovima procesora. Njihovi modeli podataka, predstavljanje informacija i protokoli komunikacije mogu biti potpuno različiti. Distribuiran sistem zahteva softver koji može da upravlja ove različite delove, i koji može da obezbedi da oni mogu da komuniciraju o da razmenjuju podatke. Termin "midlver" (middleware) se koristi za taj "posrednički softver", jer se nalazi u sredini, između između distribuiranih komponenti sistema.

Q: Navedite i objasnite nivoe slojevite arhitekture distribuiranih klijent-server sistema. Koji su problemi dvoslojne rahitekture?
A: Distribuirani sistemi kojima se pristupa preko Interneta, najčešće su organizovani kao klijent-server sistemi. To znači da korisnik komunicira sa softverom koji se izvršava na njegovom računaru (npr. veb pretraživač) a on dalje komunicira sa udaljenim računarom (npr. veb server) koji stoji na raspolaganju spoljnim klijentima. Udaljeni računar obezbeđuje servise, kao što je pristup veb stranama, namenjenim spoljnim klijentima. Kijent-server modeli su vrlo uopšteni arhitektonski modeli neke aplikacije. Oni se ne moraju primenjivati samo u slučaju distribuiranih sistema. Može server da bude na istom računaru kao i klijent. Uslučaju klijent-server arhitekture, aplikacije se moduluje kao skup servisa koje obezbeđuju serveri.

Q: Koja je razlika klijent-server arhitekture sa «debelim» i «tankim» klijentom? Navedite prednosti i nedostatke primene «tankih» a posebno, «debelih» klijenata?
A: 1. Model sa tankim klijentom (thin client)- klijent primenjuje prezentacioni nivo, a sve ostalo je na serveru (npr. veb pretraživač).
2. Model sa debelim klijentom (fat client) - deo ili cela aplikaciona obrada je na klijentu. Server upravlja podacima i funkcijama baze podataka.
Prednost modela sa tankim klijentom je u jednostavnosti upravljanja klijentima. To je od značaja kod sistema sa velikim brojem klijenata, jer je u takvim slučajevima problem instaliranja klijentskog softvera kod velikog broja korisnika. Sa tankim klijentom, koji koristi samo veb pretraživač, to nije problem jer nema nikakve dodatne instalacije (npr. veb aplikacije).

Nedostatak primene modela sa debelim klijentom je u tome što je cela obrada na serveru, te može da dođe do njegovog preopterećenja. Takođe, veće je opterećenje mreže, jer je veća interakcija klijenta sa serverom. Zato, primena modela sa tankim klijentom traži kvalitetne mreže, te i veće investicije u mreže.

Q: Koje su predmosti višeslojne arhitekture? A koji su njeni problemi?
A: ?

Q: Koja je korist od primene sistema sa distribuiranim komponentama?
A: Predmosti: odlaganje odluka za realizaciju servisa, otvorena arhitektura, fleksibilnost i proširljivost sistema i dinamičko rekonfigurisanje sistema.

Q: Koji su nedostaci sistema sa distribuiranim komponentama?
A: Složenije su za projektovanje a koriste nestandardizovani midlver. Zato ih u dosta slučajeva zamenjuje servisno-orijentisana arhitektura, mada imaju bolje performanse od nje.
1. Složenije su za projektovanje nego klijent-server sistemi. Višeslojna klijent-server sistemi su dosta intuitivni. Oni odražavaju mnoge ljudske transakcije kako ljudi traže i pružaju servise drugim ljudima. Nasuprot ovome, arhitekture sa distribuiranim komponentama su teže za razumevanje ljudi i za vizualizaciju.
2. Standardizovani midlver za sisteme sa distribuiranim komponenatama nije prihvaćen od korisnika. Umesto toga, pojedini proizvođači (Microsoft, Sun) su razvili različit i nekompatibilni. midlver. Midlver je složen sistem i oslanjanje na njih povećava ukupnu složenost sistema sa distribuiranim komponentama.

Q: Uporedite servisno-orijentisanu arhitekturu sa arhitekturom sistema sa distribuiranim komponentama. Koja ima bolje preformase? Zašto?
A: Zbog ovih problema, u mnogim slučajevima servisno-orijentisana arhitektura je zamenila arhitekturu sa distribuiranim komponentama. Međutim, sistemi sa distribuiranim komponentama imaju bolje performanse. RPC komunikacije su brže od kominikacije sa porukama koje se koriste u servisno-orijentisanim sistemima. Zato su arhitekture sa komponentama bolje za zahtevnije sisteme (traže veće brzine) .

Q: Na kojim principa rade sistemi ravnopravnih računara (peer-to-peer, or p2p)? Kako su njeniu čvorovi povezani? Kada se koristi p2p arhitektura?
A: Sistemi ravnopravnih računara su decentralizovani sistemi u kojima obrada podataka može da se vrši u bilo kom čvoru mreže, koji koriste istu kopiju aplikacije sa protokolima komunikacije. Sistemi ravnopravnih računara (peer-to-peer, or p2p) su decentralizovani sistemi u kojima obrada podataka može da se vrši u bilo kom čvoru mreže. U principu, nema nikakve razlike između između servera i klijenta. U aplikacjama ravnopravnih računara ceo sistem se

projektuje da bi se iskoristila računarska snaga i skladišta podataka koja su raspoloživa na potencijalno ogromnoj mreži računara. Standardi i protokoli koji omogućavaju komunikacije među čvorovima su ugrađene u samu aplikaciju i svaki čvor mora da izvršava kopiju te aplikacije. Svaki čvor p2p mreže mogao bi biti svestan prisustva svakog drugog čvora na mreži. Čvorovi mogu biti povezani i mogu da razmenjuju podatke direktno sa bilo kojim čvorom u mreži. U praksi, ipak, to nije moguće, već se čvorovi organizuju po lokacijama. Tada, neki čvorovi služe kao mostovi ka drugim čvorovima lokacije.

Q: Šta je polucentralizovana arhitektura sa ravnopravnim računarima? U čemu je njena prednost? Da li ona ima problem bezbednosti?

A: Decentralizovana arhitektura ravnopravnih računara (p2p) ima prednosti jer obezbeđuje redudantnost što obezbeđuje otpornost sistema na otkaze a i na isključenje pojedinih čvorova sa mreže. S druge strane, nedostaci su što više čvorova obrađuju isto pretraživanje i ima značajnih opštih troškova u repliciranju komunikacija među ravnopravnim računarima. Alternativni ahritektonski model, koji odstupa od čiste p2p arhitekture ima polucentralizovanu arhitekturu u kojoj, unutar mreže, jedan ili više čvorova služe kao serveri za olakšavanje komunikacija među čvorovima. Ovo smanjuje količinu prenosa podataka između čvorova. U polucentralizovanoj arhitekturi, uloga servera je da pomogne uspostavljanju kontakta između ravnopravnih računara u mreži, ili da koordiniše rezultate računjanja.

Q: Šta e «softver-kao-servis» (SaaS) ? Koji su ključni elementi SaaS koncepta?

A: Softver kao servis (software as a service, or SaaS) podrazumeva instalaciju softvera u nekom udaljenom serveru koji omogućava pristup klijentima preko Interneta. Ključni elementi SaaS koncepta su sledeći:

1. Softver je raspoređen na serveru (ili na više servera)
2. Softver je u vlasništvu provajdera softvera koji i njim upravlja
3. Korisnici plaćaju korišćenje softvera u skladu sa obimom njegovog korišćenja ili plaćanjem mesečne ili godišnje cene servisa. Ako je servis besplatan (npr. Youtube), onda korisnici moraju da prihvate reklame koje finansiraju servis. Primenom SaaS koncepta, korisnici su se oslobodili troškova upravljanja softvera (kupovina, instalacija, održavanje i dr.), jer je tu obavezu preuze provajder servisa. On je odgovoran i za otklanjanje grešaka u softveru i za instalaciju novih verzija, za njegovo prilagođavanje novim verzijama operativnog sistema i za osiguranje potrebnog kapaciteta hardvera. Ako neko ima više računara, nema potrebe da ima više licenci softvera. Ako se softver samo povremeno koristi, onda se najčešće primenjuje model "plaćanja premo korišćenju", Softveru se može pristupiti preko mobilnih uređaja, bilo odakle u svetu.

Q: Koji su problemi primene SaaS koncepta?
A: Naravno, i ovaj model korišćenja softvera ima neke nedostatke. Glavni problem je u troškovima transporta podataka ka udaljenom servisu. Transport se realizuje u skladu sa brzinom koju mreža može da obezbedi, što u slučaju velike količine podataka može da potraje dosta dugo. Servis provajder može formirati cenu u skladu sa količinom podataka koji od vas primi. Drugi problem je nedostatak kontrole evolucije softvera (jer provajder može da promeni softver kada poželi). Ostaju i problemi sa zakonima i propisima. Mnoge zemlje imaju zakone koji uređuju skladišćenje, upravljanje, prezentaciju i pristup podacima, te prebacivanje podataka u udaljene servise može da predstavlja kršenje ovih propisa.

Q: Koja je razlika SaaS i SOA?
A: SaaS je način isporuke funkcionalnosti aplikacije korisnicima, dok je SOA tehnologija implementacije aplikacionih sistema. Pojam SaaS i servisno-orijentisanu arhitekture (service-oriented architectures, ili SOAs) nisu isti:
1. SaaS je način obezbeđivanja funkcionalnosti na udaljenom serveru kome pristupaju klijenti preko Interneta uz korišćenje veb pretraživača. Server održava podatke korisnika i njihovo stanje za vreme interaktivne sesije. Transakcije su obično duge (npr. izmena nekog dokumenta).
2. SOA je pristup strukturisanom softverskom sistemu u vidu skupa posebnih servisa, koji nemaju svoje stanje (stateless). Ovo može da obezbeđuje veći broj provajdera i može da bude i distribuiran. Transakcije su obično kratke, i obuhvataju poziv servisa, negov rad i vraćanje rezultata korisniku.

Q: Na koje faktore morate d aobratite pažnjupri primeni SaaS?
A: Kada primenjujete SaaS, treba da znate da je moguće da su korisnici iz različitih organizacija, treba da uzmete u obzir sledeća tri faktora:
1. Konfugurabilnost: Da li ste konfigurisali softver u skladu sa specifičnim zahtevima svake od organizacije?
2. Višestruki zakup: Kako predstavite softversku uslugu korisnicima tako da oni oni dobiju utisak da rade sa svojom sopstvenom kopijom sistema, dok, u isto vreme, efikasno koriste resurse sistema.
3. Proširljivost: Kako projektujete sistem tako da on bude proširljiv da bi mogao da zadovolji nepredviđeno veliki broj korisnika?

Q: Šta je dinamičko konfigurisanje softvera? Šta dozvoljava dinamičko konfigurisanje?
A: Da bi se SaaS prilagodio specifičnim zahtevima korisnika, SaaS softver treba projektovati tako da se može dinamičke prefolofigurisati u sistem koji daje serviskoji korisnik očekuje. To se ostvaruje korišćenjem interfejsa konfiguracije koji omogućava korisnicima da specificiraju svoje prioritete, tj. ono što im više odgovara. Vi to onda upotrebljavate da bi dinamički podesili

ponašanje softvera u skladu sa očakivanjem korisnika. Ovako dinamičko konfigurisanje dozvoljava sledeće:
1. Brendiranje: Korisnici iz svake organizacije koriste interfejs koji odražava njihovu organizaciju (kompanijski logo i dr.).
2. Poslovna pravila i radni tokovi: Svaka organizacija može da definiše svoja pravila korišćenja servisa i njenih podataka.
3. Proširenja baze prodataka: Svaka organizacija definiše kako se opšti model podataka servisa proširuje da bi zadovoljio njihove specifične potrebe.
4. Kontrola pristupa: Kupci servisa kreiraju posebne račune za svoje zaposlene i definišu resurse i funkcije koje mogu da koriste njihovi zaposleni.
Višestruki zakup (multi-tenancy) je slučaj kada više različitih korisnika servisa pristupaju istom sistemu, a arhitektura sistema je tako definisana da dozvoljava efikasno deljenje resursa sistema. Pri tome, treba da korisnik ima utisak da je on jedini korisnik sistema. Da bi se to ostvarilo, sistem treba projektovati tako da se potpuno odvoje funkcionalnost sistema i podaci sistema, što znači da operacije sistema ne zavise od stanja podataka. Podatke obezbeđuju korisnici ili se uzimaju iz baze koja je dostupna sistemu. Relacione baze podataka nisu idealne za višestrukii zakup, te veliki provajderi (kao Gooogle) koriste jednostavnije sisteme baza.

Q: Šta je proširivost SaaS sistema? Koje su preoruke za realizaciju prišireња SaaS sistema?
A: Proširljivost je sposobnost sistema da radi sa povećanim brojem korisnika bez smanivenja ukupnog kvaliteta servisa (QoS) koji se pruža svakom korisniku, dodavanjem novih servera. Poseban problem sa sistemima koji nude višestruki zakup je upravljanje podacima. Najjednostavnije da se upravljanje podacime obezbedi nezavisno za svakog korisnika, te da on koristi svoju bazu podataka, tako da može da je konfiguriše kako želi. Međutim, to zahteva od provajdera da obezbedi puno tako definisanih specifičnih baza podataka. To nije efikasno sa stanovišta kapaciteta servisa i povećava ukupnu cenu servisa. Kao alternativa, servis provajder može da koristi bazu podataka za različite korisnike koji su virtualno izolovani unutar te baze. Slika 2 prikazuje takav slučaj . Korisnici baza podataka imaju svoj identifikator koji ih povezuje sa njihovim podacima u bazi. Upotrebom pogleda baze podataka (database views) mogu se izdvojiti samo podatke za servis potreban određenim korisnicima, te tako se stvaraju virtualne baze podataka koje u specifične za svakog korisnika. Ovim proširivanjem osnovne baze podataka provajdera, primenom specifične konfiguracije urađene za određenog korisnika, zadovoljavaju se njegove specifične potrebe. Kao alternativa, servis provajder može da koristi bazu podataka za različite korisnike koji su
virtualno izolovani unutar te baze. Slika 2 prikazuje takav slučaj . Korisnici baza podataka imaju svoj identifikator koji ih povezuje sa njihovim podacima u bazi. Upotrebom pogleda baze podataka (database views) mogu se izdvojiti samo podatke za servis potreban određenim korisnicima, te tako se stvaraju virtualne baze podataka koje u specifične za svakog korisnika.

Ovim proširivanjem osnovne baze podataka provajdera, primenom specifične konfiguracije urađene za određenog korisnika, zadovoljavaju se njegove specifične potrebe.

12 Servisno-orijentisano softversko inženjerstvo

Q: Šta je veb servis? Koji tip inyterfejsa koristi veb servis?
A: Veb servis se definiše kao: labavo povezana, ponovo upotrebljiva softverska komponenta koja sadrži diskretnu funkcionalnost, koja može biti distribuirana i programski pristupačna. Veb servis je servis kome se pristupa upotrebom standardnih Internet- i XML protokolima. Kritička razlika između servisa i softverske komponente je u tome što servisi moraju da budu nezavisni i labavo povezani. To znači da oni moraju uvek da rade na isti način, bez obzira na njihovo izvršno okruženje.

Q: Šta su servisno-orijentisani softverski sistemi?
A: Servisno-orijentisani sistemi su sistemi koji primenjuju ponovo upotrebljive softverske komponente kojima se pristupa preko drugih programa., a ne direktno povezivanjem korisnika na sam servis.

Q: Šta je koncept «softver kao servis»? Koja je korist od preimena ovog koncepta?
A: Primena servisno-orijentisanog softverskog inženjerstva nudi niz važnih koristi:
1. Servisi se mogu nuditi od strane bilo kog unutrašnjeg ili spoljnjeg provajdera. Aplikacija može da se razvije integracijom servisa različitih provajdera.
2. Servis provajder nudi javno informaciju o servisu tako da svaki ovlašćeni korisnik može da ga upotrebljava. Servis provajder i korisnik servisa ne moraju da pregovaraju o tome šta radi servis pre njegove integracije sa aplikacionim programom.
3. Aplikacije mogu da odlože povezivanje servisa sve do njihovog raspoređivanja ili do izvršenja. Aplikacija može dinamički da menja provajdera servisa i u vreme svog izvršenja. To znači da aplikacija može da reaguje na promene u svom izvršnom okruženju i da prilagođava tome svoj rad.
4. Moguće je oportunistička izrada novih servisa. Servis provajder može da kreira nove servise povezivanjem postojećih servisa na inovativne načine.
5. Korisnici servisa mogu da plate servis u skladu sa obimom njegove upotrebe, a ne preko provisije. Znači, umesto kupovine skupe komponente, koje se retko koriste, programer aplikacije koristi spoljne servise koji se plaćaju samo kada se traže i koriste.
6. Aplikacije sa servisima su male i pogodne za instaliranje na mobilnim uređajima sa ograničenim procesorskim i memorijskim kapacitetom. Spoljnim servisima se mogu prepustiti delovi programa koji zahtevaju veliku obradu podataka i rad sa izuzecima.
Servisno-orijentisani sistemi imaju labavo povezane (loosely coupled) arhitekture u kojime se veze među servisima menjaju za vreme izvršenja sistema. Neki sistemi mogu da koriste samo spoljne servise, a nekim mogu da kombinuju spoljne veb servise sa lokalno razvijenim komponentama.

SOA (service-oriented architecure) su labavo povezane arhitekture u kojima se veze servisa mogu menjati u toku izvršenja aplikacije. Neki sistemi se isključivo oslanjaju na primenu veb servisa, a drugi mešaju veb servise sa lokalno razvijenim komponentama.

Q: Kakva je arhitektura servisno-orijentisanih softverskih sistema? Ko su akteri u servisno-orijentisanim sistemima? Koja je njihova uloga?
A: Servisno-orijentisane arhitekture (SOA) su način razvoja distribuiranih sistema u kojima komponente sistema predstavljaju samostalne servise koji se izvršavaju na udaljenim računarima. U suštini, obezbeđenje nekog servisa je nezavisno od aplikacije koja upotrebljava taj servis. Provajderi servisa mogu da razviju specijalizovane servise i da ih ponude vrlo različitim korisnicima servisa iz različitih organizacija. Servisno-orijentisane arhitekture (SOA) su način razvoja distribuiranih sistema u kojima komponente sistema predstavljaju samostalne servise koji se izvršavaju na georgafski udaljenim računarima. XML-bazirani standardi, kao što su SOAP i WSDL su projektovani da bi podržali razvoj servisne komunikacije i razmenu informacija. Zbog toga, servisi su platforme koje su nezavisne od programskih jezika na kojima su razvijeni. Softverski sistemi se mogu konstruisati kombinovanjem lokalnih (sopstvenih) servisa i spoljnih servisa koje nude različiti provajderi, sa besprekornom interakcijom servisa u sistemu.

Q: Šta je XML? Yašta se on koristi?
A: Protokoli veb servisa pokrivaju sve aspekte SOA, od osnovnih mehanizama za servis razmene informacije (SOAP) do standarda za programski jezik (WS-BPEL). Svi ovi standardi koriste XML, jezik koji razume i čovek i računar, a koji definiše strukturu podataka, koristeći označen (tagovan) tekst, tj. koji koristi identifikator za određenim značenjem. XML koristi više podržavajućih tehnologija, kao što je XMD, za definiciju šeme podataka, a koji proširuje XML opise i omogućava manipulaciju sa njima.

Q: Šta je SOAP? Čemu služi?
A: SOAP: Ovo je standard za razmenu poruka koji podržava komunikaciju između servisa. On definiše neophodne i opcione komponente poruka koje razmenjuju servisi. Servisi u servisno-orijentisanim arhitekturama se ponekad nazivaju SOAP servisi.

Q: Šta je WSDL? Čemu služi?
A: WSDL: Web Service Dscription Language (WSDL) je standard za definisanje interfejsa servisa. On određuje kako se definišu servisne operacije (nazivi operacija, parametri, i njihovi tipovi) i veze servisa.

Q: Šta je WS-BPEL? Čemu služi?
A: WS-BPEL: Ovaj standard definiše jezik za specifikaciju radnog toka (workflow), tj. precesno orijentisane programe koji koriste nekoliko različitih servisa.

Q: Šta je UDDI? Čemu služi?
A: UDDI (Universal Description Discovery and Integration) standard pretraživanja definiše komponente specifikacije servisa koje pomažu potencijalnim korisnicima servisa da ga otkriju na

Internetu. Neke kompanije su koristili UDDI da objave svoje registratore. Međutim, sada se oni više ne koriste jer se sada koriste standardne mašine za pretraživanje (search engines) Interneta.

Q: Navedite ključne standarde sa veb SOA?
A: 1. WS-Reliability Messaging – standard za razmenu informacija koji obezbeđuje isproruku poruka, i to jednom, i samo jednom.
2. WS-Security – skup standarda koji podržavaju zaštitu veb servisa, koji uključuju standard koji specificira definiciju pravila za bezbednost i standarde koji pokrivaju upotrebu digitalnih potpisa.
3. WS-Addressing – koji definiše kako se predstavlja informacija u SOAP poruci.
4. WS-Transactions – koji definiše kako se koordinišu transakcije distribuiranih servisa.

Q: Koji su nedostaci veb servisa?
A: ?

Q: Do čega je dovale primena nove paradigme u softverskom inženjerstvu – primena servisno-orijentisanog sistema. Šta radi servis?
A: Servisno-orijentisan pristup u softverskom inženjerstvu je nova paradigma softverskog inženjerstva koja važna kao i objektno-orijentisano softversko inženjerstvo. Ova promena paradigme je sada ubrzana primenom inženjerstvom oblaka (cloud computing) u kome se servisi nude sa računarske infrastrukture instalirane kod provajdera, kao što su Google i Amazon. Ovo će imati stalni i duboki uticaj na sistemske proizvode i poslovne procese. Primena veb servisa omogućava da servisno –orijentisana preduzeća značajno poboljšaju agilnost kompanije, brzinu izbacivanja svojih novih proizvoda i servisa na tržište (time-to- market) i smanjivanje IT troškova i poboljšavanje efikasnosti rada.

Q: Šta je URI? Čemu služi?
A: Da bi koristili veb servis, morate da znate gde se on nalazi na Internetu, tj. da znate njegov URI (Uniform Resource Identifier) i detalje njegovog interfejsa. Ovi detalji se nalaze u opisu njegovog servisa napisanog u XML jeziku koji se naziva WSDL (Web Service Description Language).

Q: Šta je WSDL? Koja tri aspekta podržava WSDL?
A: WDSL specifikacija definiše tri aspekta veb servisa: šta servis radi, kako komunicira i i gde se nalazi:
1. Šta servis radi (what deo) u WSDL dokumentu je definisan u interfejsu, koji sadrži operacije koje servis podržava i definiše format poruke koja mu se šalje, odnosno, koju on može da primi.
2. Kako servis komunicira (how deo), u WSDL dokumentu se naziva "vezivanje" (binding) koji mapira (preslikava) apstraktni interfejs u konkretan skup protokola. Povezivanje specificir tehničke detalje kako se komunicira sa veb servisom.
3. Gde se servis nalazi (where deo) se nalazai u WSDL dokumentu koji opisuje lokaciju implementacije određenog veb servisa (njegove krajnje tačke).

Q: Koji su elementi WSDL modela?
A: 1. Uvodni deo koji obično definiše XML prostor imena koji se koristi i koji uključuje deo dokumenta sa dodatnim informacijama o servisu.
2. Opcioni opis tipova podataka koji se koriste u razmenjenim porukama servisa. 3. Opis interfejsa servisa, tj. operacija koje servis može da obezbedi drugim servisima ili korisnicima.
4. Opis ulaznih i izlaznih poruka koje servis obrađuje.
5. Opis vezivanja (binding) koji upotrebljava servis, tj. protokol poruka koji će biti korišćen pri slanju i primanju poruka. Početno podešena opcija je SOAP ali se mogu povezati i drugi protokoli. Povezivanje, koje u sadržaju poruke, određuje kako ulazne i izlazne poruke servisa bi trebalo da budu upakovane, kao i dodatne informacije, kao što su: sigurnosni akreditivi (credentials) ili identifikatori transakcija.
6. Specifikacija krajnje tačke koja predstavlja fizičku lokaciju servisa, izraženo sa URI adresom resursa kome se može pristupiti preko Interneta.

Q: Šta je resurs u konteksta servisno-orijentisanih sistema? Koje su osnovne operacije koje s emogu izvršiti nad jednim resursom? Koje četori akcije obezbeđuju HTTP i HTTPS?
A: Resurs predstavlja osnovni element RESTful arhitekture. Resurs je element podataka kao što je katalog ili medicinski karton, ili neki dokument i sl. Element može da se predstavlja na različite načine, tj. Da mogu biti u različitim formatima. Resursi imaju jedinstven identifikator, kao što je URL. Resursi su objekti koji upotrebaljavaju bitove, sa četiti osnovne polimorfističke operacije koje se pridodaju ovim objektima:
1. Kreiraj (Create) – dovodi resurs u postojanje.
2. Čitaj (Read) - vraća jedno predstavljanje resursa
3. Ažuriraj (Udate) – menja vrednost resursa
4. Obriši (Delete) – učini da resurs postave nepristupačan,

Veb protokoli HTTP i HTTPS se zasnivaju na četiri akcije: POST, GET, PUT i DELETE. Oni mapiraju osnovne operacije resursa, kao što je prikazano na slici 1.b.
1. POST kreira resurs. Koristi povezane podatke koje definišu resurs.
2. GET čita vrednost resursa i vraća je zahtevaocu servisa u obliku specifičnog prikaza, kao što je XHTML, koji može da se koristi veb pretraživačima (web browsers). 3. PUT menja vrednost resursa.
4. DELETE briše resurs.

Q: Šta je RESTful servis? Koji princip projektovanja on podržava?
A: Da bi zadovoljili očekivanje većine korisnika veb servisa, razvijen je jednostavniji standard za veb servise koji se bazira na REST arhitektonskom stilu, gde REST je skraćenica od Representational State Transfer. REST je je arhitektonski stil koji se bazira na prenosu predstavljanja resursa od servera do klijenta. Ovaj arhitektonski stil koristi mnogo jednostavniji metod za implementaciju intervejsa servisa od SOAP/WSDL.

Q: Koja je primena RESTful servisa u kod servisa «oblaka», tj. klaud servisa?
A: ?

Q: Koji su problemi pri primeni *RESTFul servisa?
A: 1. Kada neki servis ima složeni interfejs i kada on nije jednostavan resurs, može biti teško da se projektuje set (skup) RESTful servisa koji predstavljaju taj interfejs.
2. Ne postoji standard za opis RESTful interfejsa, te korisnik interfejsa mora da koristi samo neformalnu dokumentaciju radi rezumevanja interfejsa.
3. Kada koristite RESTful servise, morate upotrebiti sopstvenu infrastrukturu i upravljanje kvalitetom servisa i pouzdanošću servisa. U slučaju primene SOAP servis postoje standardi za dodatnu infrastrukturu kao što su WS-Reliability i WS- Transaction.

Q: Šta je inženjerstvo servisa? Koje zahteve opn mora da podrži?
A: Inženjerstvo servisa je proces razvoja servisa za korišćenje od strane servisno-orijentisanih aplikacija. Servis mora da bude ponovno upotrebljiva apstrakcija koja može da bude od koristi u različitim sistemima. Projektanti servisa moraju da projektuju i razviju funkcionalnost od šireg značaja koji se povezuju sa apstrakcijom, pri čemu treba da obezbede da servis bude robustan i pouzadan. Moraju da dokumetuju servis tako da može da bude otkriven od strane potencijalnih korisnika, i njima razumljiv.

Q: Koje su tri faze inženjering servisa?
A: 1. Utvrđivanje mogućih servisa, kada utvrđujete moguće servise koje bi mogli da iskoristite i kada definišete zahteve za servis.
2. Projektovanje servisa, kada projektujete logičke i WSDL interfejse servisa.
3. Implementacija i raspoređivanje servisa, kada razvijete i testirate programski kod komponeneta koje obezbeđuju servise i kada ga činite dostupnim korisnicima.

Q: Koja su tri osnovna tipa mogućih servisa? Objasnite ih.
A: 1. Uslužni servisi (utility services): Ovi servisi primenjuju neku opštu funkcionalnost koja se može koristiti u različitim poslovnim procesima. Primer uslužnog servisa je servis za konverziju valuta (npr. evra u dolare).
2. Osnovni servisi: Ovi servisi su povezani sa specifičnim poslovnim funkcijama. Na primer, registracija studenata za neku izborni predmet.
3. Servisi koordinacije ili procesa: Ovo su servisi koji podržavaju uopšteniji poslovni proces u kome obično učestvuje više aktera i aktivnosti. Na primer, sistem za podršku nabavki (roba, usluga, nalaženje isporučioca, plaćanje).

Q: Koja je razlika servisa za zadatke od servisa za entitete?
A: 1. Servisi za zadatke: Povezani su sa nekom aktivnošću.
2. Servisis za entitete: Povezani su sa nekim poslovnim entitetom

Q: Kako bi birali servis preko Interneta? Na koja pitanja bi obratili pažnu?
A: ?

Q: Koje su faze projektovanja interfejsa servisa? Navedite operacije koje taj interfejs treba da podrži.
A: 1. U slučaju servisa entiteta, da li je servis (usluga) povezana sa jednoim logičkim entitetom koji se koristi u više poslovnih procesa? Koje se operacije obično sprovode nad navedenim entitetom, koji se treba podržati servisom?
2. Za slučaj servisa za zadatke, da li zadatak izvršava više ljudi u organizaciji? Da li su oni spremni da prihvate neophodnu standardizaciju koju uvodi servis koji treba da podržava ovaj zadatak?
3.Da li je servis nezavisan, tj. u kojoj meri zavisi od drugih servisa
4. Da li pri svom radu servis mora da održava svoje stanje? Servisi su bez stanja, što znači da oni ne održavaju svoje stanje. Ako se traži informacija o stanju, onda se mora koristiti baza podataka, a to može da ograniči ponovnu upotrebljivost servisa.
5. Da li servis mogu da koriste klijenti van organizacije? Na primer, servis koji obezbeđuje katalog proizvoda, može da ima i unutrašnje i spoljnje korisnike.
6. Da li se može desiti da različiti korisnici servisa imaju različite nefunkcionalne zahteve. Ako imaju, da li to znači da servis mora da nudi loše verzija servisa?

????

13 Projektovanje softvera u realnom vremenu

Q: Šta su sistemi u realnom vremenu? U čemu je razlika ovih sistema u odnosi na ostale sofverske sisteme? Šta su “meki” i “tvrdi” sistemi u realnom vremenu?
A: Računari se danas široko koriste za upravljanje proizvodnim mašinama, robotima, avionima, automobilima i mnogim drugim uređajima. To su sistemi za upravljanje raznim fizičkim sistemima. U njima, računari su u interakciji sa hardverskim uređajima. Softver računara mora da reaguje na događaje generisane od strane hardvera i reaguju na te događaje tako što proizvode upravljačke signale. Ovi signali učestvuju u pokretanju određenih akcija. Softver u ovim sistemima je ugrađen u hardver sistema, često u memoriju samo za čitanje (ROM, read-only memory). Softver reaguje, u realnom vremenu, na događaje koji potiču iz okruženja sistema. Ukoliko je vreme reagovanje duže od datih rokova, onda sistem ne radi kako treba. Kao što se vidi, vreme je je findamentalan element softverskih sistema u realnm vremenu, i fundament njihove definicije:
Softverski sistem u realnom vremenu (real-time software system) je sistem čiji ispravan rad zavisi i od rezultata koji proizvodi i od vremena u kome se ti rezultati proizvedu. Pod "mekim" sistemima u realnom vremenu: podrazumevaju se sistemi koji neispravno rade ako se rezultati ne proizvode u skladu sa specificiranim vremenskim zahtevima. Pod "tvrdim" sistemima u realnom vremenu; se podrazumevaju sistemi koji ne proizvode rezultate u skladu sa specifikacijom vremena reagovanja, i u tom slučaju, sistem je neuspešan;.
Vreme reagovanja ugrađenih sistema u realnom vremenu je vrlo važan faktor. Međutim, to ne znači da svi ugrađeni sistem zahtevaju vrlo brz odgovor, tj. vrlo brzo reagovanje.

Q: Šta je podsticaj, a šta je ponašanje sistema u realnom vremenu? Koje vrste podsticaje postoje? Objsnite ih.
A: Najopštiji pristup u projektovanju ugrađenih sistema u realnom vremenu se zasniva na modelu podsticaja (stimulus) i odgovora (response). Podsticaj je događaj koji se javlja u okruženju softverskog sistem koji prouzrukuje da sistem reaguje na neki način – a to je signal ili poruka koje sistem šalje u svoje okruženje.
Definisanje ponašanja sistema u realnom vremenu se može definisati listom podsticaja koje sistem prima, odgovarajućih odgovora sistema, i sa vremenom potrebnim da se proizvede odgovor sistema.

Q: Koja je funkcija senzora, a koja- pokretača (actuator)?
A: Podsticaji (stimuli) dolaze od senzora u okruženju sistema (slika 2), a odgovori se šalju pokretačima (actuators). I pokretači mogu da generišu podsticaje.Pokretači obično označavaju javljanje nekog problema u radu pokretača, a koji sistem mora da reši.

Q: Navedite aktivnosti procesa projektovanja sistema u realnom vremenu.
A: Mogu se koristiti sledeće aktivnosti procesa projektovanja sistema u realnom vremenu:
1. Izbor platforme: Birate izvršnu platformu koju čine hardver i operativni sistem u realnom vremenu. Faktori za izbor: vremenska ograničenja, ograničenja napajanja, iskustvo tima za razvoj i ciljna cena sistema koji se razvija.
2. Identifikacija podsticaja/odgovora: Utvrđuju se podsticaji koje sistem treba da obra]uje i odgovori na svaki od njih.
3. Analiza vremena: Za svaki podsticaj i odgovor određujete vremensko ograničenje koje se odnosi i na podsticaj i na obradu odgovora. Na ovaj način se postavljaju rokovi za izvršenje procesa sistema.
4. Projektovanje procesa: Projektovanje procesa predstavlja objedinjavanje podsticaja i obradu odgovora u određeni broj konkuretnih procesa. Projektovanje počinjete izborm odgovarajućeg šablona projektovanja, a onda optimizujete arhitekturu procesa u skladu sa specifičnim zahtevima sistema koji projektujete.
5. Projektovanje algoritma: Za svaki podsticaj i odgovor, projektujete odgovarajući algoritam za realizaciju potrebnih obračuna. Određivanje algoritama se radi u ranoj fazi procesa projektovanja kako bi se odredio obim obrade podataka i potrebno vreme obrade.
6. Projektovanje podataka: Određujete informacije koje se razmenjuju između procesa i događaje koji koordinišu ovu razmenu informacija. Projektujete i strukturu podataka za upravljanje ovom razmenom informacija. Nekoliko konkurentnih procesa obično deli ove strukture podataka.
7. Planiranje procesa: Potrebno je da projektujete sistem planiranja koji će obezbediti da procesi počinju na vreme i da ostvaruju definisane rokove za završetak svog posla.

Q: Zašto je potrebna sinhronizacija procesa? Šta je cilj sinhronizacije? Šta je kružni bafer? Ćemu služe Get i Put operacije?
A: U radu sistema u realnom vremenu više procesa paralelno radi i deli pojedine resurse, vrlo je važno obezbediti adekvatnu koordinaciju njihovog rada.