OSNOVI SOFTVERSKOG INŽENJERSTVA

Odgovori na pitanja sa rokova 2021. i 2022. godine + dodatna pitanja

Marko Dunović

1. SCRUM softverski procesni model.

Scrum spada u najvažnije agilne metode. Dakle, to je agilna metoda koja se fokusira na upravljanje iterativnim razvojem, a ne specifične agilne prakse. U SCRUM-u postoje tri faze:

- Početna faza je faza planiranja u kojoj se uspostavljaju opšti ciljevi projekta i dizajnira softverska arhitektura.
- Nakon toga slijedi serija sprint ciklusa, gdje svaki ciklus razvija inkrement sistema.
- Faza zatvaranja projekta završava projekat, kompletira potrebnu dokumentaciju i procjenjuje naučene lekcije iz projekta.

SCRUM terminologija, pojmovi i definicije.

Razvojni tim – samoorganizujuća grupa softver developera koju ne bi trebalo činiti više od 7 ljudi. Oni su odgovorni za razvoj softvera i drugih bitnih projektnih dokumenata.

Product backlog – lista „to do“ stvari sa kojima se scrum tim mora pozabaviti. To mogu biti definicije funkcija za softver, zahtjevi, korisničke priče, opisi dodatnih zadataka itd.

Product owner – pojedinac ili mala grupa čiji je posao da identifikuje karakteristike ili zahtjeve proizvoda, odredi ih po prioritetima za razvoj i kontinuirano pregledava zaostatak proizvoda kako bi osigurali da projekat nastavlja da ispunjava kritične poslovne potrebe. To može biti kupac, ali može biti i menadžer proizvoda u softverskoj kompaniji ili neki drugi predstavnik stakeholdera (zainteresovanih strana).

Scrum – dnevni sastanak scrum tima koji razmatra napredak i daje prioritete onim poslovima koje treba uraditi taj dan. U idealnom slučaju ovo bi trebalo da bude kratak sastank, licem u lice, gdje učestvuje cijeli tim.

Scrum master – odgovoran je za praćenje procesa i vođenje tima u efikasnom korištenju scrum-a. Odgovoran je za povezivanje sa ostatkom kompanije i spriječava ometanje tima od vanjskih uticaja. Scrum developeri smatraju da se scrum master ne bi trebalo smatrati menadžerom projekta.

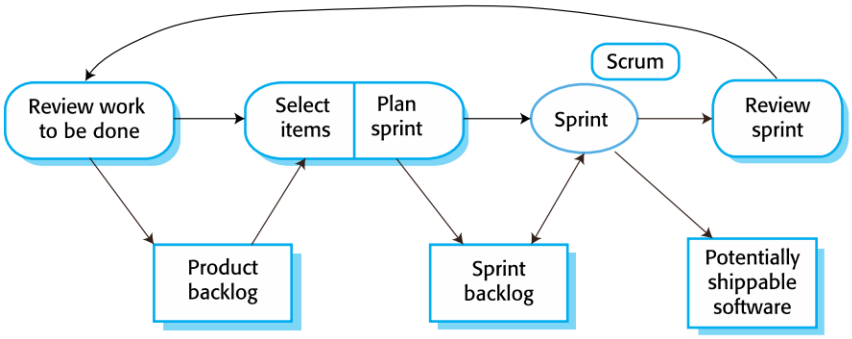
Sprint – razvojna iteracija, obično traje 2-4 sedmice.

Brzina – procjena koliko product backlog-a tim može uraditi u jednom sprintu. Razumijevanje brzine tima pomaže da se procijeni koliko posla se može uraditi u jednom sprintu.

Scrum sprint ciklus

Sprintovi su fiksne dužine, obično 2-4 sedmice. Polazna tačka za planiranje je product backlog koji predstavlja spisak poslova koje treba obaviti u projektu. Faza odabira uključuje cijeli razvojni tim i kupca za odabir karakteristika i funkcionalnosti iz product backloga koji će se razviti tokom sprinta. Kada se ovo dogovori, tim se organizuje za razvoj softvera. Tokom ove faze, tim je izolovan od korisnika i oragnizacije sa svim komunikacijama koje idu preko SM-a. Uloga scrum mastera je da zaštiti tim od vanjskih smetnji.

Na kraju sprinta, vrši se pregled urađenog i predstavlja stackholderima nakon čega počinje sljedeći ciklus sprinta.



Timski rad u Scrumu

Scrum master je osoba koja oragnizuje dnevne scrum sastanke, prati poslove koje treba uraditi, bilježi odluke, mjeri napredak u odnosu na zaostatak i komunicira sa klijentima i menadžmentom van samog tima. Cijeli tim prisustvuje dnevnim kratkim sastancima (Scrums) na kojima svi članovi tima dijele informacije, opisuju svoj napredak od posljednjeg sastanka i probleme koji su se pojavili i šta je planirano za sljedeći dan.

Prednosti Scrum-a

- Proizvod je podijeljen na skup upravljivih i razumljivih dijelova.
- Nestabilni zahtjevi ne usporavaju napredak.
- Cijeli tim ima uvid u sve što se dešava, samim tim se poboljšava komunikacija unutar tima.
- Kupci vide isporuku inkrementa na vrijeme i dobijaju povratne informacije o tome kako proizvod funkcioniše.
- Uspostavlja se povjerenje između kupaca i programera.

2. Nefunkcionalni zathjevi.

Nefunkcionalni zahtjevi

- Ograničenja na usluge ili funkcije koje nudi sistem kao što su vremenska ograničenja, ograničenja u procesu razvoja, standardi itd.
- Često se primjenjuje na sistem u cjelini, a ne na pojedinačne usluge ili karakteristike.

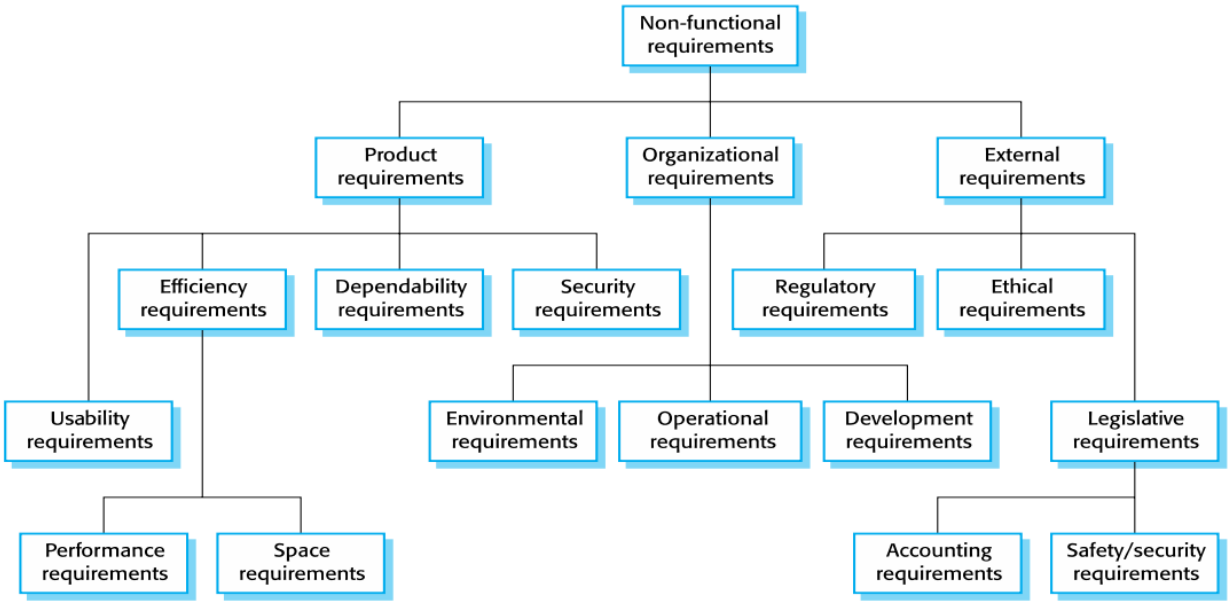
Kako funkcionalni tako i nefunkcionalni zahtjevi moraju biti kompletni i dosljedni. Pod kompletnim se smatra da bi trebali sadržavati opis svih potrebnih objekata. Pod dosljednim se smatra da ne bi trebalo biti konflikata i kontradikcija u opisu sistemskih objekata. U praksi, zbog složenosti sistema i životne sredine, nemoguće je da se izrade kompletni i dosljedni dokumenti zahtjeva.

Klasifikacija nefunkcionalnih zahtjeva

Klasifikacija:

- NFZ proizvoda – zahtjevi koji preciziraju da se isporučeni proizvod mora ponašati na određeni način, npr. brzina izvršavanja, pouzdanost

- Organizacioni NFZ – zahtjevi koji su posljedica organizacionih polisa i procedura, npr. zahtjevi za implementaciju
- Eksterni nefunkcionalni zahtjevi – zahtjevi koji proizilaze iz spoljnih faktora i procesa njegovog razvoja, npr. interoperabilnost – sposobnost heterogenih sistema da rade zajedno, zakonodavstvo



Implementacija NFZ

Nefunkcionalni zahtjevi mogu uticati na sveobuhvatnu arhitekturu sistema, prije nego na pojedinačne komponente. Jedan nefunkcionalni zahtjev, poput bezbjedonosnog zahtjeva, može generisati mnoge povezane funkcionalne zahtjeve koji definišu sistemske servise koji su potrebni. Takođe može da generiše zahtjeve koji ograničavaju već postojeće zahtjeve. Nefunkcionalne zahtjeve je veoma teško precizno navesti i neprecizne zahtjeve je teško provjeriti. **Cilj** je da se ispuni **opšta namjera korisnika** kao što je jednostavnost korištenja. Ciljevi su korisni developerima jer prenose namjere korisnika sistema.

Mjere za speficikaciju NFZ:

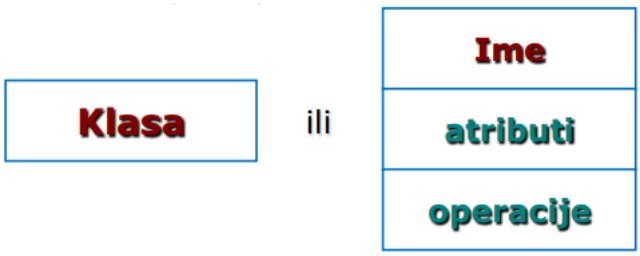
- Brzina – obrađene transakcije po sekundi, vrijeme odgovora korisnika/događaja, vrijeme osvježavanja ekrana.
- Veličina – MB, broj ROM čipova
- Lakoća korištenja – vrijeme uvježbavanja, broj okvira za pomoć
- Pouzdanost – srednje vrijeme do pada, vjerovatnoća nedostupnosti, stopa pojava padova, dostupnost
- Robusnost – vrijeme potrebno za pokretanje nakon pada, procenat događaja koji izazivaju pad, vjerovatnoća korupcije podataka prilikom pada
- Prenosivost – procenat iskaza zavisnih od cilja, broj ciljnih sistema.

3. Dijagram klasa.

Strukturni UML dijagram – modelovanje statičke strukture softverskog sistema. Koristi se za različite namjene i u različitim fazama razvoja softverskog sistema. Omogućava prikaz klasa i njihovih međusobnih veza, bez mogućnosti modelovanja vremenski zavisnog ponašanja sistema.

Klasa je deskriptor za skup entiteta sa sličnom strukturom, ponašanjem i vezama sa drugim entitetima. (entiet – stvari, pojmovi, itd.)

Klasa se na dijagramu prikazuje u obliku trodjelnog pravougaonika, gdje gornja sekcija sadrži naziv klase, srednja atribute, a donja operacije. Kad detalji nisu bitni, atributi i operacije ne moraju da se prikazuju. Naziv apstraktne klase se prikazuje kurzivom (italik).



Klase – specifikacije atributa i operacija.

- Specifikacija atributa je oblika: vidljivost naziv : tip [M] = vrijednost {ograničenje},

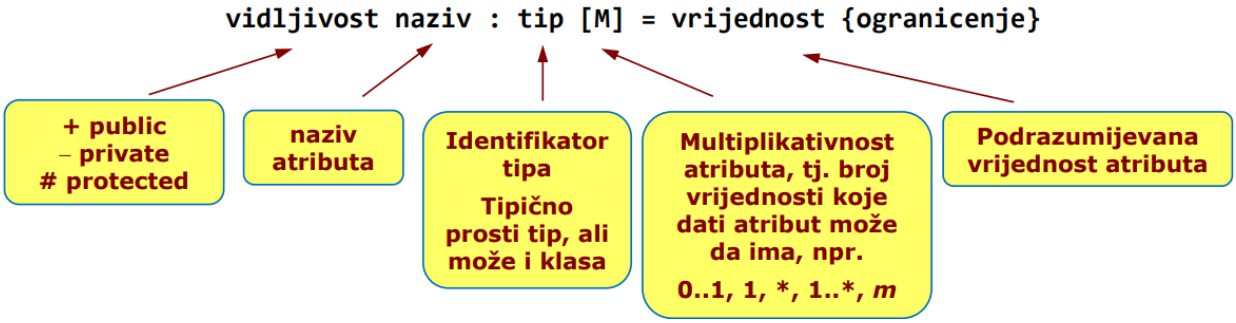
vidljivost – + public, - private, # protected

naziv – naziv atributa

tip – identifikator tipa, tipično prosti tip ali može i klasa

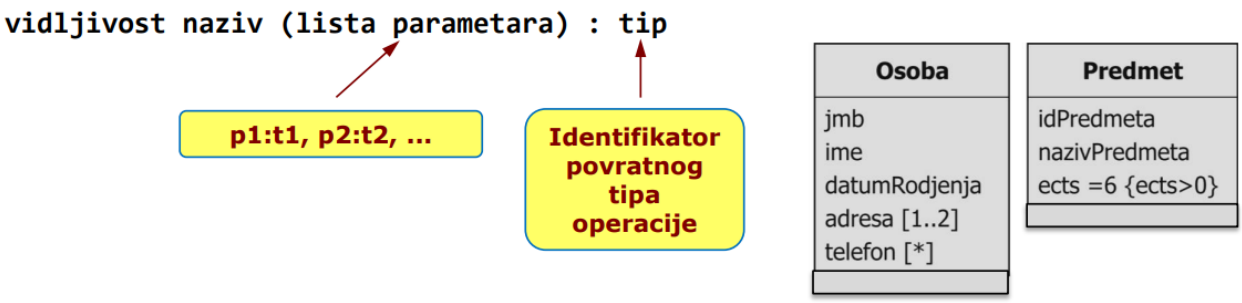
M – multiplikativnost atributa

vrijednost – podrazumijevana vrijednost atributa



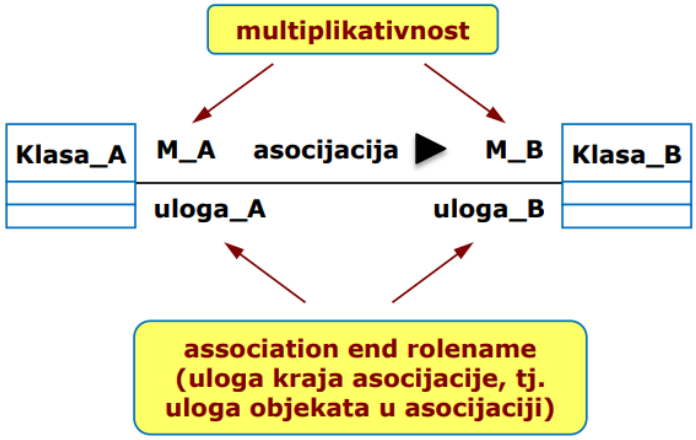
- Specifikacije operacija je oblika: vidljivost naziv (lista parametara) : tip

tip – identifikator povratnog tipa operacije

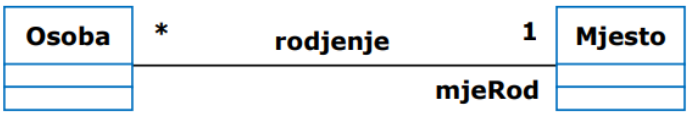


Veze između klasa

- Asocijacija – najopštiji tip veze između klasa koji reprezentuje trajne veze između objekata. Naziv asocijacije oslikava vrstu veze objekata, najbolje jedna riječ, npr. „prebivalište“ između objekata klasa Osoba i Mjesto, ili „prijavljuje“ između Student i Ispit. Trajna veza je veza između objekata koja reprezenzuje svojstvo objekta, npr. osoba je rođena u nekom mjestu, profesor predaje predmet, student je upisao studije itd. Trougao ukazuje na smjer čitanja asocijacije, smjer nije obavezan. Multiplikativnost pokazuje koliko objekata date klase učestvuje u vezi sa jednim objektom druge klase, npr. M_A pokazuje broj objekata klase A koji su u vezi sa jednim objektom klase B.



Primjer:

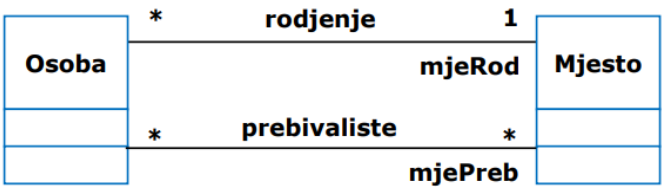


U jednom mjestu može biti rođeno više osoba, pa je zato multiplikativnost objekata klase Osoba jednaka *. Svaka osoba može biti samo rođena u jednom mjestu pa je multiplikativnost objekata klase Mjesto jednaka 1.

simbol	značenje
0	nijedan
1	jedan
m	neki cijeli broj
0..1	nijedan ili jedan
m..n	najmanje *m*, a najviše *n*
*	više (proizvoljno)
0..*	nijedan ili više
1..*	jedan ili više

Između dvije klase može da postoji više asocijacija.

Primjer:



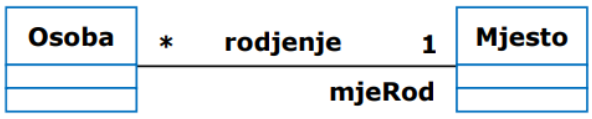
U jednom mjestu prebivalište može da ima više osoba pa je multiplikativnost objekata klase Osoba jednaka *, a svaka osoba tokom vremana može da ima više prebivališta pa je multiplikativnost na strani Mjesto *.

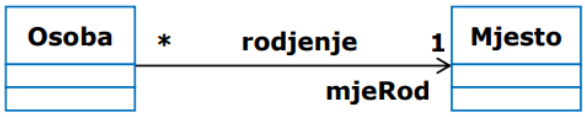
REFLEKSIVNA ASOCIJACIJA – asocijacija između objekata iste klase.

Primjer:

Asocijacije mogu biti:

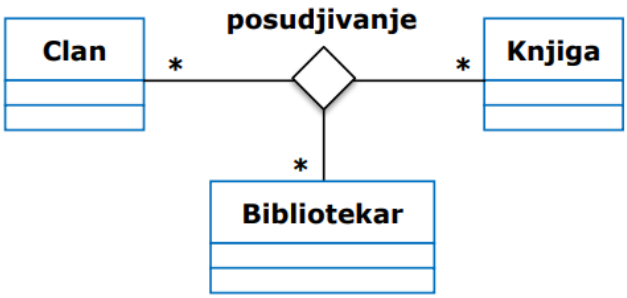
- Neusmjerene – bidirekcione
- Usmjerene – unidirekcione





Ako je asocijacija usmjerena, tada su samo objekti jedne klase "svjesni" objekata druge klase, npr. svaka osoba ima atribut mjeRod koji reprezentuje mjesto rodjenja date osobe, ali objekti klase Mjesto nemaju niz objekata sa podacima o osobama koje su rođene u datom mjestu.

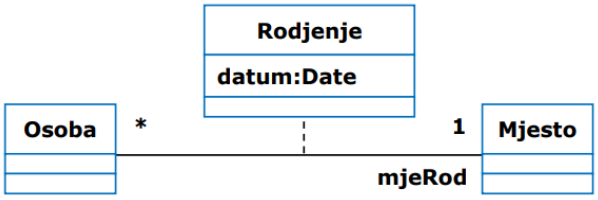
N-arna asocijacija – asocijacija koja ima više od dva kraja.



N-arna asocijacija „posuđivanje" reprezentuje činjenicu da je neki član posudio neku knjigu, pri čemu je neki bibliotekar evidentirao to posuđivanje. Svaki član može da ima više posuđivanja (*na strani klase Clan), svaka knjiga može biti više puta posuđena (* na strani klase Knjiga), a svaki bibliotekar može da evidentira više posuđivanja (* na strani klase Bibliotekar).

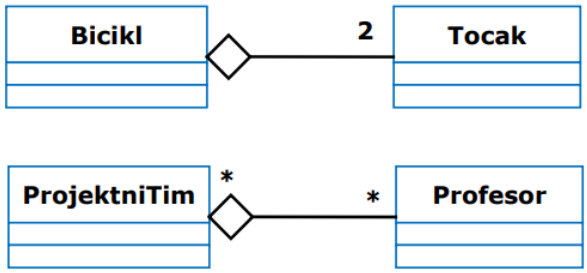
Vezna (pridružena) klasa

Asocijacija koja ima atribute reprezentuje se klasom koja je pridružena toj asocijaciji.



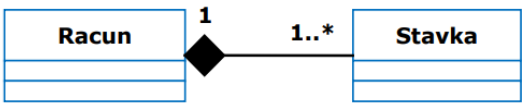
Agregacija

Asocijacija koja predstavlja odnos cjelina – dio. Dijelovi su entiteti koji mogu da imaju vlastiti identitet, tj. mogu da egzistiraju nezavisno od cjeline. Prikazuje se kao asocijacija koja na strani cjeline završava praznim rombom.



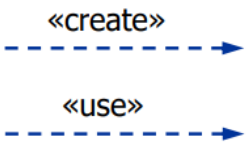
Kompozicija

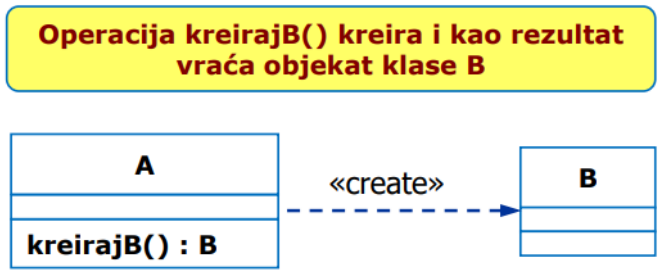
Snažniji (stroži) oblik asocijacije u odnosu na agregaciju. Dijelovi kompozicije ne mogu da egzistiraju nezavisno od cjeline. Prikazuju se kao asocijacija koja na strani cjeline završava obojenim rombom.

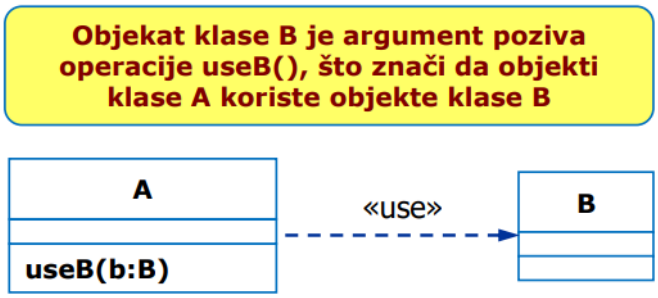


Zavisnost

Osim trajnih veza između objekata, između objekata postoje i privremene veze. Privremena veza između objekata reprezentuje se kao zavisnost između korespodentnih klasa. Privremene veze su veze kreiranja, korištenja, uništavanja, itd.

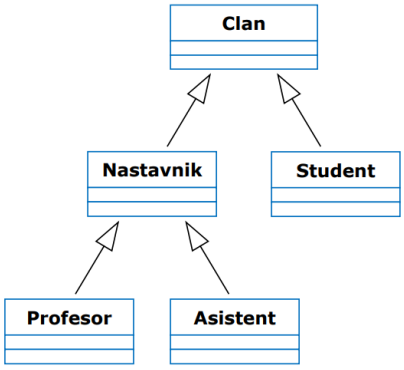






Generalizacija

Veza između uopštenijeg opisa nekog entitetskog skupa i specifičnijeg opisa, na bazi uopštenog, kojim se uopšteni opis proširuje, tj. specijalizuje. Specijalizovani opis uključuje sve karakteristike definisane opštijim opisom uz mogućnost definisanja dodatnih. Generalizovana specifikacija – superklasa, natklasa. Specijalizovana specifikacija – subklasa, potklasa.



Ne postoji jedinstven i u potpunosti definisan način za kreiranje dijagrama klasa. Dijagram klasa se kreira ručno. Tekstualna specifikacija toka događaja u slučaju upotrebe je dobar osnov za kreiranje dijagrama klasa.

4. Specifikacija slučaja upotrebe.

Slučaj upotrebe je apstrakcija koja opisuje jednu klasu scenarija upotrebe sistema.

Redoslijed koraka pri specifikaciji slučaja upotrebe:

1. Jedinstveno identifikovati slučaj upotrebe, npr. SU-1
2. Imenovati slučaj upotrebe, najčešće glagolska imenica + objekat, npr. podizanje gotovine
3. Identifikovati učesnike, npr. Student, Bibliotekar
4. Identifikovati tok događaja – koristiti neformalni (prirodni) jezik i/ili jednostavne dijagrame aktivnosti, treba koristiti aktivne fraze, svi koraci treba da su jedinstveno specifikovani, a redoslijed jasan, osim glavnog toka, treba specifikovati i alternativne tokove
5. Identifikovati početne uslove
6. Identifikovati izlazne kriterijume
7. Identifikovati izuzetke
8. Idenfitikovati nefunkcionalne zahtjeve

Šabloni za specifikaciju slučaja upotrebe

Specifikacija slučaja upotrebe nije standardizovana.

Oznaka		
Naziv		
Učesnici		
Kratak opis		
Preduslovi		
Osnovni tok događaja	Učesnik	Sistem
Postuslovi		
Alternativni tokovi		
Nefunkcionalni zahtjevi		
Skica korisničkog interfejsa	Dijagram aktivnosti	Dijagram sekvence

5. Review kao tehnika za osiguranje kvaliteta softvera.

Review je tehnika za osiguravanje kvaliteta softvera koja se temelji na čitanju i pregledavanju koda, specifikacija i drugih dokumenata vezanih za softver. Cilj review-a je identifikovati probleme, nedostatke i potencijalne izmjene u softveru prije nego što se postavi u proizvodnju. Review se obično obavlja u nekoliko koraka:

- Priprema – priprema dokumenata i materijala za review, kao što su kod, speficikacije, dizajn i dokumentacija
- Pregled – pregled koda i drugih dokumenata od strane jednog ili više review-era. Review-eri se fokusiraju na kvalitet, sigurnost, performanse i druge važne kriterijume.
- Razmjena mišljenja – review-eri diskutuju o svom pregledu i iznose svoje mišljenje o identifikovanim problemima kao i potencijalnim izmjenama
- Izvještaj – Review-eri zapisuju stavke za popravku i izmjene koje su identifikovati tokom reviewa.

Review može pomoći u povećanju kvaliteta i sigurnosti softvera, kao i smanjenju vremena i troškova tokom otklanjanja problema u kasnijim fazama razvoja. Takođe pomaže u podsticanju timskog rada, razmjeni znanja i unapređenju procesa razvoja softvera.

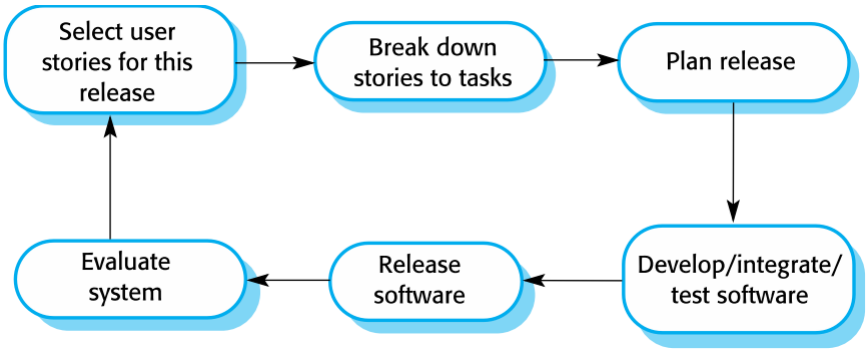
6. XP agilni metod.

Ekstremno programiranje (XP – extreme programming) je veoma uticajna agiilna metoda koja je uvela niz tehnika agilnog razvoja. XP koristi „ekstremni“ pristup iterativnom razvoju.

- Nove verzije se mogu praviti nekoliko puta dnevno
- Inkrementi se isporučuju kupcima svake dvije nedjelje

- Svi testovi se moraju pokrenuti za svaku izgradnju
- Izgradnja se prihvata samo ako se testovi uspješno pokreću.

XP ciklus izdavanja:



Prakse ekstremnog programiranja:

- Inkrementalno planiranje – Zahtjevi se bilježe na karticama priča, koje priče će biti uključene u izdanje zavisi od njihovog prioriteta i raspoloživog vremena. Programeri razbijaju priče u adekvatne zadatke.
- Mala izdanja – Najprije se razvija minimalni korisni skup funkcionalnosti koji pruža poslovnu vrijednost. Izdanja sistema su česta i postepeno se dodaje funkcionalnost prvom izdanju.
- Jednostavan dizajn – Izrađeno je dovoljno dizajna da zadovolji trenutne zahtjeve i ništa više.
- Test-first razvoj – Okvir za automatizovani jedinični test se koristi za pisanje testova za novi dio funkcionalnosti prije nego što ta funkcionalnost bude sama implementirana.
- Refaktoring – Od svih programera se očekuje da redovno refaktorišu kod čim primjete da postoji potreba za tako nešto. Ovo čini kod jednostavnim i održivim.
- Programiranje u paru – Programeri rade u paraovima, provjeravaju rad jedni drugih i pružaju podršku da posao uvijek bude dobro urađen
- Kolektivno vlasništvo – Svi programeri preuzimaju odgovornost za cijeli kod, svako može promjeniti bilo šta.
- Stalna integracija – Čim se neki zadatak završi, odma se integriše u cijeli sistem. Nakon svake integracije, svi jedinični testovi moraju biti uspješni.
- Održiv tempo – Velike količine prekovremenog rada se ne smatraju prihvatljivim.
- Kupac na licu mjesta – Kupac treba da bude prisutan puno radno vrijeme u korist XP tima, član je razvojnog tima.

XP i agilni principi

Postepen razvoj je podržan kroz mala, česta izdanja sistema. Uključenost kupca znači puno radno vrijeme klijenta sa timom. Ljudi, a ne proces – kroz programiranje u paru, kolektivno vlasništvo, dobija se proces koji izbjegava dugo radno vrijeme. Promjena je podržana kroz duga izdanja sistema. Održavanje jednostavnosti kroz stalno refaktorisanje koda.

Uticajne XP prakse

XP ima tehnički fokus i nije ga lako integrisati u većini organizacija. Prema tome, dok agilni razvoj koristi prakse iz XP-a, metod kako je prvobitno definisan nije široko prihvaćen. Ključne prakse su: korisnićke priče za specifikaciju, refaktorisanje, test – prvi razvoj, programiranje u paru.

7. Arhitektonski projektni stilovi.

Arhitektonski projektni stilovi (obrasci). Obrasci su sredstvo za predstavljanje, dijeljenje i ponovnu upotrebu znanja. Arhitektonski obrazac je stilizovan opis dobre dizajnerske prakse koja je isprobana i testirana u različitim okruženjima. Obrasci treba da sadrže informacije o tome kada jesu i kada nisu korisni. Obrasci se mogu predstaviti korišćenjem tabelarnih i grafičkih opisa. Neki važniji arhitektonski obrasci su: Klijent – Server, MVC, slojeviti, repozitorijum, cijev i filter.

Klijent – Server

Model distribuiranog sistema koji pokazuje kako se podaci i obrada distribuiraju kroz niz komponenti (može se implementirati na jednom računaru). To je skup samostalnih servera koji pružaju specifične usluge kao što su štampanje, upravljanje podacima itd. To je skup klijenata koji se obraćaju ovim uslugama. Mreža omogućava klijentima pristup serverima. U arhitekturi klijent – server funkcionalnost servera je organizovana u usluge, pri čemu se svaka usluga isporučuje sa posebnog servera. Klijenti su korisnici ovih usluga i pristupaju serverima da bi ih koristili.

Kada se koristi?

- Kada se podacima u dijeljenoj bazi podataka mora pristupiti sa više lokacija
- Pošto serveri mogu da se repliciraju, koristi se takođe kada je opterećenje na sistemu promjenljivo

Prednosti?

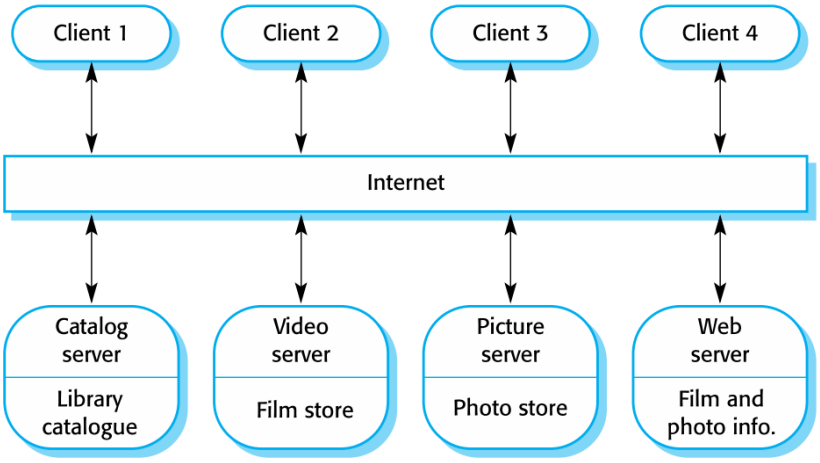
- Serveri se mogu distribuirati preko mreže
- Opšte funkcionalnosti mogu biti dostupne svim klijentima, i ne moraju biti implementirane od strane svih servisa

Mane?

- Svaka usluga je jedna tačka kvara pa je podložna kvaru sistema
- Performanse mogu biti nepredvidive jer zavise od mreže i sistema
- Mogu se pojaviti problemi sa upravljanjem ako su serveri u vlasništvu različitih organizacija

A client–server architecture for a film library

Figure shows an example of a film and video/DVD library organized as a client–server system.



MVC (Model – View – Controller)

MVC obrazac odvaja prezentaciju i interakciju od sistemskih podataka. Sistem je struktuisan u tri logičke cjeline koje međusobno djeluju.

- Komponenta Model – upravlja sistemskim podacima i povezanim operacijama nad tim podacima
- Komponenta View – definiše i upravlja načinom na koji se podaci predstavljaju korisniku.
- Komponenta Controller – upravlja interakcijom korisnika (pritisak tastera, klika miša) i prosljeđuje sve interakcije u View i Model.

Kada se koristi?

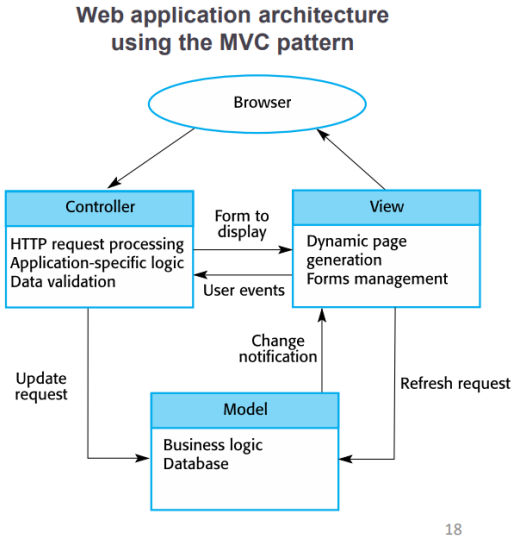
- Kada postoji više načina za pregled i interakciju sa podacima, takođe i kada su budući zahtjevi za interakciju i prezentaciju podataka nepoznati.

Prednosti?

- Omogućava da se podaci mijenjaju nezavisno od njihovog predstavljanja i obrnuto
- Podržava prezentaciju istih podataka na različite načine.

Mane?

- Može uključivati dodatan kod i složenost koda kada su model podataka i interakcija jednostavni.



Slojeviti arhitektonski obrazac

Organizuje sistem u skup slojeva od kojih svaki pruža skup usluga za slojeve iznad. Jedan sloj pruža usluge sloju iznad njega tako da slojevi najnižeg nivoa predstavljaju osnovne usluge koje će vjerovatno služiti cijelom sistemu. Ovaj stil podržava inkrementalni razvoj podsistema u različitim slojevima. Kada se promjeni interfejs sloja, to utiče samo na susjedni sloj.

Kada se koristi?

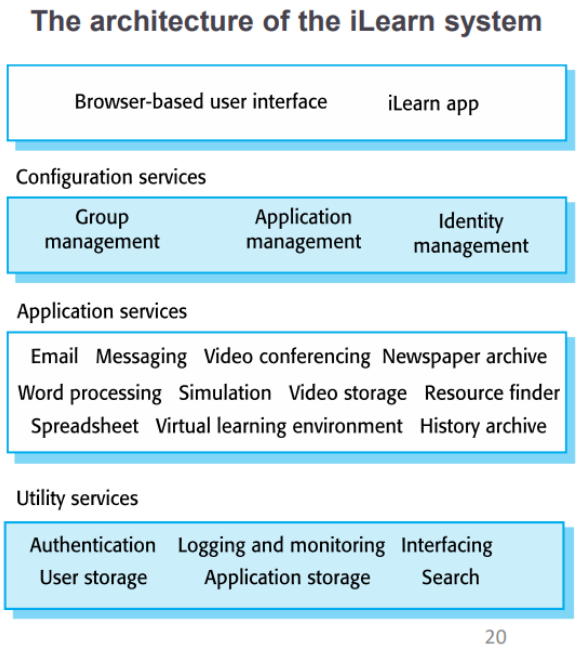
- Prilikom izgradnje novih objekata na osnovu već postojećih sistema, kada je razvoj raspoređen na nekoliko timova tako da je svaki odgovoran za sloj funkcionalnosti

Prednosti?

- Omogućava zamjenu cijelih slojeva dok god se održava interfejs
- Suvišni objekti se mogu obezbjediti u svakom sloju da bi se povećavala pouzdanost sistema

Mane?

- U praksi, obezbjeđivanje čistog razdvajanja između slojeva je teško i sloj visokog nivoa će često morati da stupi u interakciju sa najnižim slojem direktno, a ne preko međuslojeva
- Performanse mogu biti problem zbog više nivoa interpretacije zahtjeva za uslugu dok se obrađuje na svakom sloju



Repozitorijumski model

Pošto podsistemi moraju da razmjenjuju podatke, ovo se može uraditi na dva načina:

1. Zajednički podaci se čuvaju u centralnoj bazi podataka ili repozitorijumu i mogu im pristupiti svi podsistemi
2. Svaki podsistem održava sopstevnu bazu podataka i eksplicitno prosljeđuje podatke drugim podsistemima

Kada se velike količine podataka dijele najčešće se koristi **model repozitorijuma**, jer je to efikasan mehanizam dijeljenja podataka. Svim podacima u sistemu se upravlja u centralnom repozitorijumu koje je dostupno svim komponentama sistema. Komponente ne komuniciraju direktno, već samo preko repozitorijuma.

Kada se koristi?

- Kada imamo sistem u kome se stvaraju velike količine informacija koje se moraju čuvati dugo vremena
- U data-driven sistemima gdje uključivanje podataka u repozitorijum pokreće akciju ili alat.

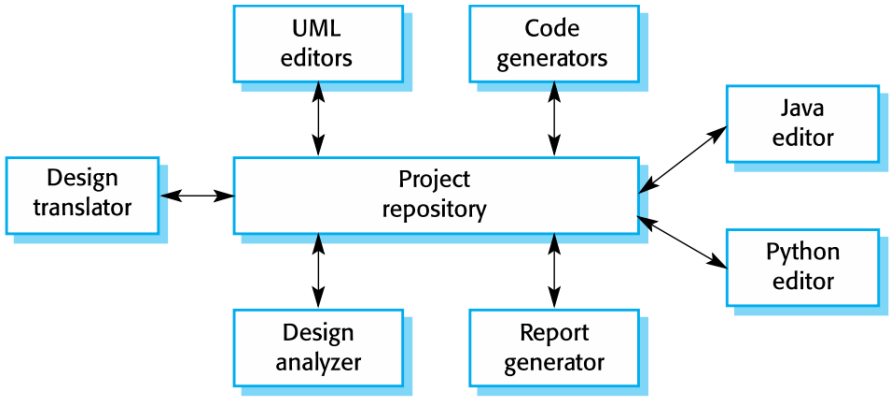
Prednosti?

- Komponente mogu biti nezavisne, tj. nije im potrebno da znaju za postojanje ostalih komponenti
- Podacima se može dosljedno upravljati pošto je sve na jednom mjestu.

Mane?

- Repozitorijum je jedna tačka kvara, tako da kvar u repozitorijumu utiče na cijeli sistem
- Može biti neefikasno oragnizovati sve komunikacije kroz repozitorijum
- Distribucija repozitorijuma na nekoliko računara može biti teška.

A repository architecture for an IDE



8. Integraciono testiranje.

Definisano je kao sistematska tehnika za konstruisanje softverske arhitekture. U isto vrijeme kada se odvija intergracija, sprovode se testovi da bi se otkrile greške povezane sa interfejsima. Cilj je uzeti module testirane na jedinici i izgraditi strukturu programa na osnovu propisanog dizajna.

Postoje dva pristupa, a to su:

- Neinkrementalno testiranje integracije (Big bang)
- Inkrementalno tesitranje integracije (odozgo prema dole, odozdo prema gore, sendvič).

Big bang pristup

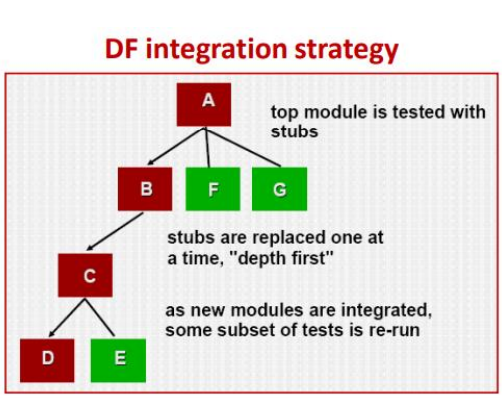
Zasniva se na tome da se sve komponente kombinuju unaprijed i cijeli program se testira kao cjelina, te se nailazi na mnoge, naizgled nepovezane greške (rezultat haosa). Ispravljanje grešaka je teško jer je izolacija uzroka komplikovana, a kada se ispravi jedan skup grešaka, pojavljuju se nove greške (beskonačna petlja).

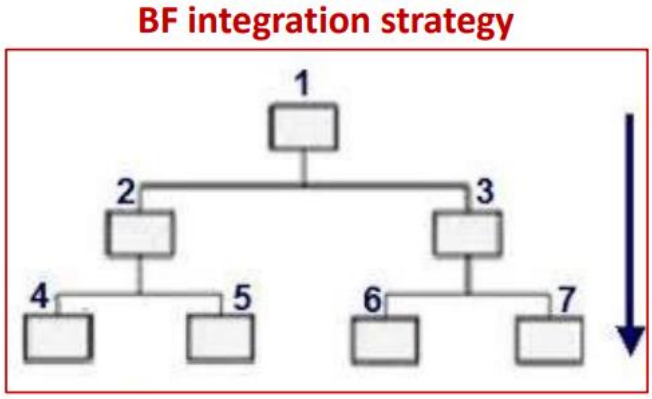
Inkrementalne strategije se zasnivaju na tome da se program konstruiše i testira u malim koracima. Sa takvim pristupom greške je lakše izolovati i ispraviti, interfejski će vjerovatno biti u potpunosti testirani.

Odozgo prema dole strategija integracije

Počinje se od glavnog modula i kreće se naniže kroz kontrolnu hijerarhiju. Podređeni moduli se obrađuju po dubini (svi moduli na glavnoj kontrolnoj putanji su integrisani) ili po širini (svi moduli direktno podređeni integrisani su na svakom nivou).

Prednosti su što ovakav pristup verifikuje glavne kontrolne tačke ili tačke odlučivanja na početku procesa testiranja. Nedostaci su to što treba kreirati stubove[1] da bi zamjenili module koji još nisu napravljeni ili testirani (ovaj kod se kasnije odbacuje), a pošto se stubovi koriste za zamjenu modula nižeg nivoa dugo ne može doći do značajnog protoka podataka.

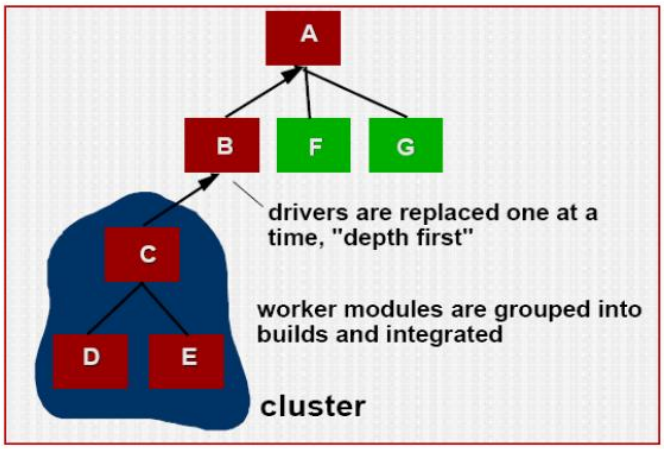




Odozdo prema gore strategija integracije

Kod ove strategije integracije, integracija i testiranje se kreće od najizvedenijeg modula i kreće se prema gore kroz kontrolnu hijerarhiju. Prednosti ovakvog pristupa su što on verifikuje podatke niskog nivoa rano u procesu testiranja, te nema potrebe za stubovima. Nedostaci su ti što moduli drajvera moraju biti napravljeni za testiranje modula nižeg nivoa (kod se kasnije proširuje i ubacuje u verziju sa punim funkcijama). Takođe upravljački programi ne sadrže kompletne algoritme koji će na kraju koristiti usluge modula nižeg nivoa – testiranje može biti nepotpuno ili će možda biti potrebno više testiranja na višim nivoima.

1. Komponente niskog nivoa se kombinuju u klastere koji obavljaju određenu softversku podfunkciju.
2. Upravljački program je napisan da koordinira unos i izlaz testnog slučaja, nakon toga klaser je testiran.
3. Drajveri se uklanjaju i klasteri se kombinuju krećući se nagore u strukturi programa.



Strategija integracije sendviča

[1] Stubovi u SI su pojednostavljeni dijelovi koda koji simuliraju ponašanje softverskih komponenti ili modula.

Kombinuje pristupe odozgo prema dole i odozdo prema gore i odvija se na modulima najvišeg nivoa, kao i na modula najnižeg nivoa. Sprovodi se korišćenjem funkcionalnih grupa modula, pri čemu je svaka grupa završena prije sljedeće. Moduli niskog i visokog nivoa su grupisani na osnovu kontrole i obrade podataka koju pružaju za određenu funkciju programa. Integracija unutar grupe napreduje naizmjenično između modula visokog i niskog nivoa grupe. **Kada se završi integracija za određenu funkcionalnu grupu, integracija i testiranje prelaze na sljedeću grupu.** U ovakvoj strategiji su iskorištene prednosti oba tipa integracije te se minimizuje potreba za drajverima i stubovima. Pristup mora biti disciplinovan da integracija ne bi težila scenariju „Big bang-a".

9. Spiralni softverski procesni model.

Spiralni model je evolutivni softverski procesni model koji spaja iterativnu prirodu prototipa sa kontrolisanim i sistematskim aspektima linearnog sekvencijalnog modela. Pruža potencijal za brzi razvoj inkrementalnih verzija softvera. Koristeći spiralni model, softver se razvija u nizu inkrementalnih izdanja. Tokom ranih iteracija, inkrementalno izdanje može biti papirni model ili prototip. Tokom kasnijih iteracija proizvode se sve potpunije verzije projektovanog sistema.

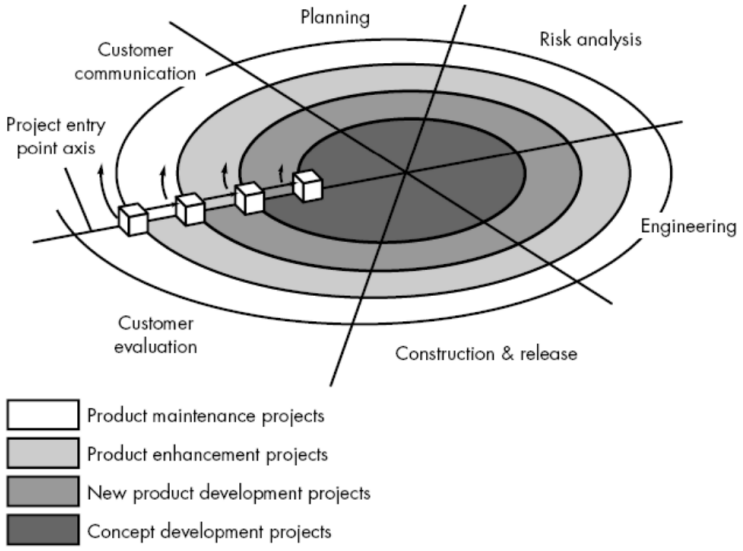
Spiralni model je podijeljen na nekoliko okvirnih aktivnosti koji se nazivaju regiona zadataka, obično 3 do 6, a to su:

- Komunikacija sa kupcima
- Planiranje
- Analiza rizika
- Inženjering
- Izgradnja i izdavanje
- Procjena korisnika.

Svaki region zadatka je popunjen radnim zadacima – set zadataka – koji su prilagođeni karakteristikama projekta koje treba preduzeti. Kako ovaj evolutivni proces počinje SI tim se kreće oko spirale:

1. Može rezultovati specifikacijom proizvoda
2. Može se koristiti za razvoj prototipa, itd.

Evolutionary software process models – The spiral model



10. Dijagram sekvence.

P-08: Dijagrami interakcije.

11. Projektni obrasci i njihova primjena u projektovanju softvera.

Obrazac dizajna je način ponovnog korišćenja apstraktnog znanja o problemu i njegovom rješenju. Obrazac je opis problema i suštine njegovog rješenja. Trebalo bi da bude dovoljno apstraktno da se može ponovo koristiti u različitim okruženjima. Opisi obrazaca obično koriste objektno-orijentisane karakteristike kao što su nasljeđivanje i polimorfizam.

Elementi obrazacu su:

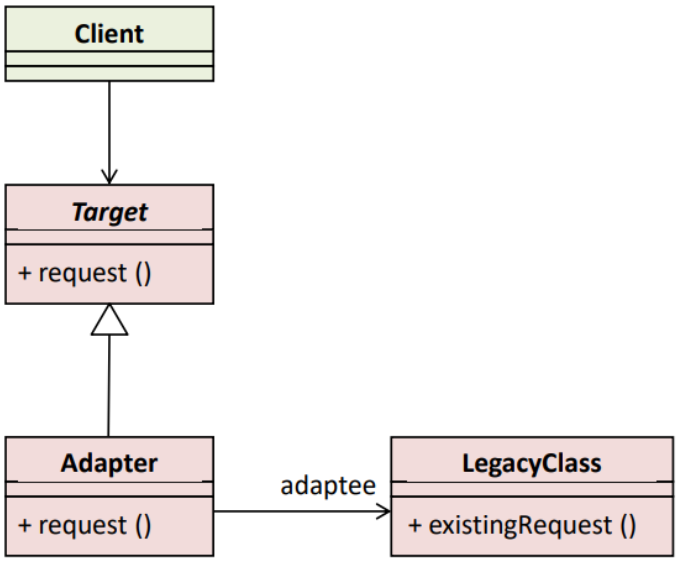
- Ime – značajan identifikator šablona
- Opis problema
- Opis rješenja – ne konkretan dizajn, već šablon za dizajnersko rješenje koje se može instancirati na različite načine
- Posljedice – rezultati i kompromisi primjene šablona

Obrazac dizajna – Adapter

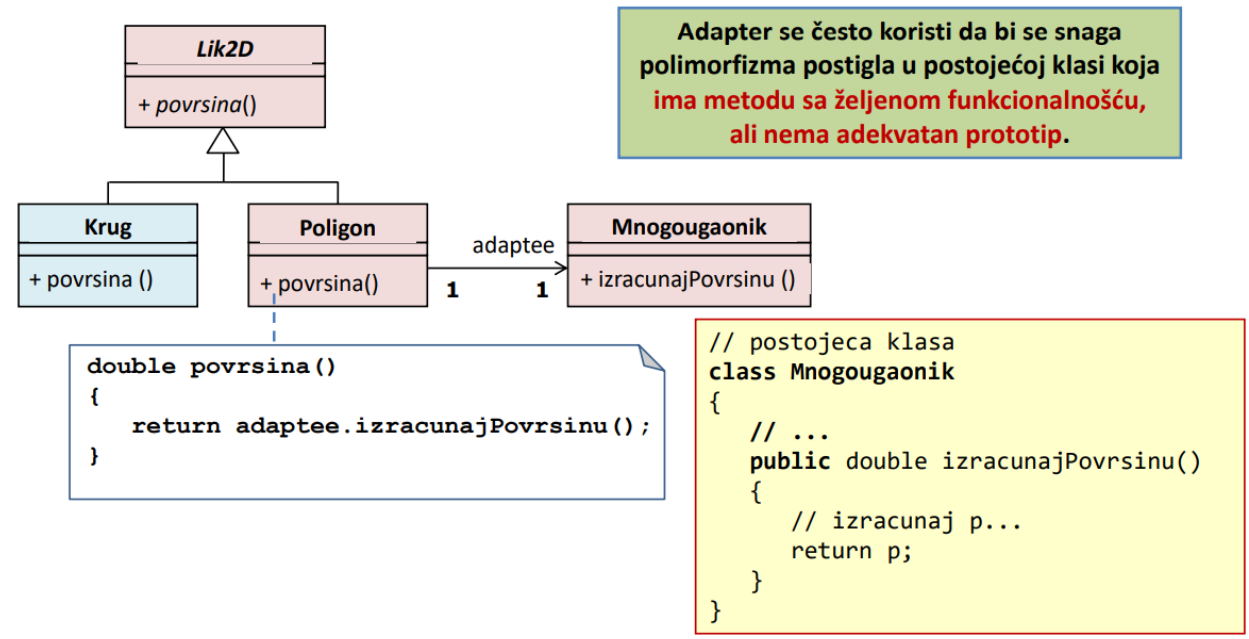
Obezbjeđuje interfejs između različitih klasa. Prilagođava interfejs jedne klase u interfejs kakav očekuje druga klasa čime omogućava komunikaciju kakva bi inače bila neizvodljiva.

- Client – objekat koji zahtjeva interfejs Target
- Target – definiše specifični interfejs koji koristi klasa Client
- Adapter – prilagođava postojeći interfejs LegacyClass (Adaptee) prema interfejsu klijentske klase, realizacija – nasljeđivanje interfejsa Target ← Adapter, delegacija – Adapter → Adaptee, adaptee.existingRequest()
- LegacyClass (Adaptee) – reprezentuje postojeći interfejs koji treba da se prilagodi

Nefunkcionalni sistemski zahtjevi koji su osnov za ADAPTER: „mora da komunicira sa postojećim objektom“.



Example:



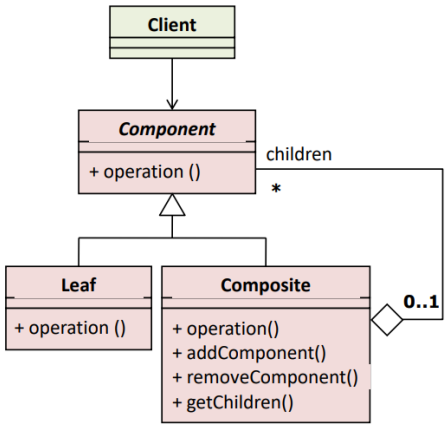
Kompozitni obrazac

Formira složenu hijerarhijsku strukturu proizvoljne širine i dubine. Omogućava korisniku da na isti način tretira proste i složene objekte koji su dio složene strukture.

- Component – deklariše interfejs za objekte u kompoziciji, implementira ponašanje zajedničko svim klasama, deklariše interfejs za pristup podelementima, (opciono) definiše i implementira interfejs za pristup roditeljskom elementu u rekurzivnim strukturama
- Leaf – reprezentuje prosti objekat u kompoziciji, definiše ponašanje za proste objekte u kompoziciji

- Composite – definiše ponašanje složenih elemenata, čuva podelemente, implementira operacije za podelemente iz interfejsa klase Component
- Client – manipuliše objektima kompozicije kroz interfejs klase Component

Nefunkcionalni sistemski zahtjevi koji su osnov za COMPOSITE: „kompleksna struktura“, „proizvoljne širine i dubine“, ...



12. Korisničko testiranje.

Tipovi korisničkog testiranja:

1. Alfa testiranje – sprovode krajnji korisnici na lokaciji programera. Softver se koristi u prirodnom okruženju sa programerima koji pažljivo posmatraju. Testiranje se sprovodi u kontrolisanoj životnoj sredini.
2. Beta testiranje – Sprovodi se na lokacijama krajnjih korisnika, programer obično nije prisutan, služi kao živa aplikacija softvera u okruženju koje ne može kontrolisati developer (programer), krajnji korisnik bilježi sve probleme sa kojima se susretao i o njima izvještava developere u redovnim intervalima.

13. Navesti i objasniti nivoe zrelosti softverskih procesa.

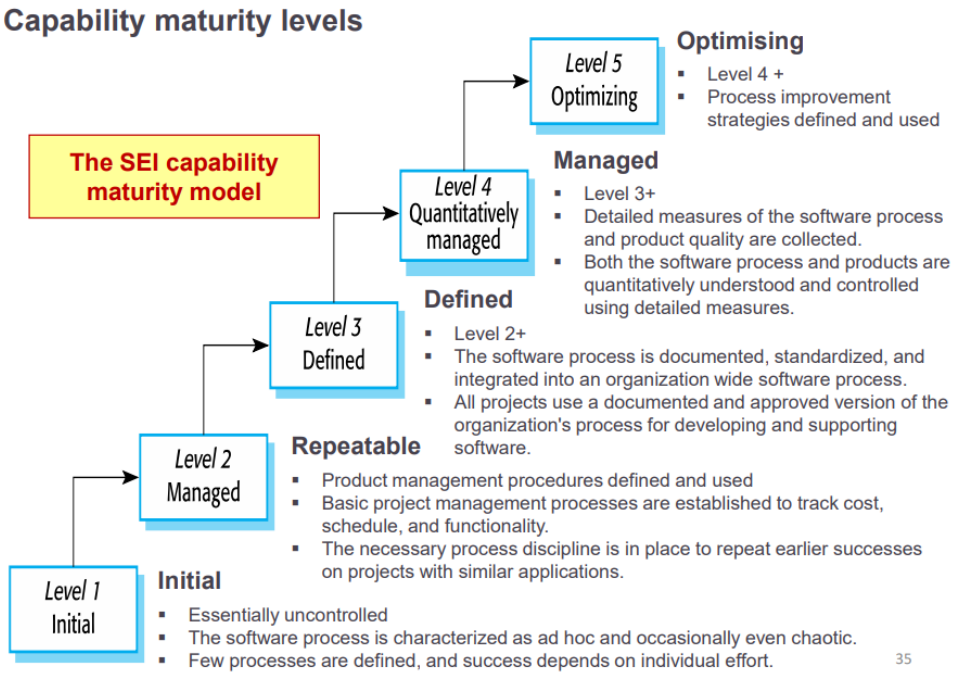
Nivo 1 (inicijalni nivo) – U suštini nije kontrolisan. Softverski proces je okarakterisan kao „ad hod“ i povremeno je čak haotičan. Nekoliko procesa je definisano, a uspjeh zavisi od individualnog napora.

Nivo 2 (ponovljivi) – Definisanje i korišćenje procedure za upravljanje softverom. Osnovni procesi upravljanja projektom su uspostavljeni da bi se pratili troškovi, raspored i funkcionalnost. Neophodna procesna disciplina je uspostavljena za ponavljanje ranijih uspjeha na projektima sa sličnim aplikacijama.

Nivo 3 (definisanost) – Nivo 2+. Softverski proces je dokumentovan, standardizovan i integrisan u organizovani softverski proces. Svi projekti koriste dokumentovanu i odobrenu verziju organizacionog procesa za razvoj i podršku softvera.

Nivo 4 (upravljanje) – Nivo 3+. Prikupljene su detaljne mjere softverskog procesa i kvaliteta proizvoda. I softverski proces i proizvod su kvantativno shvaćeni i kontrolisani korištenjem detaljnih mjera.

Nivo 5 (optimizovanje) – Nivo 4+. Strategija unapređenja procesa je definisana i korištena.

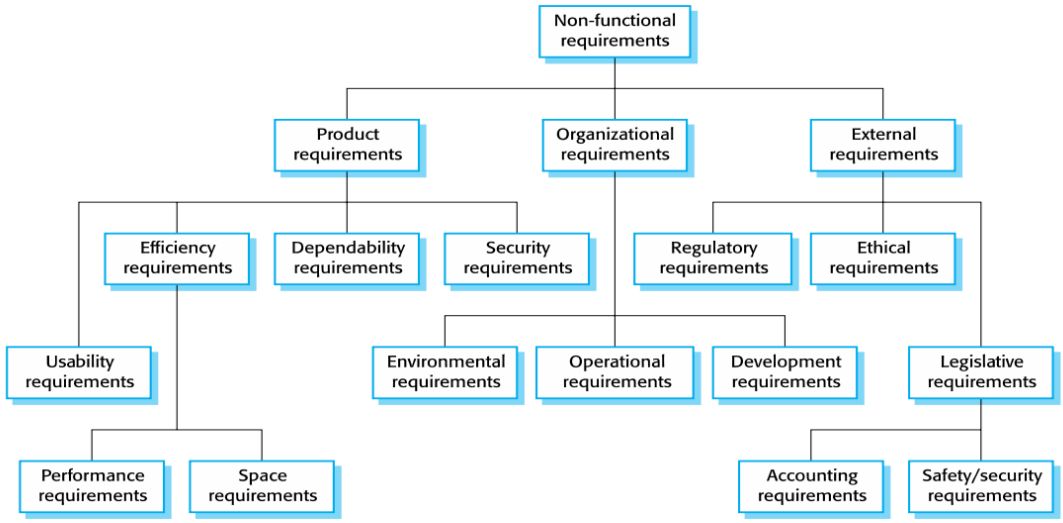


14. Klasifikacija nefunkcionalnih zahtjeva.

Nefunkcionalni zahtjevi proizvoda – ograničenja na usluge ili funkcije koje nudi sistem, kao što su vremenska ograničenja, ograničenja u procesu razvoja, standardi itd. Često se primjenjuje na sistem u cjelini, a ne na pojedinačne usluge ili karakteristike.

Klasifikacija:

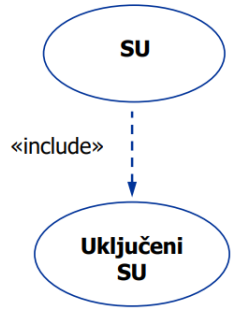
1. Nefunkcionalni zahtjevi proizvoda – zahtjevi koji preciziraju da se sistem mora ponašati na određeni način, npr. brzina izvršavanja, pouzdanost...
2. Organizacioni nefunkcionalni zahtjevi – zahtjevi koji su posljedica organizacionih polisa i procedura, npr. zahtjevi za implementaciju
3. Eksterni nefunkcionalni zahtjevi – zahtjevi koji proizilaze iz spoljnih faktora i procesa njegovog razvoja, npr. interoperabilnost, zakonodavstvo.



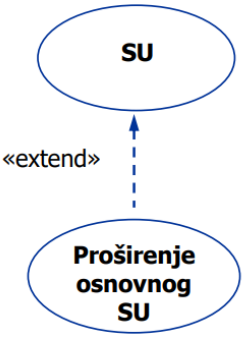
15. Veze između slučajeva upotrebe u dijagramu slučajeva upotrebe.

Slučaj upotrebe (use case) je apstrakcija koja opisuje jednu klasu scenarija upotrebe sistema – konkretna funkcionalnost koju sistem obezbjeđuje učesniku. Veze između slučajeva upotrebe:

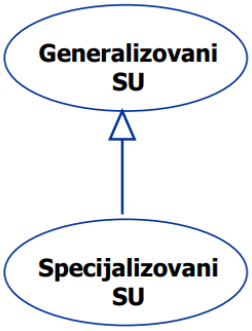
- Zavisnost <<include>> - dio funkcionalnosti koju slučaj upotrebe pruža učesnicima, nezavisno od slučaja upotrebe koja se obezbjeđuje učesnicima

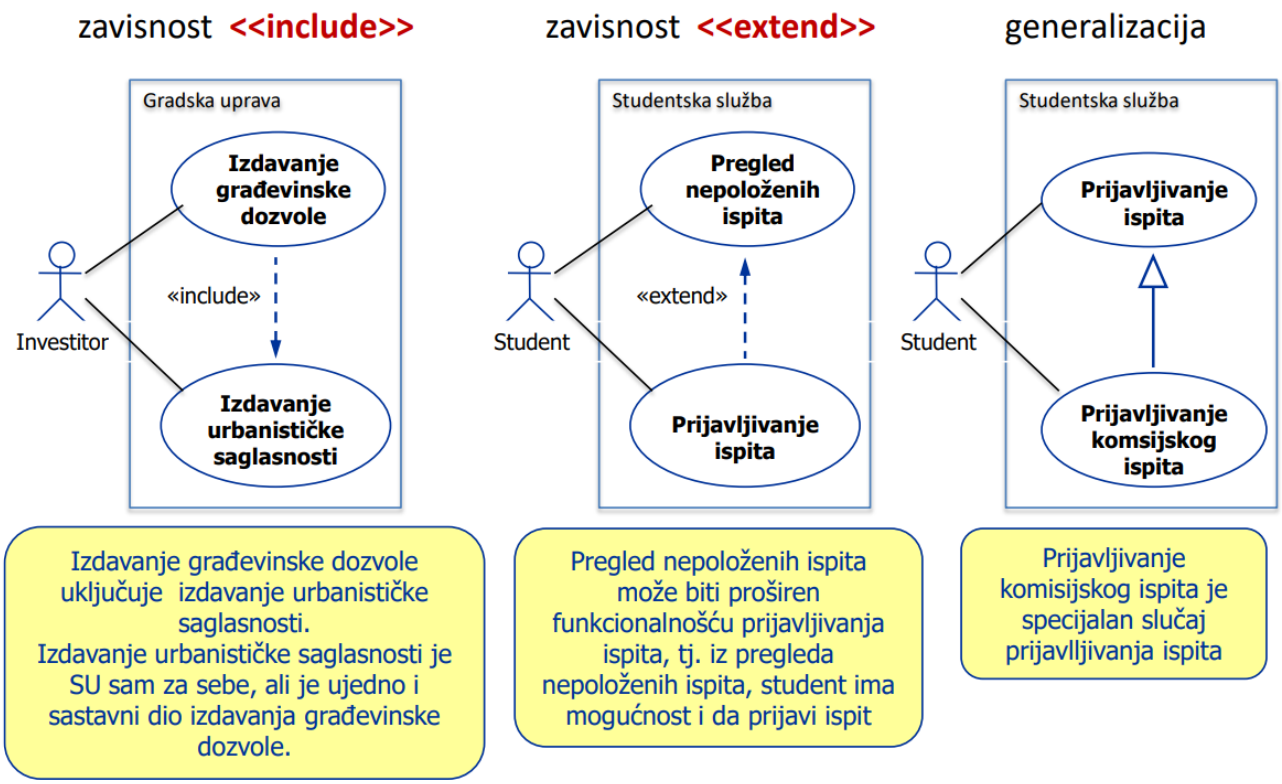


- Zavisnost <<extend>> - funkcionalnost proširenja, može, a i ne mora, proširiti osnovni slučaj upotrebe



- Generalizacija – specijalni oblik osnovnog slučaja upotrebe





16. Navesti, objasni i ilustrovati veze „cjelina - dio“ u dijagramu klasa.

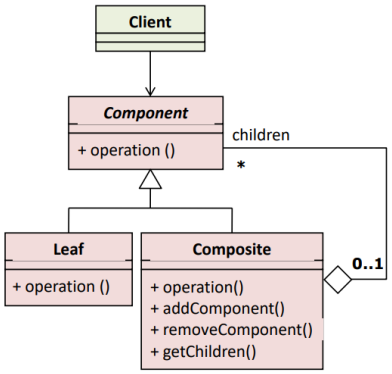
P-07: Dijagram klasa.

17. Objasniti i ilustrovati projektni obazac „composite“.

Formira složenu hijerarhijsku strukturu proizvoljne širine i dubine. Omogućava korisniku da na isti način tretira proste i složene objekte koji su dio složene strukture.

- Component – deklariše interfejs za objekte u kompoziciji, implementira ponašanje zajedničko svim klasama, deklariše interfejs za pristup podelementima, (opciono) definiše i implementira interfejs za pristup roditeljskom elementu u rekurzivnim strukturama
- Leaf – reprezentuje prosti objekat u kompoziciji, definiše ponašanje za proste objekte u kompoziciji
- Composite – definiše ponašanje složenih elemenata, čuva podelemente, implementira operacije za podelemente iz interfejsa klase Component
- Client – manipuliše objektima kompozicije kroz interfejs klase Component

Nefunkcionalni sistemski zahtjevi koji su osnov za COMPOSITE: „kompleksna struktura“, „proizvoljne širine i dubine“, ...



18. „Top-down“ strategija integracionog tesitranja.

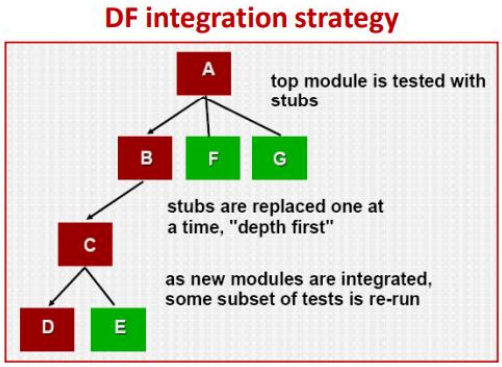
Integraciono testiranje je definisano kao sistemska tehnika za konstruisanje softverske arhitekture. Kada se vrši integracija, rade se i testovi da bi se otklonile greške koje povezuju interfejse.

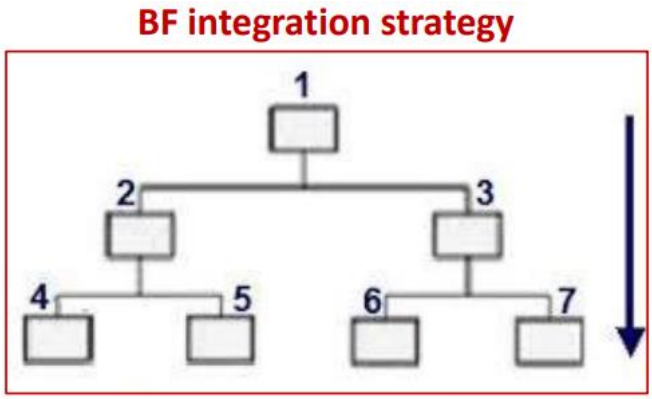
Postoje dva pristupa: neinkrementalni (Big bang) i inkrementalni (top-down, down-top i sendvič).

„Top-down“ strategija integracije

Počinje se od glavnog modula i kreće se naniže kroz kontrolnu hijerarhiju. Podređeni moduli se obrađuju po dubini (svi moduli na glavnoj kontrolnoj putanji su integrisani) ili po širini (svi moduli direktno podređeni integrisani su na svakom nivou).

Prednosti su što ovakav pristuo verifikuje glavne kontrolne tačke ili tačke odlučivanja na početku procesa testiranja. Nedostaci su to što treba kreirati stubove da bi zamjenili module koji još nisu napravljeni ili testirani (ovaj kod se kasnije odbacuje), a pošto se stubovi koriste za zamjenu modula nižeg nivoa dugo ne može doći do značajnog protoka podataka.





19. Sekvencijalni softverski procesni modeli.

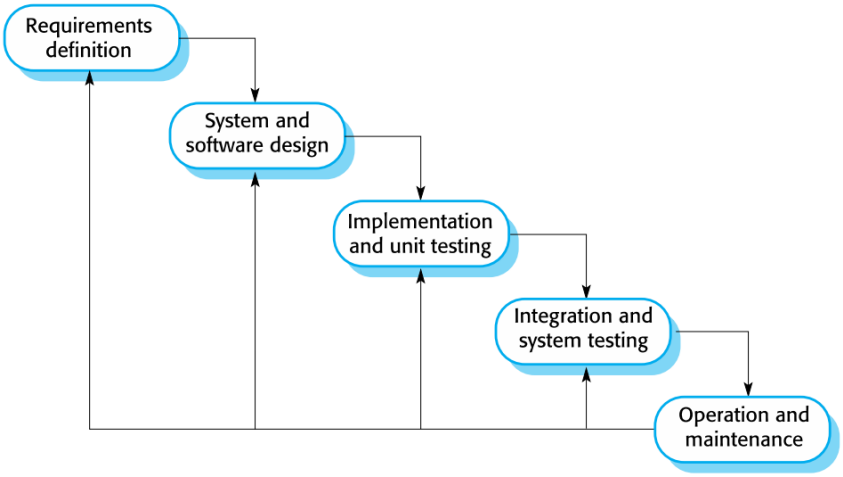
Sekvencijalni softverski procesni modeli su planski vođeni modeli koji imaju odvojene i različite faze specifikacije i razvoja. U njih spadaju model vodopada i V model.

Model vodopada

Tipično se zove i klasični životni ciklus. U ovom modelu razlikujemo sljedeće faze:

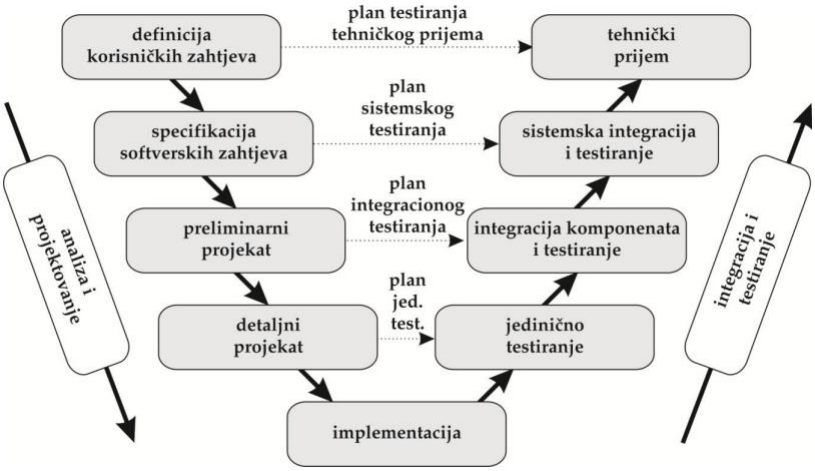
- analiza i definicija zahtjeva
- dizajn sistema i softvera
- implementacija i jedinično testiranje
- integracija i testiranje sistema
- rad i održavanje

Glavni nedostatak ovog modela je teškoća promjena nakon što se faza završi. U principu, faza mora biti završena prije nego što se pređe na drugu fazu. Takođe, nefleksibilna podjela projekta na različite faze otežava odgovor promjenljive zahtjeve kupaca. Kupcu je teško da navede sve zahtjeve, a model vodopada to zahtjeva. Model vodopada se teško prilagođava na promjene. Pravi projekti rijetko koriste sekvencijalni tok aktivnosti. Osim toga, kupac mora biti strpljiv jer radna verzija sistema neće biti dostupna do kasno u vremenskom periodu projekta. Ovakav model se uglavnom koristi za velike sistemske inžinjerske projekte gdje se sistem razvija na nekoliko lokacija.



V – model

V-model obezbjeđuje sredstva za testiranje softvera u svakoj fazi na obrnuti način. U svakoj fazi kreiraju se planovi testiranja i test slučajevi kako bi se verifikovao i validirao proizvod u skladu sa zahtjevima te faze. Verifikacije i validacija idu paralelno. Ovaj model se takođe može nazivati modelom verifikacije i validacije.



20. Izmamljivanje zahtjeva.

Takođe se naziva i okupljanje/otkrivanje zahtjeva. Ono uključuje tehničko osoblje koje radi sa klijentima kako bi saznalo o domenima aplikacije, uslugama koje sistem treba da pruži i operativnim ograničenjima sistema. Može uključivati krajnje korisnike, menadžere, inženjere uključene u održavanje, stručnjake iz domena, sindikate itd. Oni se nazivaju zainteresovanim stranama (stakeholderima). Problemi koji se javljaju jesu to što zainteresovane strane ne znaju šta zaista žele, i ne znaju svoje zahtjeve da izraze pravilnim terminima. Pored toga, različite zainteresovane strane mogu imati različite zahtjeve. Takođe, organizacioni i politički faktori mogu uticati na sistemske zahtjeve.

Aktivnosti i proces izmamljivanja zahtjeva se odvija u četiri faze:

- Otkrivanje zahtjeva – vrši se interakcija sa zainteresovanim stranama kako bi se otkrili njihovi zahtjevi
- Klasifikacija i organizacija zahtjeva – grupišu se povezani zahtjevi i organizuju se u klastere
- Određivanje prioriteta i pregovaranje (rješavanje sukova zahtjeva)
- Specifikacije zahtjeva – zahtjevi se dokumentuju i unose u sljedeći krug spirale.

Otkrivanje zahtjeva – prikupljanje informacija o potrebnim i postojećim sistemima.

Tehnike izmamljivanja – intervjui (formalni i neformalni, zatvoreni i otvoreni), upitnici, posmatranje, prototipovi, grupne radionice.

U praksi se najčešće koristi mješavina zatvorenog i otvorenog intervjua. Intervjui nisu dobri za razumjevanje zahtjeva domena, stručnjaci za aplikacije mogu da koriste jezik da opišu svoj rad koji inžinjeri zahtjeva ne razumiju.

Izmamljivanje zahtjeva posmatranjem (etnografija) – društveni naučnik provodi dosta vremena posmatrajući i analizirajući kako ljudi stvarno rade, te ljudi ne moraju objašnjavati svoj rad. Etnografija je efikasna za razumijevanje postojećih procesa, ali ne može da identifikuje nove karakteristike koje treba dodati sistemu.

Priče i scenariji – primjeri iz stvarnog života kako se sistem može koristiti.

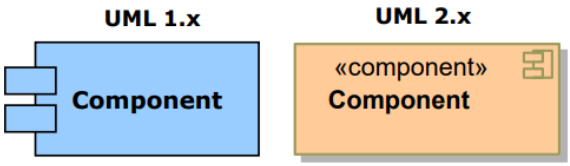
Scenariji – struktuirani oblik korisničke priče. Scenariji treba da uključuju opis početne situacije, opis normalnog toka događaja, opis onoga što može poći po zlu, informacije o drugim istovremenim aktivnostima, opis stanja kada se scenarijo završi.

21. UML dijagrami za reprezentaciju arhitekture softverskih sistema.

Koriste se dvije UML notacije za predstavljanje softverskih arhitektura, a to su:

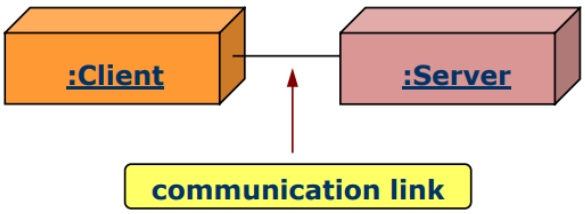
- Dijagram komponenti – predstavlja logičku arhitekturu softverskog sistema (razvojni prikaz modela 4+1)
- Deployment dijagram – predstavlja postavljanje softverskih artefakata na hardverske čvorove

Dijagram komponenti može se nazvati i “dijagram ožičenja softvera”. On prikazuje softverske komponente i njihove veze. Veze komponenti = zavisnost. Intefejs = skup operacija koje zahtjeva ili obezbjeđuje(daje) neka komponenta.

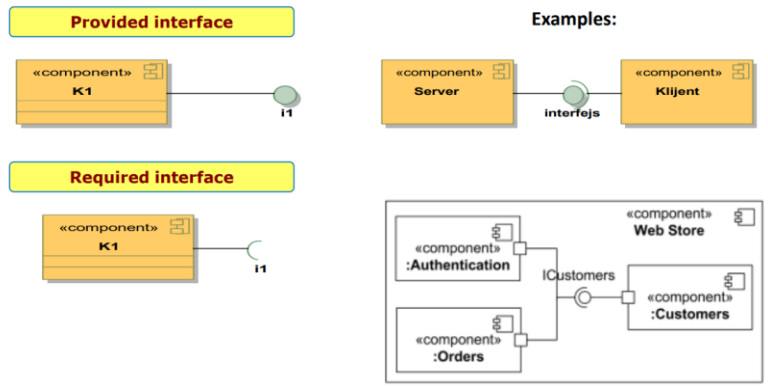


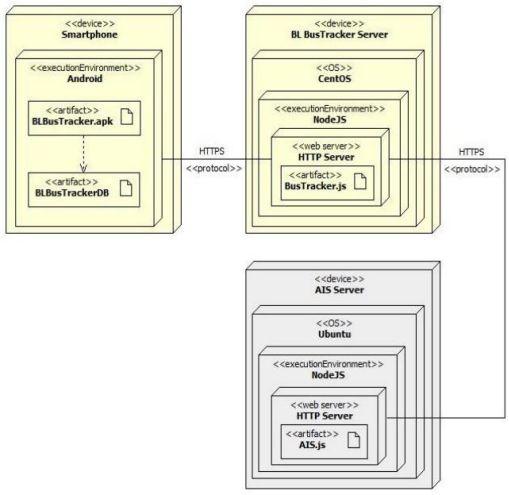
Deployment dijagram – predstavlja primjenu softverskih artefakata na hardverskim čvorovima (run-time). On sadrži:

- Čvorove – apstrakcija hardverskih čvorova (npr. server) i okruženja za izvršavanje (npr. operativni sistem)
- Komunikacione veze – veze između čvorova



Component diagram





22. Statička verifikacija.

Statička verifikacija se bavi analizom prikaza statičkog sistema radi otkrivanja problema. U statičku verifikaciju spadaju revjui/inspekcije. Može se nadopuniti na osnovu alata za analizu dokumenata i koda.

Review

Šta je review?

- Sastanak koji predvode tehnički ljudi za tehničke ljude
- Review je tehnička procjena radnog proizvoda nastalog tokom procesa softverskog inzinjeringa
- Mehanizam za osiguranje kvaliteta softvera
- Poligon za obuku.

Šta review nije?

- Rezime projekta ili procjena napretka
- Sastanak namjenjen isključivo prenošenju informacija
- Mehanizam političke ili lične odmazde

Šta tražimo?

- Greške i nedostatke
    - Greška – problem kvaliteta pronađen prije nego što je softver pušten krajnjim korisnicima
    - Defekt – problem kvaliteta koji se javlja tek nakon što je softver pušten krajnjim korisnicima
- Ovu razliku pravimo zato što greške i defekti imaju veoma različit ekonmski, psihološki, poslovni i ljudski uticaj

Isplativost Review-a

Napor koji se ulaže kada se koriste review-i se povećava na početku razvoja softvera, ali ovo rano ulaganje u review-e je isplativo jer se smanjuje napor na testiranju i korekciji. **Datum implementacije za razvoj sa pregledima je prije nego što je datum implementacije bez pregleda.** Review-i ne uzimaju vrijeme, oni ga čuvaju!

Analiza troškova pokazuje da proces bez review-a košta otprilike 3-4 puta više od procesa sa pregledima, uzimajući u obzir troškove ispravljanja latentnih defekata.

Neformalni review-i uključuju

- Jednostavnu provjeru radnog softverskog proizvoda sa kolegom na stolu.
- Slučajan sastanak, koji uključuje više od dvije osobe u svrhu pregleda radnog proizvoda
- review-orijentisani aspekti programiranja u paru

Programiranje u paru uključuje kontinuirani pregled dok se stvara radni proizvod (dizajn ili kod). Prednost je trenutno otkrivanje grešaka.

Formalni tehnički review

Ciljevi:

- da se otkriju greške u funkciji, implementaciji ili logici u svakoj reprezentaciji softvera

- da se provjeri da li softver ispunjava potrebne zahtjeve
- da se osigura da je softver predstavljen sa unaprijed definisanim standardima
- da se postigne softver koji je razvijen na jedinstven način
- da se projektom može lakše upravljati

Formalni tehnički pregled je u stvari klasa pregleda koja uključuje uputstva i inspekcije.

Sastanak:

- tipično 3-5 ljudi
- trebala bi se uraditi unaprijed priprema ali ne bi trebala uzeti više od dva sata svakom učesniku
- sastanak bi trebalo da bude kraći od 2 sata
- fokus bi trebalo da bude na radnom proizvod

Igrači:

- Proizvođač – pojedinac koji je razvio softver, obavještava vođu projekta da je radni proizvod završen i da je potrebno uraditi review
- Vođa review-a – procjenju spremnost proizvoda, generiše kopije materijala i distribuira ih do 2-3 revjuera za daljne pripreme
- Reviewer(s) – očekuje se da potroši 1-2 sata pregledavajući proizvod, pravljeći bilješke i da zna o čemu se radi
- Recorder(s) – recenzent koji zapisuje pismeno sve bitne stavke

Sprovođenje:

- Pregleda se proizvod, a ne proizvođač
- Treba se postaviti dnevni red i pridržavati ga se
- Treba ograničiti debatu i pobijanje
- Treba navesti problematična područja, ali ne i pokušati riješiti svaki problem
- Trebaju se voditi pisanje bilješke
- Ograničiti broj učesnika i insistirati na prethodnoj pripremi
- Razviti kontrolnu listu za svaki proizvod koji će biti pregledan
- Zakazati vrijeme i naći sve potrebne resurse
- Treba se provesti obuka za sve recenzente
- Treba pregledati ranije recenzije

<u>Review na osnovu uzorka (Sample-driven-Review)</u>

U idealnom slučaju, svaki proizvod bi prošao FTR, ali u stvarnosti, resursi su ograničeni i vremena je malo, pa se recenzije često preskaču.

SDR pristup:

- samo uzorci proizvoda se pregledavaju, da bi se otkrili proizvodi sa najviše grešaka
- onda se primjenjuje potpuni FTR samo na tim proizvodima, na osnovu uzorka

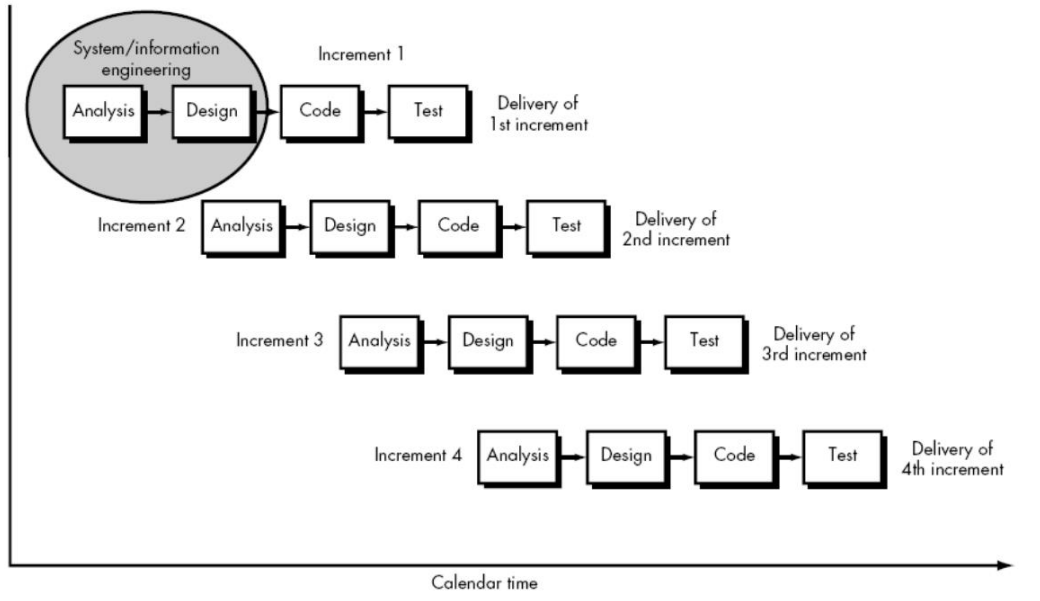
SDR proces:

- Ispitati djelić (ai) svakog softverskog radnog proizvoda i
- zabilježiti broj grešaka fi unutar određenog ai
- procjeniti konačan broj grešaka u radnom projektu i, množeći fi * 1/ai
- sortirati proizvode u opadajućem redoslijedu prema određenom broju grešaka u svakom proizvodu
- fokusirati dostupne review resurse na onim proizvoima koji imaju najviše grešaka

23. Evolutivni procesni model.

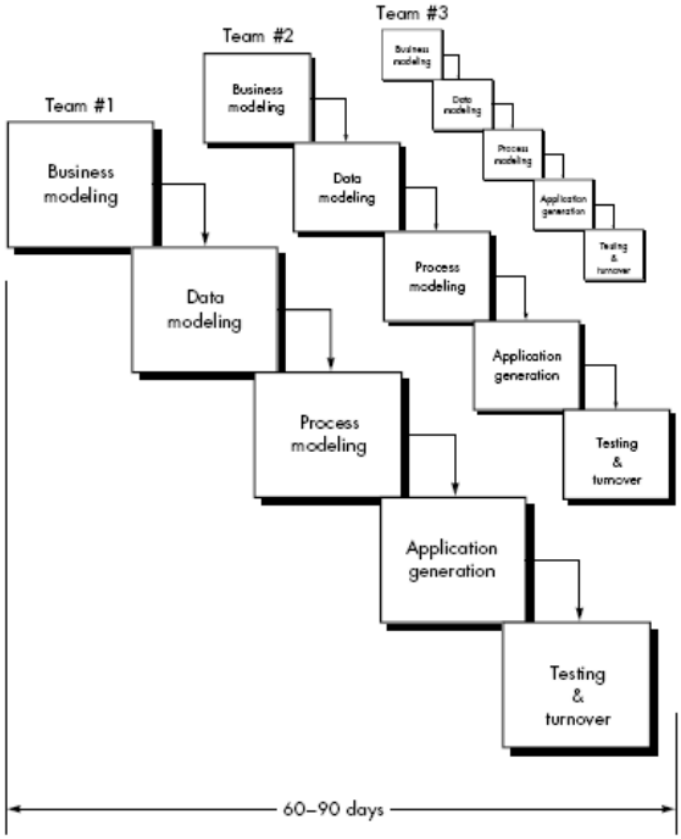
Specifikacija, razvoj i validacija se mogu preplitati. Proces može biti planski ili agilan. U ovu grupu spadaju inkrementalni model, RAD, RUP, spiralni model. Sve je više prihvaćeno da softver, kao i ostali sistemi, evoulira tokom određenog vremenskog perioda. Zahtjevi poslovanja i proizvoda se mijenja tokom razvoja, tako da praviti put do kranjeg proizvoda je nerealno. Skup osnovnih proizvoda ili sistemskim zahtjeva je dobro shvaćen ali detalji o proširenju sistema i proizvoda tek trebaju da budu definisani. U ovakvim i sličnim situacijama, softverskim inženjerima je potreban model koji je u stanju da se razvija tokom vremena.

Inkrementalni model kombinuje elemente linearnog sekvencijalnog modela i primjenjuje ih na stepenast način kako kalendarsko vrijeme ide. Svaka linearna sekvenca proizvodi isporučivi "inkrement" softvera. Prvi inkrement je često osnovni proizvod. Osnovni proizvod koristi kupac te kao rezultat razvija se plan za sljedeći inkrement. Plan je bavi modifikacijom osnovnog proizvoda kako bi se zadovoljile potrebe kupca isporukom dodatnih funkcija i funkcionalnosti. Ovaj proces se ponavlja sve dok se ne proizvede kompletan proizvod.



RAD model (Rapid application development – brzi razvoj aplikacija) je inkrementalni model procesa razvoja softvera koji naglašava izuzetno kratak razvojni ciklus. RAD model je adaptacija "velike brzine" modela vodopada u kome se brz razvoj postiže korišćenjem konstrukcije zasnovane na komponentama.

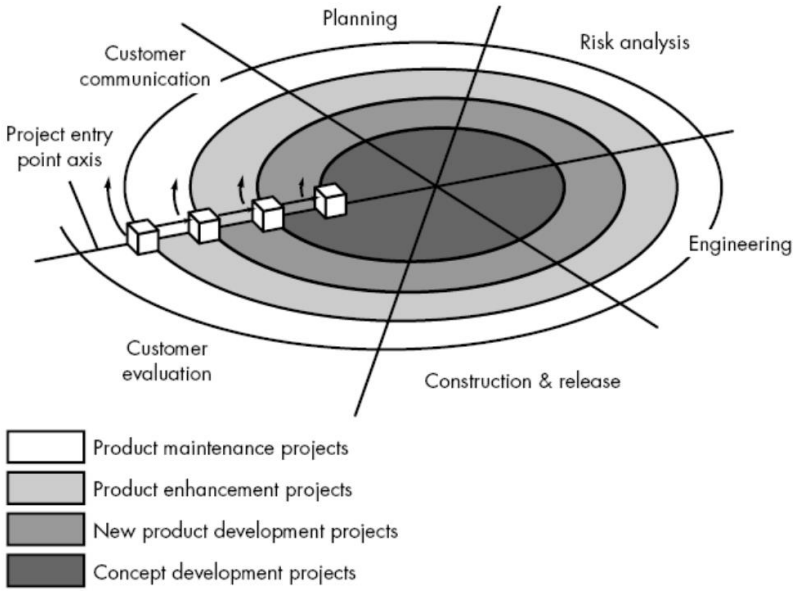
Ako su zahtjevi dobro shvaćeni i obim projekta ograničen, RAD proces omogućava razvojnom timu da stvori "potpuno funkcionalan sistem" u veoma kratkim vremenskim periodama (60-90 dana).



Spiralni model je evolutivni softverski procesni model koji spaja iterativnu prirodu prototipa sa kontrolisanim i sistematskim aspektima linearnog sekvencijalnog modela. Pruža potencijal za brzi razvoj inkrementalnih verzija softvera. Koristeći spiralni model, softver se razvija u nizu inkrementalnih izdanja. Tokom ranih iteracija, inkrement može biti papirni model ili prototip. Tokom kasnijih iteracija proizvodi se sve potpunije verzije projektovanog softvera. Spiralni model je podijeljen na nekoliko okvirnih aktivnosti koji se nazivaju regioni zadatka (obično 3-6) i to su:

- komunikacija sa kupcima
- planiranje
- analiza rizika
- inženjering
- izgradnja i oslobađanje
- procjena korisnika.

Svaki region je popunjen radnim zadacima koji su priagođeni karakteristikama projekta. Kako ovaj evolutivni proces počinje, SI tim se kreće oko spirale.



RUP model (The rational unified process – racionalni objedinjen proces) koristi UML alat za specifikaciju i dizajn. RUP je vođen slučajem upotrebe, arhitektura je usredsređena, iterativna i uključuje cikluse, faze, tokove posla, smanjenje rizika, kontrolu kvaliteta, upravljanje projektima i kontrolu konfiguracije. RUP dekomponuje rad velikog projekta na manje dijelove ili mini projekte, a svaki mini projekat je iteracija koja rezultuje povećanjem proizvoda. Iteracija dovodi do rasta proizvoda.

Prednosti inkrementalnog razvoja?

- troškovi promjene korisničkih zahtjeva se smanjuju, manja količina analize i dokumentacije u odnosu na vodopad
- lakše je dobiti povratne informacije od kupca
- moguća brža isporuka i primjena korisnog softvera kupcu

Nedostaci inkrementalnog razvoja?

- proces se ne vidi – menadžerima su potrebni redovni rezultati, ako se sistem brzo razvija, nije isplativo sve dokumentovati
- struktura ima tendeciju da degradira kako se dodaju novi koraci – kvari se struktura softvera, cijena

24. Specifikacija zahtjeva.

Specifikacija zahtjeva je proces pisanja korisničkih i sistemskih zahtjeva u dokument sa zahtjevima. Zahtjevi korisnika moraju biti razumljivi krajnjim korisnicima i kupcima koji nemaju tehničku pozadinu. Sistemski zahtjevi su detaljniji i mogu uključivati više tehničkih informacija. Zahtjevi mogu biti dio ugovora za razvoj sistema, te je potrebno da oni budu što kompletniji. Zahtjevi treba da navode šta to sistem treba da radi, a dizajn kako treba da to uradi. U praksi, zahtjevi i dizajn su neodvojivi.

Načini pisanja specifikacije sistemskih zahtjeva:

- Prirodni jezik – Zahtjevi su napisani korištenjem numerisanih rečenica na prirodnom jeziku. Svaka rečenica treba da izrazi jedan zahtjev. Lako za razumijevanje kupcima. Teško je postići preciznost bez otežavanja čitanja dokumenta.
- Strukturirani prirodni jezik – Zahtjevi su napisani prirodnim jezikom na standardnom obrazcu ili šablonu. Svako polje pruža informaciju o aspektu zahtjeva.
- Dizajn opisni jezici – Koristi se jezik poput programerskog jezika ali sa apstraktnijim karakteristikama za specifikaciju zahtjeva definisanjem operativnog modela sistema. Može biti koristan za specifikacije interfejsa, iako se sad rijetko koristi.
- Grafičke notacije – Koriste se grafički modeli dopunjeni tekstualnim napomenama (UML slučajevi upotrebe i dijagrami sekvence).
- Matematičke specifikacije – Ove specifikacije su zasnovane na matematičkim konceptima kao što su mašine stanja ili skupovi. Iako ove specifikacije smanjuju dvosmislenost u zahtjevima, većina kupaca ne razumije formalnu specifikaciju, jer ne mogu da vide da li to predstavlja ono što žele, i nerado to prihvataju kao sistemski ugovor.

Smjernice za pisanje specifikacije zahtjeva:

- izmislite standardan format i koristite ga za sve specifikacije
- koristiti jezik na dosljedan način
- koristiti isticanje teksta da bi se mogao izdvojiti ključan dio zahtjeva
- izbjegavati upotrebu kompjuterskog žargona
- uključiti obrazloženje zašto je zahtjev neophodan.

Specifikacije zasnovane na formi:

- definicija funkcije ili entiteta
- opis ulaza (input-a) i odakle oni dolaze
- opis output-a i gdje oni idu
- opis radnje koju treba preduzeti
- pre i post uslovi
- neželjeni efekti, ako ih ima

Tabelarna specifikacija:

- koristi se za dopunu prirodnog jezika
- posebno korisna kada morate da definišete niz alternativnih pravaca akcije

Za specifikaciju zahtjeva se mogu koristiti slučajevi upotrebe (use case) – to su vrsta scenarija koji su uključeni u UML. Oni identifikuju aktere u interakciji i opisuju samu interakciju. Skup slučajeva upotrebe bi trebalo da opiše sve interakcije sa sistemom. To je grafički model koji je dopunjen detaljnim tabelarnim opisom. UML dijagrami sekvence se mogu koristiti za dodavanje detalja slučajevima upotrebe tako što će prikazati redoslijed obrade događaja u sistemu.

Dokument sa softverskim zahtjevima (SRS – softver requirements specification) je zvanična izjava o tome šta se traži od sistem developera (programera). **Nije** projektni dokument, trebalo bi da odredi ŠTA sistem treba da radi, a ne KAKO treba to da odradi. Sistemi koji se razvijaju postepeno će imati manje detalja u SRS dokumentu.

Struktura dokumenta sa zahtjevima:

- Zaglavlje – definiše očekivanu čitanost dokumenta i opisuje njegovu istoriju verzija, uključujući objašnjenje za kreiranje nove verzije i rezime izmjena napravljenih u svakoj verziji
- Uvod – definiše potrebu za sistemom. Ukratko opisuje funkcije sistema i kako će sistem funkcionisati sa ostalim sistemima.
- Rječnik – definiše tehničke termine koji se koriste u dokumentu, ne treba praviti pretpostavke o stručnosti čitaoca.
- Korisnička definicija zahtjeva – opisuju se usluge koje se pružaju korisniku (FRs). Ovaj opis može da sadrži prirodan opis, dijagrame i ostale notacije koje su razumljivu korisniku. Treba specifikovati standarde proizvoda i procesa koji se moraju poštovati.
- Arhitektura sistema – treba da predstavlja pregled na visokom nivou predviđene arhitekture sistema, pokazujući distribuciju funkcija po sistemskim modulima. Arhitektonske komponente koje se ponovo koriste treba istaći.
- Sistemska definicija zahtjeva – treba da opiše FRs i NFRs detaljnije. Mogu se definisati i interfejsi za druge sisteme.
- Modeli sistema – Ovo može uključiti grafičke modele sistema koji pokazuju odnose između komponenti sistema sa sistemom i njegovog okruženja.
- Evolucija sistema – Opisuje osnovne pretpostavke na kojima je sistem zasnovan i sve predviđene promjene usljed evolucije hardvera, promjena potreba korisnika itd. Ovaj odjeljak je koristan za dizajnere sistema jer im može pomoći da izbjegnu odluke o dizajnu koje bi ograničile moguće buduće promjene sistema.
- Prilozi – Detaljne specifične informacije koje se odnose na aplikaciju u razvoju (npr. opis hardvera i baze podataka). Hardverski zahtjevi definišu minimalnu i optimalnu konfiguraciju sistema. Zahtjevi baze podataka definišu logičku organizaciju podataka koje sistem koristi i odnos između podataka.
- Indeks – U dokument mogu biti uključeni nekoliko indeksa. Kao i normalni alfabet indeks može postojati i indeks dijagrama, indeks funkcija itd.

25. Prototipski razvoj softvera.

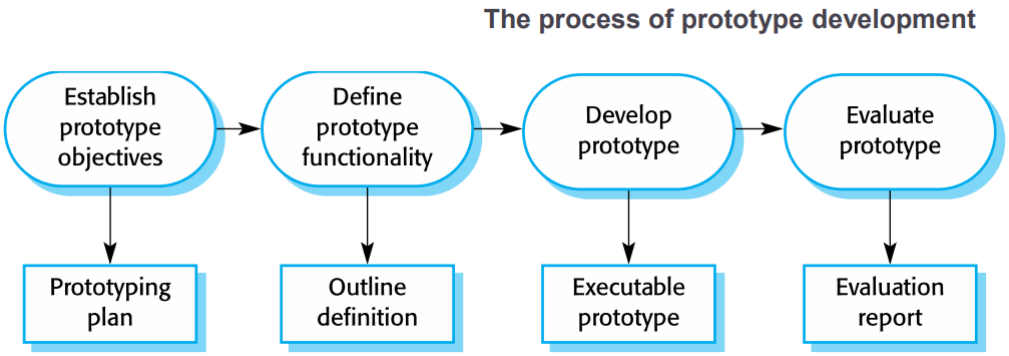
Prototip je početna verzija softvera koja se koristi za demonstriranje koncepata i isprobavanje opcija dizajna. Prototip se može koristiti u:

- u inženjeringu zahtjeva za pomoć u traženju i validaciji zahtjeva
- u procesima dizajna za istraživanje opcija i razvoj korisničkog interfejsa
- u procesu testiranja
- za obuku korisnika.

Prednosti izrade prototipa:

- poboljšana upotrebljivost sistema
- bliže podudaranje sa stvarnim zahtjevima korisnika
- poboljašava kvalitet dizajna

- poboljšana mogućnost održavanja
- smanjen napor pri razvoju.



Razvoj prototipa može se zasnivati na jezicima ili alatima za brzo pravljenje prototipa, te može izostaviti funkcionalnost. Prototip treba da se fokusira na oblasti proizvoda koje nisu dobro shvaćene. Provjera grešaka možda neće biti uključena u prototip. Prototip treba da se fokusira na funkcionalne, a ne na nefunkcionalne zahtjeve.

Prototipove se dijele na:

- prototipovi za bacanje (mock-away) – nakon razvoja ih treba odbaciti jer nisu dobra osnova za proizvodni sistem, npr. nemoguće podesiti sistem tako da ispuni nefunkcionalne zahtjeve
- evolutivni – prototipovi koji se ne odbacuju, već se koriste kao polazna osnova za razvoj ciljanog sistema.

26. MVC arhitekturni projektni obrazac.

MVC (Model-View-Controller) obrazac odvaja prezentaciju i interakciju od sistemskih podataka. Sistem je struktuisan u tri logičke cjeline koje međusobno djeluju.

- Komponenta Model – upravlja sistemskim podacima i povezanim operacijama nad tim podacima
- Komponenta View – definiše i upravlja načinom na koji se podaci predstavljaju korisniku.
- Komponenta Controller – upravlja interakcijom korisnika (pritisak tastera, klika miša) i prosljeđuje sve interakcije u View i Model.

Kada se koristi?

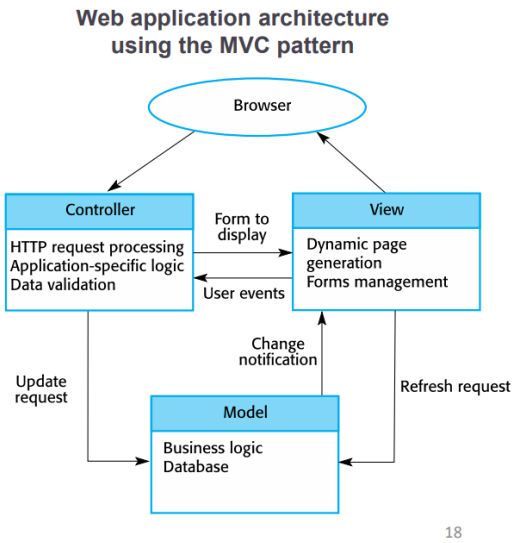
- Kada postoji više načina za pregled i interakciju sa podacima, takođe i kada su budući zahtjevi za interakciju i prezentaciju podataka nepoznati.

Prednosti?

- Omogućava da se podaci mijenjaju nezavisno od njihovog predstavljanja i obrnuto
- Podržava prezentaciju istih podataka na različite načine.

Mane?

- Može uključivati dodatan kod i složenost koda kada su model podataka i interakcija jednostavni.



27. Agile manifesto.

„Otkrivamo bolje načine razvoja softvera radeći to i pomažući drugima da to urade. Kroz ovaj rad svmo shvatili značaj: “

- **individualaca i interakcija umjesto procesa i alata**
- **radnog softvera umjesto sveobuhvatne dokumentacije**
- **saradnje sa klijentima umjesto pregovora oko ugovora**
- **odgovora na promjenu umjesto praćenja plana.**

Kent Bek i 16 ostalih priznatih ljudi iz SI zajednice.

Principi:

- uključivanje kupaca – kupac bi trebalo da bude blisko uključen kroz proces razvoja, njegova uloga je da daje prioritet novom sistemu zahtjeva i da procjeni iteracije sistema
- inkrementalna isporuka – softver se razvija u inkrementima uz specifikovanje zahtjeva, od strane korisnika, koji trebaju biti uključeni u sljedećem inkrementu
- ljudi, a ne proces – članove tima treba ostaviti da razviju sami svoje načine rada bez propisa procesa
- prihvatanje promjena – očekivati da će se sistemski zahtjevi mijenjati i treba dizajnirati sistem tako da se on prilagodi tim promjenama
- održavanje jednostavnosti – fokusirati se na jednostavnost i u softveru koji se razvija i u samom procesu razvoja, eliminisati kompleksnost iz sistema.

28. Klasifikacija sistemskih zahtjeva.

U softverskom inženjerstvu sistemski zahtjevi se klasifikuju na različite načine, ali neke uobičajene kategorije su:

- funkcionalni zahtjevi – šta sistem treba da radi i koje funkcije treba obavljati
- nefunkcionalni zahtjevi – specifične karakteristike sistema kao što su kvalitet, sigurnost, prilagodljivost

- zahtjevi sigurnosti – mjere koje se moraju poduzeti kako bi se osigurala sigurnost podataka i privatnost korisnika
- zahtjevi performansi – učinkovitost i brzina sistema, kao i sposobnost obavljanja zadataka u određenom vremenskom roku.

Razumijevanje i precizno definisanje sistemskih zahtjeva važan je korak u procesu dizajna i razvoju softvera.

29. Intervjuisanje kao tehnika za izmamljivanja zahtjeva.

Tehnike izmamljivanja – intervjui (formalni i neformalni, zatvoreni i otvoreni), upitnici, posmatranje, prototipovi, grupne radionice.

Formalni ili neformalni intervjui sa stakleholderima (zainteresovanim stranama) su dio većine procesa u inžinjeringu zahtjeva. Tipovi:

- zatvoreni – na osnovu unaprijed određene liste pitanja
- otvoreni – istražuju se različita pitanja sa stakeholderima.

Efikasno intervjuisanje:

- budite otvoreni, izbjegavajte unaprijed stvorene ideje o zahtjevima i budite voljni poslušati stakeholdere
- podstaknite sagovornika da pokrene diskusiju pomoću pitanja.

U praksi se najčešće koristi mješavina zatvorenog i otvorenog intervjua. Intervjui nisu dobri za razumjevanje zahtjeva domena, stručnjaci za aplikacije mogu da koriste jezik da opišu svoj rad koji inžinjeri zahtjeva ne razumiju.

Intervjui su dobri za sticanje opšteg razumijevanja šta zainteresovane strane žele i načina na koji bi zainteresovane strane mogle stupiti u interakciju sa sistemom.

Morate predlagati zahtjeve sistema, prije nego samo postavljati pitanja.

Mane intervjua?

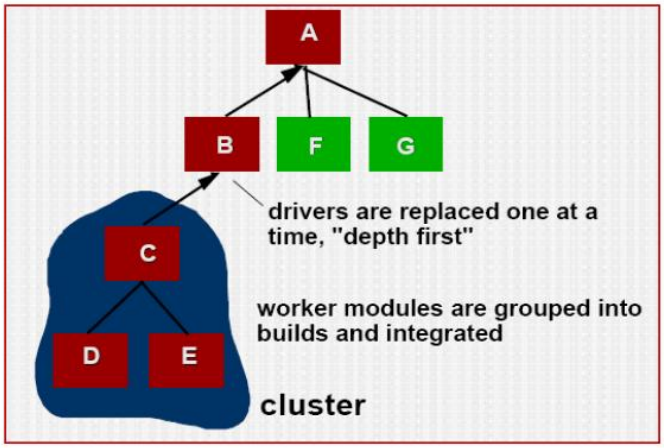
- stručnjaci za aplikacije mogu da koriste jezik da opišu svoj rad koji inženjeri zahtjeva ne mogu da razumiju
- nisu dobri za razumijevanje zahtjeva domena.

30. “Botton-up” strategija integracionog testiranja.

Kod ove strategije integracije, integracija i testiranje se kreće od najizvedenijeg modula i kreće se prema gore kroz kontrolnu hijerarhiju. Prednosti ovakvog pristupa su što on verifikuje podatke niskog nivoa rano u procesu testiranja, te nema potrebe za stubovima. Nedostaci su ti što moduli drajvera moraju biti napravljeni za testiranje modula nižeg nivoa (kod se kasnije proširuje i ubacuje u verziju sa punim funkcijama). Takođe upravljački programi ne sadrže kompletne algoritme koji će na kraju koristiti usluge

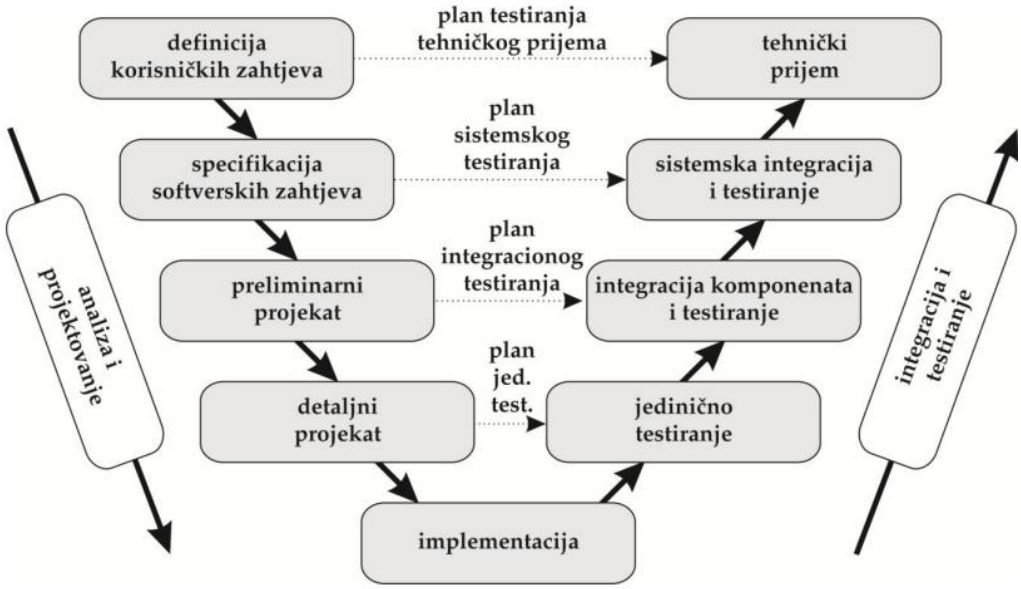
modula nižeg nivoa – testiranje može biti nepotpuno ili će možda biti potrebno više testiranja na višim nivoima.

1. Komponente niskog nivoa se kombinuju u klastere koji obavljaju određenu softversku podfunkciju.
2. Upravljački program je napisan da koordinira unos i izlaz testnog slučaja, nakon toga klaser je testiran.
3. Drajveri se uklanjaju i klasteri se kombinuju krećući se nagore u strukturi programa.



31. V-model softverskih procesa.

V-model obezbjeđuje sredstva za testiranje softvera u svakoj fazi na obrnuti način. U svakoj fazi kreiraju se planovi testiranja i test slučajevi kako bi se verifikovao i validirao proizvod u skladu sa zahtjevima te faze. Verifikacije i validacija idu paralelno. Ovaj model se takođe može nazivati modelom verifikacije i validacije.



32. Programiranje u paru.

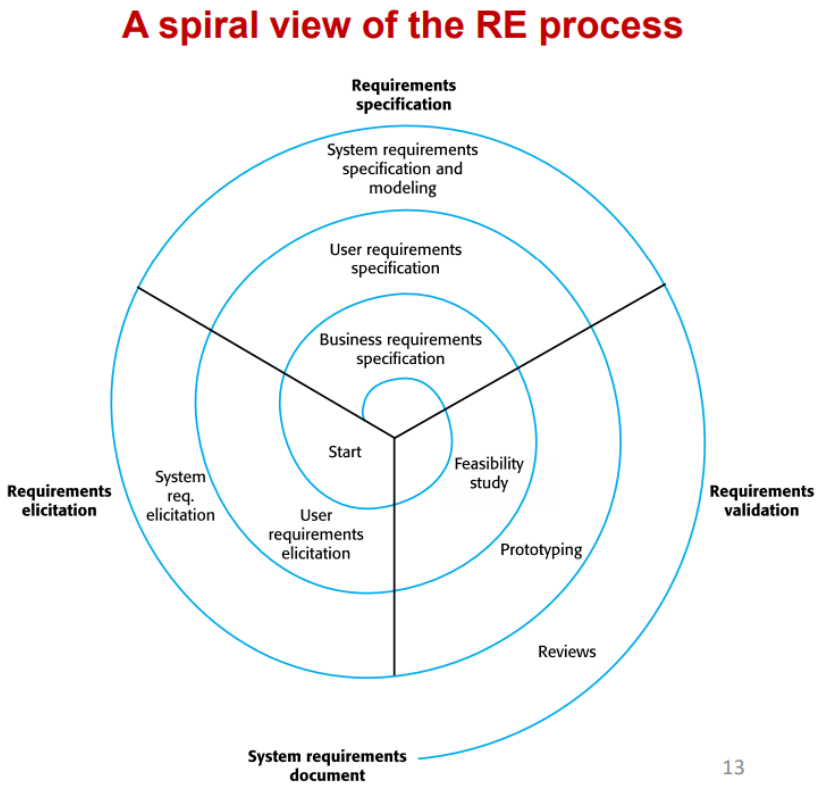
Programiranje u paru uključuje programere koji rade u parovima i zajedno razvijaju kod. Ovo pomaže u razvoju zajedničkog vlasništva nad kodom i širi znanje kroz tim. Služi kao neformalni proces pregleda jer svaki red koda pregleda više osoba. To podstiče refaktorisanje jer cijeli tim može imati koristi od poboljšanja sistemskog koda. Kod programiranja u paru, programeri sjede zajedno za istom računarom da bi razvili softver. Parovi se stvaraju dinamički, tako da tokom procesa razvoja, svi članovi tima rade jedni sa drugima. Razmjena znanja koja se dešava tokom programiranja u paru je veoma važna jer smanjuje ukupne rizike za projekat kada članovi tima odu. Programiranje u paru nije nužno neefikasno i postoje neki dokazi koji sugreši da je par koji programira zajedno efikasniji od dva programera koji rade odvojeno.

33. Spiralni model inženjeringa zahtjeva.

Procesi koji se koriste u inženjeringu zahtjeva uveliko variraju u zavisnosti od domena aplikacije, uključenih ljudi i organizacije koja razvija zahtjeve. Međutim postoji niz generičkih aktivnosti povezanih za sve procese, a to su:

- izmamljivanje
- analiza
- specifikacija
- validacija
- menadžment.

U praksi inženjering zahtjeva je iterativni proces u kome se ove aktivnosti prepliću.



34. Notacija dijagrama aktivnosti.

P-06: Dijagram aktivnosti – notacija dijagrama aktivnosti.

35. “4+1” model arhitekture sistema.

Svaki arhitektonski model prikazuje samo jedan pogled ili perspektivu sistema. Može pokazati:

- kako se sistem razlaže na module
- kako procesi tokom izvršavanja međusobno djeluju
- različite načine na koje se komponente sistema distribuiraju preko mreže.

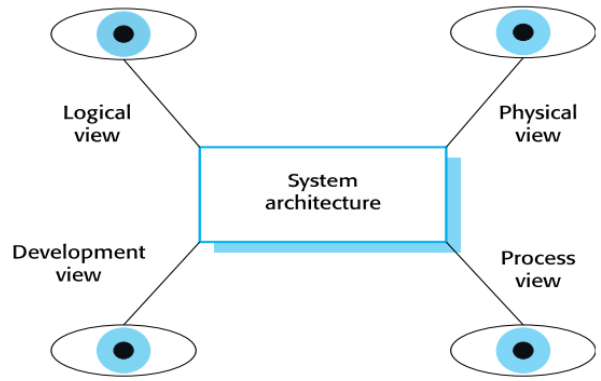
I za dizajn i za dokumentaciju, **obično moramo da predstavimo više pogleda na arhitekturu softvera**.

4+1 model:

- Logički prikaz – prikazuje ključne apstrakcije u sistemu kao objekte ili klase objekata
- Prikaz procesa – prikazuje kako je sistem, u toku rada, sastavljen od procesa koji međusobno djeluju
- Prikaz razvoja – prikazuje kako se softveer razlaže za development
- Fizički prikaz – prikazuje sistemski hardver i kako su softverske komponente raspoređene po procesima u sistemu.

Povezano korišćenje use cases ili scenarija.

4 + 1 view model of software architecture



36. Formalni tehnički pregled (review).

Ciljevi:

- da se otkriju greške u funkciji, implementaciji ili logici u svakoj reprezentaciji softvera
- da se provjeri da li softver ispunjava potrebne zahtjeve
- da se osigura da je softver predstavljen sa unaprijed definisanim standardima
- da se postigne softver koji je razvijen na jedinstven način
- da se projektom može lakše upravljati

Formalni tehnički pregled je u stvari klasa pregleda koja uključuje uputstva i inspekcije.

Sastanak:

- tipično 3-5 ljudi
- trebala bi se uraditi unaprijed priprema ali ne bi trebala uzeti više od dva sata svakom učesniku
- sastanak bi trebalo da bude kraći od 2 sata

- fokus bi trebalo da bude na radnom proizvod

Igrači:

- Proizvođač – pojedinac koji je razvio softver, obavještava vođu projekta da je radni proizvod završen i da je potrebno uraditi review
- Vođa review-a – procjenju spremnost proizvoda, generiše kopije materijala i distribuira ih do 2-3 revjuera za daljne pripreme
- Reviewer(s) – očekuje se da potroši 1-2 sata pregledavajući proizvod, pravljeći bilješke i da zna o čemu se radi
- Recorder(s) – recenzent koji zapisuje pismeno sve bitne stavke

Sprovođenje:

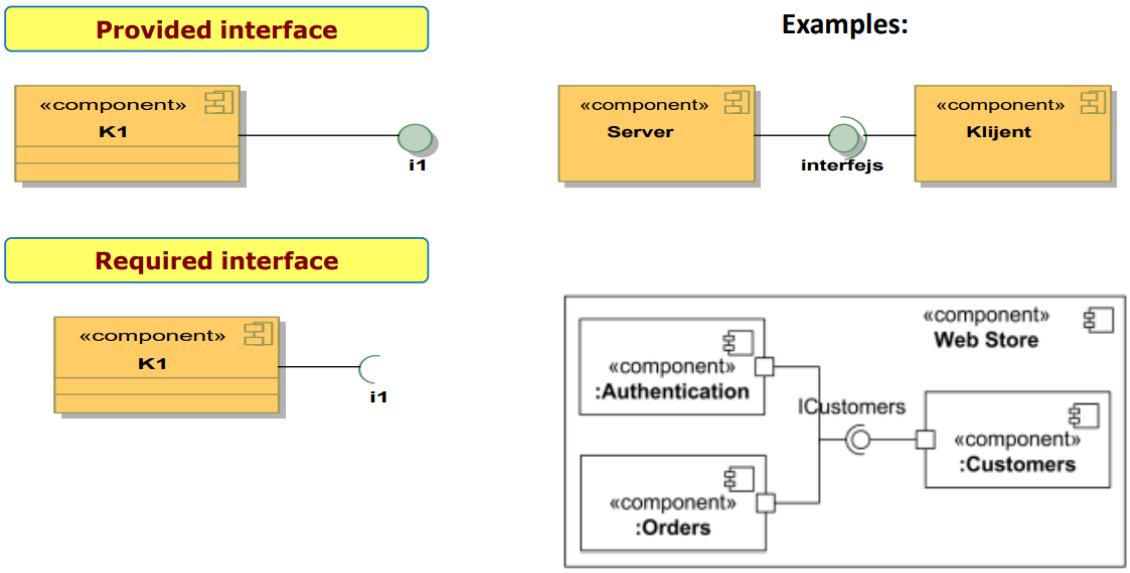
- Pregleda se proizvod, a ne proizvođač
- Treba se postaviti dnevni red i pridržavati ga se
- Treba ograničiti debatu i pobijanje
- Treba navesti problematična područja, ali ne i pokušati riješiti svaki problem
- Trebaju se voditi pisanje bilješke
- Ograničiti broj učesnika i insistirati na prethodnoj pripremi
- Razviti kontrolnu listu za svaki proizvod koji će biti pregledan
- Zakazati vrijeme i naći sve potrebne resurse
- Treba se provesti obuka za sve recenzente.
- Treba pregledati ranije recenzije.

37. Objasniti i ilustrovati modelovanje petlji u dijagramu sekvence.

P-08: Dijagrami interakcije.

38. Dijagram komponenti.

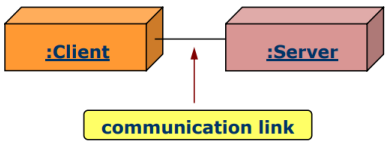
Pored dijagrama razmješanja, UML notacija za predstavljanje softverske arhitekture. Dijagram komponenti – predstavlja logičku arhitekturu softverskog sistema (vrijeme projektovanja/razvojni prikaz modela “4+1”). “Dijagram ožičenja softvera” – prikazuje komponente softvera i njihove veze. Veze komponenti = zavisnost. Interfejs = skup operacija koje obezbjeđuje/zahtjeva neka komponenta.



39. Dijagram razmještanja (Deployment diagram).

Takođe UML notacije za predstavljanje softverske arhitekture. Predstavlja primjenu softverskih artefakata na hardverskim čvorovima (vrijeme izvođenja/fizički prikaz modela "4+1"). Sadrži:

- čvorove – apstrakcija hardverskih čvorova (npr. server) i okruženja za izvršavanje (npr. operativni sistem)
- komunikacione veze – veze između čvorova



40. Slojeviti arhitektonski stil (obrazac).

Organizuje sistem u skup slojeva od kojih svaki pruža skup usluga za slojeve iznad. Jedan sloj pruža usluge sloju iznad njega tako da slojevi najnižeg nivoa predstavljaju osnovne usluge koje će vjerovatno služiti cijelom sistemu. Ovaj stil podržava inkrementalni razvoj podsistema u različitim slojevima. Kada se promjeni interfejs sloja, to utiče samo na susjedni sloj.

Kada se koristi?

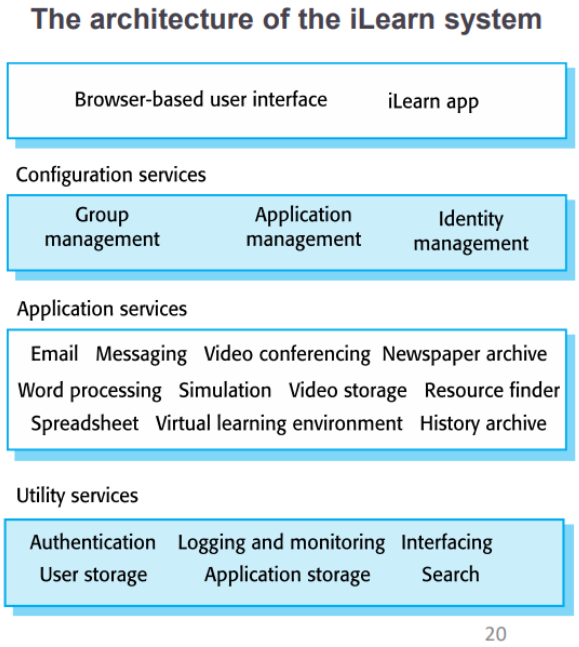
- Prilikom izgradnje novih objekata na osnovu već postojećih sistema, kada je razvoj raspoređen na nekoliko timova je svaki odgovoran za sloj funkcionalnosti

Prednosti?

- Omogućava zamjenu cijelih slojeva dok god se održava interfejs
- Suvišni objekti se mogu obezbjediti u svakom sloju da bi se povećavala pouzdanost sistema

Mane?

- U praksi, obezbjeđivanje čistog razdvajanja između slojeva je teško i sloj visokog nivoa će često morati da stupi u interakciju sa najnižim slojem direktno, a ne preko međuslojeva
- Performanse mogu biti problem zbog više nivoa interpretacije zahtjeva za uslugu dok se obrađuje na svakom sloju



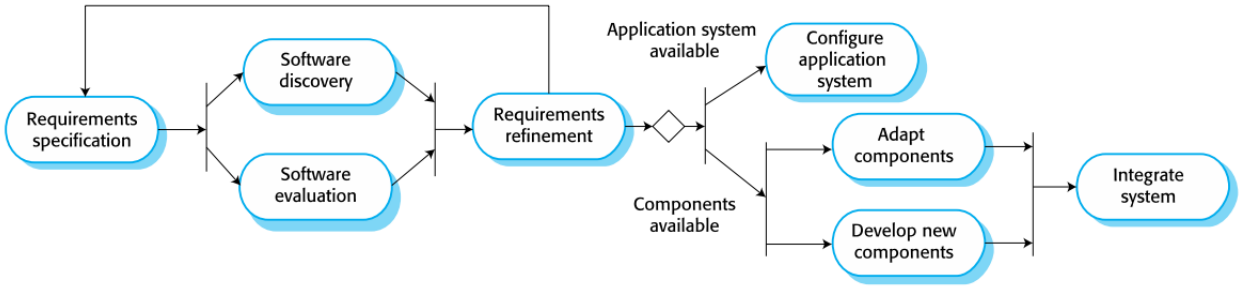
41. Softverski procesni modeli bazirani na integraciji i konfiguraciji.

Zasnovano na ponovnoj upotrebi softvera gdje su sistemi integrisani iz postojećih komponenti ili aplikativnih sistema. Ponovno korišćeni elementi mogu biti konfigurisani da prilagode svoje ponašanje i funkcionalnost zahtjevima korisnika. Ponovna upotreba je sada standardni pristup za izgradnju mnogih tipova poslovnih sistema.

Vrste softvera za višekratnu upotrebu:

- Samostalni sistemi aplikacija koji su konfigurisani za upotrebu u određenom okruženju
- Kolekcije objekata koji su razvijeni kao paket koji se integriše sa komponentnim okvirom kao što je .NET ili J2EE
- Veb servisi koji su razvijeni u skladu sa standardima usluga i koji su dostupni za daljinsko pozivanje.

Reuse-oriented software engineering



Prednosti?

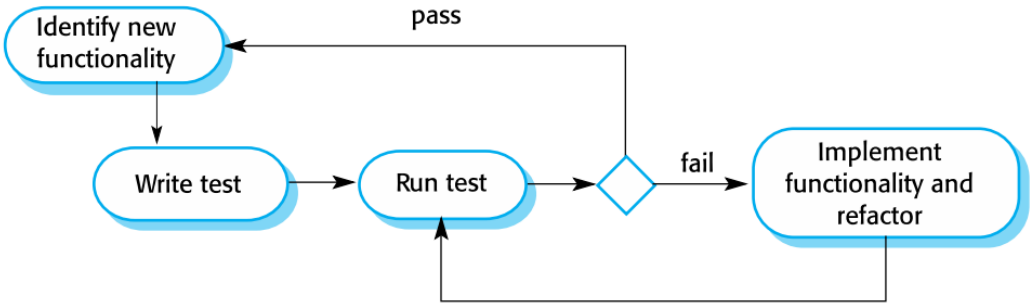
- Smanjeni troškovi i rizici jer se manje softvera razvija od nule
- Brža isporuka i implementacija sistema.

Mane?

- Kompromisi zahteva su neizbežni pa sistem možda neće zadovoljiti stvarne potrebe korisnika
- Gubitak kontrole nad evolucijom ponovo korišćenih elemenata sistema.

42. Razvoj softvera vođen testovima (Test-driven-development).

Test-driven development (TDD) je pristup razvoju programa u kojem se prepliću testiranje i razvoj koda. Testovi se pišu pre koda i „prolaženje“ testova je ključni pokretač razvoja. Razvijate kod postepeno, zajedno sa testom za taj prirast. Ne prelazite na sledeći korak dok kod koji ste razvili ne prođe test. TDD je uveden kao dio agilnih metoda kao što je ekstremno programiranje. Međutim, TDD se takođe može koristiti u razvojnim procesima vođenim planom.



TDD procesne aktivnosti:

- Počnite tako što ćete identifikovati inkrement funkcionalnosti koji je potreban. Ovo bi obično trebalo da bude mali inkrement i implementiran u nekoliko linija koda
- Napišite test za ovu funkcionalnost i implementirajte ovo kao automatizovani test
- Pokrenite test, zajedno sa svim ostalim testovima koji su implementirani
- Implementirajte funkcionalnost i ponovo pokrenite test

- Kada se svi testovi uspješno pokrenu, prelazite na implementaciju sledećeg dela funkcionalnosti.

Prednosti TDD-a:

- pokrivenost koda – cijeli kod ima barem jedan test
- regresija testiranje
- pojednostavljeno debagovanje – kada test padne, očigledno je gdje leži problem
- sistemska dokumentacija – testovi su na neki način dokumentacija, jer svaki test govori šta kod treba da radi.

43. Jedinično testiranje.

Jedinično testiranje je proces testiranja pojedinačnih komponenti u izolaciji. To je proces ispitivanja kvara.

Jedinice mogu biti:

- pojedinačne funkcije ili metode unutar objekta
- klase objekata sa nekoliko atributa i metoda
- kompozitne komponente sa definisanim interfejsima koji se koriste za pristup njihovoj funkcionalnosti.

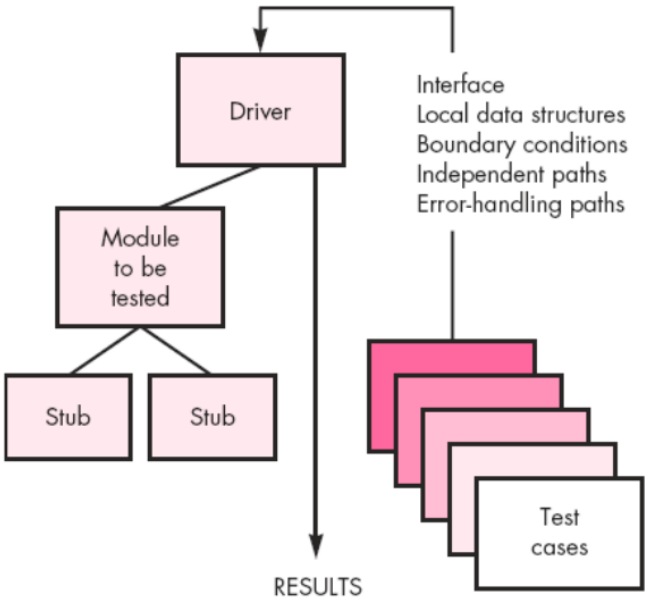
Koncentriše se na unutrašnju logiku obrade i strukturu podataka. Takođe, koncentriše se na kritične module i one sa visokom ciklomatskom složenošću kada su resursi za testiranje ograničeni.

Ciljevi za slučajeve jediničnih testova:

- interfejs modula – uvjerite se da informacije pravilno ulaze i izlaze iz modula
- lokalne strukture podataka – uvjerite se da podaci koji su privremeno uskladišteni održavaju svoj integritet tokom svih koraka u izvršavanju algoritma
- granični uslovi – uvjerite se da modul ispravno radi na graničnim vrijednostima koje su utvrđene da ograničavaju obradu
- nezavisne putanje (osnovne putanje) – putanja se koristi kako bi se osiguralo da su svi izrazi u modulu izvršeni barem jednom
- greška pri rukovanju putanjama – uvjerite se da algoritmi pravilno reaguju na specifične uslove greške.

Procedura jediničnog testiranja:

- drajver – jednostavan glavni program koji prihvata podatke test slučaja, prosljeđuje takve podatke komponenti koja se testira i štampa vraćene rezultate
- stubovi – služe za zamjenu modula koje poziva komponenta koja se testira
    - stub koristi tačan interfejs modula, može da obavlja minimalnu manipulaciju podacima, obezbjeđuje verifikaciju unosa i vraća kontrolu modulu koji je podvrgnut testiranju.



I drajveri i stubovi predstavljaju eng. „overhead“. Oba moraju biti napisana, ali ne čine dio instaliranog softverskog proizvoda.

Automatsko testiranje jedinica

Kad god je moguće, jedinično testiranje treba da bude automatizovano tako da se testovi izvode i proveravaju bez ručne intervencije. **U automatskom testiranju jedinica, treba koristiti okvir za automatizaciju testova** (kao što je **JUnit**) za pisanje i pokretanje testova programa. Okviri za testiranje jedinica pružaju generičke testne klase koje proširujete da biste kreirali specifične testne slučajeve. Oni tada mogu pokrenuti sve testove koje ste implementirali i izvjesiti o tome, često kroz neki GUI, o uspehu drugih testova.