**LEKCIJA 1**

1. Šta je to konstruisanje softvera?

Konstruisanje softvera je deo modela procesa razvoja softvera i odnosi se na neposredne aktivnosti čiji krajnji rezultat čine softverski proizvod, pomoćne datoteke, konfiguracione datoteke, podaci, dokumentacija za instalaciju i kofigurisanje softvera, uputstva, itd.

2. Navesti konkretne aktivnosti kod konstruisanja softvera?

Konkretne aktivnosti konstruisanja obuhvataju: verifikaciju zahteva, konstrukciono planiranje i detaljni dizajn softvera, verifikaciju arhitekture, dizajn i implementaciju klasa i potprograma, proveru ispravnosti koda, upravljanje verzijama izvornog koda, refaktorisanje, jedinično i integraciono testiranje, otkrivanje i ispravljanje grešaka, optimizacija algoritama i performansi, dokumentovanje izvornog koda, kreiranje konfiguracionih i pomoćnih datoteka, itd.

3. Šta su to IEEE softverski standardi?

IEEE softverski standardi su skup formalno definisanih procesa, metoda, notacija, nomenklatura, tehnika, rešenja, itd, koja se koriste u procesu razvoja softvera i softverski zavisnih sistema, i nastaju na osnovu akademskih istraživanja i kao rezultat prepoznate dobre prakse iz industrije.

4. Od kojih komponenti se sastoji naziv nekog standarda?

Što se tiče IEEE stanarda, naziv se formalno definiše iz sledećih informacija: broj dokumenta standarda (nnn), godine kada je standard usvojen (yyyy), i imena standarda. Ovi elementi se kombinuju prema sledećoj šemi u naziv standarda:  
  
IEEE Std <nnn>-<yyyy>, <naziv standarda>

5. Šta je to eksterna dokumentacija softvera?

Eksterna dokumentacija se odnosi na dokumentaciju i informacije van izvornog koda. U eksternim dokumentima je, na višem nivou apstrakcije, opisan softver iz perspektive dizajna na različitim nivoima, struktura modela, itd. Pored toga u eksterne dokumente spadaju i izveštaji nastali u procesu konstruisanja, uputstva za instalaciju i podešavanja, itd.

6. Navesti tipične softverske dokumente?

\* Sprecifikacija softverskih zahteva (SRS);  
\* Klasni model;  
\* Dizajn arhitekture i detaljni dizajn;  
\* Vremenski plan i raspored projekta;  
\* Plan obezbeđivanja kvaliteta, plan testiranja, i izveštaje o testiranju;  
\* Izvorni kod softvera;  
\* Uputstvo za instalaciju, konfiguraciju, i integraciju;  
\* Uputstvo za korišćenje; itd.

7. Navesti teoremu formatiranja?

"Dobar vizuelni raspored prikazuje logičku strukturu programa".

8. Koji su ciljevi dobrog struktuiranja programskog koda?

Cilj formatiranja programa je da se što bolje vizuelno prikaže logička struktura programa. Dobra vizuelna prezentacija programskog koda omogućuje nekom ko čita program da ga brzo i lako razume.

**LEKCIJA 2**

1. Navesti konvencije za imenovanje varijabli?

Potrebno je praviti razliku prilikom imenovanja klasa, imena instanci, imena potprograma, koristiti prefikse kao sto su "g\_" i "m\_" za globalne varijable i atribute klasa respektivno, koristiti velika slova za imana konstanti a reci u slozenim imenima odvojiti underscore znakom, koristiti konencije koje su specificne za odredjeni programski jezik ili neke od prihvacenih konvencija kao stu su madjarska notacija ili Camel case notacija, itd.

2. Koje su preporuke za inicijalizaciju varijabli?

Deklarisati i inicijalizovati varijable blizu mesta koriscenja, atribute klasa inicirati u konstruktrorima, izbegavati magicne vrednosti, odnosno treba koristiti imena konstanti, ako je potrebno varijable kao sto su brojaci ili koje se ponovo koriste treba reinicijalizovati pre mesta ponovne upotrebe, voditi racuna o dinamickoj alokaciji memorijskog prostora i oslobadjati rezervisani prostor, uzeti u obzir upozorenja prevodioca u vezi varijabli, proveravati vrednosti stvarnih argumenata potporgrama, obratiti paznju na bocne efekte kod argumenata koje se prenose po referenci

3. Šta su to globalne varijable?

Globalne varijable su varijable ciji je opseg vazenja i dostupnosti na nivou modula ili aplikacije.

4. Navesti preporuke za upotrebu globalnih varijabli?

Upotrebu globalnih varijabli treba izbegavati jer dovode do mogucnosti za pojavu gresaka, posebno ako se radi o programima koji koriste niti. Ako se koriste, globalne varijable treba dobro dokumentovati, koristiti prefiks u imenu varijable kojim se naznacava njen glopalni karakter, enkapsulirati globalne varijable u singleton instance sa metodom za pristup kojom bi se obezbedio thred-safe pristup, itd.

5. Objasniti preporuke za sekvencijalne instrukcije?

Preporuka je da se sekvence instrukcija grupisu u blokove prema medjusobnoj zavisnosti ili logickoj povezanosti, refaktorisati kod ako postoji preklapanje izmedju blokova, definisati nove potprograme ako se odredjene sekvence instrukcija ponavljaju vise puta.

6. Kako se utvrđuje međuzavisnost pojedinih podprograma?

Za jasno oznacavanje medjuzavisnosti potprograma se mogu koristiti komentari, imena potprograma kojima se naznacava svrha, parametri koji se prosledjuju potprogramima i njihova svrha mogu da ukazu na zavisnost, mogu se koristiti varijable za cuvanje statusne vrednosti i njihova provera.

7. Koje vrste if-instrukcija postoje?

\* if-then - jednostavni uslovni blok,  
\* if-then-else - uslovni blok sa alternativnim tokom koji ce se izvrsiti ako primarni uslov nije ispunjen,  
\* if-then-else-if - konstrukcija sa vise uslovnih tokova gde i alternativni tokovi zavise od ispunjenosti uslova.

8. Opisati preporuke za upotrebu if-instrukcija?

Pri formiranju konstrukcije prvo navesti primarni uslovni tok, a zatim alternativne tokove; obratiti paznju na relacione operatore u uslovu jer mogu da promene tok izvrsenja u nezeljenom smeru; komplikovane usove treba pojednostaviti funkcijama koje vracaju Bulovu vrednost i njih koristiti na mestu provere uslova

9. Navesti tipične greške kod petlji?

Pogresna inicijalizacija i provera uslove; pogresan ulaz ili izlaz iz petlje; preklapanje blokova (pogresno ugnjezdavanje); pogresan izbor petlje; konstruisanje petlje sa dugackim blokom koje nisu citljive; neoprezna manipulacijama vrednostima indeksa.

10. Navesti preporuke za petlje?

Koristiti male blokove; ako je blok veliki deo se moze zameniti potprogramom radi pojednostavljenja; koristiti do tri nivoa ugnjezdavanja; razmotriti zamenu petlje sa vise izlaza duzom petljom sa jednim izlazom; neke petlje se mogu zameniti rekurzivnim funkcijama (tail recusion).

11. Šta je to duboko ugneždavanje?

Duboko ugnjezdavanje je preterano umetanje konstrukcija sa blokovima u druge blokove (preko 3-4 nivoa).

12. Objasniti tehniku dubokog ugneždavanja?

Ako se radi o uslovnim iskazima, ugnjezdeni iskazi se mogu spojiti kombinovanjem uslova; treba razmotriti upotrebu switch-case iskaza; zavisni i logicki povezani sekvencijalni iskazi u ugnjezdenim blokovima se mogu refaktorisati u potprograme; restruktuiranje ugnjezdenih if-then konstrukcija u if-then-else-if konstrukcije.

13. Navesti koje sve vrste kontrolnih instrukcija postoje?

\* uslovna grananja: if-then, if-then-else, if-then if-then-else-if, switch-case;  
\* bezuslovno grananje;  
\* pozivanje potprograma;  
\* prekid i povratak iz potprograma;  
\* iteracije: for, foreach, while, do-while

14. Kako se iz programa može eliminisati instrukcija "goto"?

U zavisnosti od mesta i konteksta upotrebe, moze se eliminisati uslovnim iskazima, petljama, i pozivom potprograma.

15. Opisati višestruki izlaz iz petlje?

Pod visestrukim izlaskom iz petlje spada mogucnost koriscenja instrukcija break i continue za konstrolu toka izvrsavanja iteracije. U kombinaciji sa uslovnim iskazima koji odredjuju uslove, break se moze koristiti za totalni prekid petlje, a continue za prekid trenutne iteracije i prelazak na sledecu iteraciju. Broj break i continue poziva moze biti visestruk, a mogu se koristiti i u kombinaciji sa osnakama u slucaju da se pozivaju iz ugnjezdenih petlji kako bi se odredlo to na koju se petlju odnosi poziv.

16. Opisati višestruki izlaz iz podprograma?

Visestruki izlazak is potprograma se odnosi na mogucnost izlaska iz potprograma pre nego sto se izvrse svi iskazi iz potprograma i dosegne njegov kraj. Regularan nacin izlaska je upotreba return naredbe koja u kombinaciji sa uslovnim iskazima moze da kontrolise tok izvrsenja i povratka iz potprograma. Pored toga se moze izaci i emitovanjem izuzetka, koji ce prekinuti potprogram, ali i sam program u slucaju da nije obradjen na visem nivou, ili koriscenje assert naredbi koje se opet odnose na prekid kompletnog programa u slucaju da uslov koji se proverava nije zadovoljen.

17. Objasniti koncept strukturnog programiranja?

Strukturno programiranje je paradigma programiranja koja se bazira na ideji da se svaki program moze kreirati upotrebom sledeca tri gradivna bloka: sekvencom instrukcija, konstrukcijama grananja, i ponavljanjima (iteracijama). Cilj je pojednostavljenje izvornog koda cime se povecava kvalitet, smanjuje broj gresaka, kod se lakse testira, krace je vremen implementacije

18. Navesti preporuke kod strukturnog programiranja?

Preporuke strukturnog programiranja su:  
\* koristiti petlje sa jednim izlazom i ulazom,  
\* izbegavanje visestrukih izlaza u petljama i potprogramima, nekoriscenje goto naredbe,  
\* organizacija zavisnih i logicki povezanih iskaza u blokove,  
\* koriscenje potprograma radi pojednostavljenja koda, itd.

19. Šta je to "pseudokod"?

Pseudokod je pojednostavljeni opis koraka, odnosno algoritma, koji se treba zivrsiti, i moze biti napisan nekom formalnom notacijom koja nije zavisna od sintakse nekog programskog jezika i hardverske platforme. Na osnovu kavo specificiranog opisa koraka, kodiranje se moze izvrsiti u odabranom programskom jeziku i dobiti izvrsna verzija programa.

20. Šta je to kompjuterska procedura?

Kompjuterska procedura ili potprogram je deo programa, odnosno funkcionalni modul koji sadrzi instrukcije potrebne za izvrsenje jednog zadatka na racunaru. Kombinovanje potprograma kao gardivnog elementa i kompleksnije celine dobija se konacni program za resavanje nekog problema. Prednosti koriscenja potprograma su laksa implementacija i odrzavanje, lakse testiranje, mogucnost odranizovanja u biblioteke koje se mogu ponovo koristit u drugim projektima, pojednostavljenje izvornog koda

21. Kako se meri kompleksnost podprograma?

Kompleksnost programa se meri kao suma sledcih tezina: 1 za osnovnu putanju, kljucne reci, i svaku case opciju u switch konstrukciji, zatim se dodaje broj instrukcija u programu, broj ulaznih parametara i izlaza, i broj i nivo ugnjezdenih blokova.

22. Koje su preporuke u vezi sa kompleksnim podprogramima?

Nivo kompleksnosti potprograma treba da je ispod 10, u suprotnom potprogram treba refaktorisati u manje potprograme koji zadovoljavaju ovaj uslov.

**LEKCIJA 3**

1. Navesti aktivnosti kod životnog ciklusa softvera?

\* Analiza izvodljivosti;  
\* Analiza trzista;  
\* Analiza domena;  
\* Prikupljanje, specifikacija i validacija zahteva;  
\* Planiranje projekta;  
\* Dizajn sistema;  
\* Implementacija;  
\* Testiranje;  
\* Isporuka i integracija; i  
\* Odrzavanje.

2. Šta su to modeli životnog ciklusa?

Modeli zivotnog ciklusa softvera predstavljaju konceptualne radne okvire koji definisu skup aktivnosti potrebnih da bi se realizovao softverski proizvod. Medju najpoznatijim modelima su linearni sekvencijalni metod, prototipski model, inkrementalni model, spiralni model, agilne metode.

3. Koje su to agilne metode razvoja softvera?

\* Rational Unified Process (RUP);  
\* Rapid Application Development (RAD);  
\* Ekstremno programiranje (XP);  
\* Scrum;  
\* Kanban, itd.

4. Uporediti agilne i tradicionalne metode razvoja softvera?

Agilne metode odstupaju od formalnih procesa tradicionalnih modela razvoja softvera i prednost daju kolaboraciji izmedju korisnika i clanova razvojnog tima. Isporuka proizvoda krajnjim korisnicima kod agilnih metoda se bazira na:  
\* adaptivnom planiranju u toku razvoja na racun planiranja unapred pre pocetka razvoja;  
\* iterativni i evolutivni razvoj;  
\* cesta isporuka nove verzije (posle svake iteracije);  
\* funkcionise sa malim timovima;  
\* fleksibilnost na ceste izmene zahteva u saradnji sa korisnicima; itd.

5. Navesti aktivnosti kod konstruisanja softvera?

\* Detaljan dizajn arhitekture i modelovanje sistema;  
\* Kodiranje resenja;  
\* Pisanje i pokretanje jedinicnih testova;  
\* Otkrivanje, analiza, i ispravljanje defekata;  
\* Integraciono testiranje modula i komponenti; i  
\* Verifikacija resenja.

6. Navesti alate koji se koriste kod konstruisanja softvera?

\* Alati za dizajn i modelovanje (Visual Paradigm, Power Designer, itd);  
\* Integrisana razvojna okruzenja (NetBeans, Eclipse, Visaul Studio, Xcode, itd);  
\* Alati za testiranje;  
\* Sistemi za upravljanje i kontrolu izvornog koda (Git, Subversion, CVS, itd);  
\* Sistemi za kontinuiranu integraciju (Jenkins, GitLab, itd); itd.

7. Šta je to UP?

UP (Unified Process) je proces razvoja softvera koji se oslanja na objektno-orjentisani dizajn i primenu UML jezika za dizajn i modelovanje softvera. Sastoji se od cetiri faze: pocetak, elaboracija, konstrukcija, i tranzicija).

8. Opisati modele kod UP?

\* Use-case model - model opisuje kako korisnici interaguju sa sistemom;  
\* Modeli analize - dijagrami aktivnosti, sistem sekvencijalni dijagrami, dijagrami stanja, itd;  
\* Modeli dizajna - dijagrami paketa i klasa, sekvencijalni dijagrami, ER dijagrami, itd;  
\* Modeli razmestanja - dijagram komponenti;  
\* Modeli implementacije - dijagram komponenti i podsistemal;  
\* Test model.

9. Objasniti izbor modela životnog ciklusa?

Model zivotnog ciklusa se bira prema tipu softverskog sistema, velicini projekta, vremenu za realizaciju, raspolozivom budzetu, profilu razvojnog tima, utvrdjenom riziku, postojanoscu i dinamikom promena zahteva, itd. Primera radi, razvoju softvera za ugradjene sisteme vise odgovaraju formalniji tradicionalni modeli sa detaljnim planiranjem i analizom pre pocetka razvoja. Agilne metode se lako mogu aplicirati kod web aplikacija i slicno.

10. Opisati pripreme za konstruisanje?

Pre same faze konstrukcije, potrebno je ustanoviti precizne zahteve softvera jer je nedostatke najjeftinije resiti za vreme formulisanja zahteva. Posle specifikacije i validacije zahteva uspostavlja se plan razvoja projekta. Nakon toga sledi detaljna analiza i dizajn arhitekture sistema, i ova vaza takodje predstavlja kriticnu fazu jer se odluke vezane za arhitekturu teze ispravljaju u toku konstruisanja, ili jos gore nakon toga. Kada su poznati svi detalji vezani za dizajn, moze se zapoceti sa konstrukcijom.

11. Navesti listu ključnih odluka na početku konstruisanja?

\* Izbor programskog jezika, verzije, i paradigme koju podrzava, na osnovu tipa softverskog sistema koji se gradi, znanja i iskustva clanova razvojnog tima, itd;  
\* Izbor radnih okvira, biblioteka, i komponenti, na osnovu arhitekture sistema, itd;  
\* Izbor alata za specifikaciju modela i arhitekture sistema, razvojnog okruzenja, itd;  
\* Definisanje parametara razvojnog procesa kao sto su organizacija tima, model po kome se vrsi inspekcija koda, metrika koje ce se koristiti za ocenjivanje, utvrdjivanje rizika, itd;  
\* Definisanje plana testiranja;  
\* Definisanje procedura integracije i instalacije, itd.

12. Opisati ukratko osobine pojedinih programskih jezika?

C je proceduralni programski jezik opste namene. Zbog svojih osobina se cesto koristi u sistemskom programiranju, programiranju softvera za ugradjene sisteme, itd.  
  
C++ je jezik koji je nastao kao prosirenje programskog jezika C, dodavanjem objektno-orijentisane paradigme i funkcionalnog programiranja (lambda funkcije). Sama osnova jezika je prosirena takozvanom STL (Standard Template Library) bibliotekom koja dopunjuje sam jezik mocnom bibliotekom generickih struktura i algoritama. Jezik ima siroku primenu u mission critical sistemima, sistemskom programiranju, razvoju desktop aplikacija, a poslednje vreme i u implementaciji HPC softvera za multiprocesorske sisteme i GPU akceleratore.  
  
Java je objektno-orjentisani programski jezik opste namene. Postoje tri varijante: Java SE (Standard Edition) koji se deli na JDK (Development Kit) i JRE (Runtime Environment). Pored ove verzije postoji i Java ME (Micro Edition) za ugradjene sisteme. Na kraju verzija Java EE (Enterprise Edition) je prosirenje Java SE infrastrukturom za projektovanje i izgradnju velikih poslovnih sistema. Java se takodje koristila kao primarni jezik za razvoj Android softvera, ali je od 2019. godine zamenjen jezikom Kotlin (iako Java i dalje postoji kao opcija za razvoj ovog tipa aplikacija).  
  
Python je objektno-orjentisan i funkcionalan multiplatformski skript jezik koji se najvise koristio za web i internet aplikacije, ali je u poslednje vreme zauzeo veliko mesto primene u oblasti nauke o podacima, masinskog ucenja, itd.

13. Koje vrste projekata postoje?

Prema razlozima za pokretanje projekta, projekti mogu biti:  
\* projekti koji resavaju novi problem,  
\* projekti koji unapredjuju postojeca resenja, i  
\* projekti koji dopunjuju postojeca resenja.  
  
Prema korisniku, razvoj projekta moze biti:  
\* razvoj za potrebe jednog korisnika ili organizacije,  
\* razvoj za odredjeno trziste i njene potrebe,  
\* razvoj za siroku upotrebu.  
  
Prema velicini projekti mogu biti: kratki (do par meseci), srednje velicine (do godinu dana), i veliki projekti (vise godina).  
  
Softverski projekti se takodje mogu podeliti i prema dostupnosti pa mogu biti: besplatni, licencirani, sa otvorenim kodom, komercijalni sa ogranicenom besplatnom verzijom (trial, ogranicene funkcije, itd), itd.

14. Uporediti produktivnost na malim i velikim projektima?

Produktivnost na manjim projektima moze biti znatno veca nego na vecim. Manje projekte radi manji broj ljudi, na njima se mogu primeniti agilne metode razvoja koje ubrzavaju implementaciju, komunikacija je jednostavnija i brza, brze su vidljivi rezultati, i slicno. Kod projekata veceg obima razvoj rade veci timovi ili vise timova sto otezava komunikaciju i koordinaciju, Na vecim projektima se na pocetku trosi znatno vece vreme za pripremu, formulisanje zahteva, dizajn i planiranje, i druge aktivnosti koje nisu direktno vezane za samo konstruisanje proizvoda. Svi ovi faktori uticu na to da je kod manjih projekata moguce ostvariti veci broj linija izvodnog koda po jedinici vremena u odnosu na trajanje projekta, za raziku od projekata srednje i vece velicine.

15. Opisati tehnike motivacije programera?

\* Organizovati rad u parovima;  
\* Organizovati neformalnu ili formalnu inspekciju koda;  
\* Uspostaviti sistem nagradjivanja zaposlenih;  
\* Jacati veze izmedju clanova tima i osecaja pripadnosti timu;  
\* Omoguciti edikaciju i usavrsavanja zaposlenih;  
\* Omoguciti clanovima tima da ucestvuju u diskusijama i daju svoje misljenje o prokektu

16. Kako se procenjuje veličina projekta?

Jedan nacin je upotreba formalnih metoda kao sto je COCOMO 2. Procena velicine se moze dobiti i upotrebom alata za upravljanje projektima. Dalje, mogu se angazovati spoljasni saradnici i inzinjeri koji su specijalizovani za vrsenje procena, ili procenu mogu dati clanovi tima za razvoj projekta na osnovu svog ranijeg iskustva.

17. Opisati tehniku upravljanja konfiguracijom?

Upravljanje konfiguracijom je proces koji se sastoji od identifikovanja artifakata i upravljanja promena artifakata projekta, radi ocuvanja konzistentnosti, funkcionalnih i fizickih atributa, zahteva i dizajna projekta, tokom njegovog zivotnog ciklusa. Artifakti kod softverskih projekata mogu biti zahetvi, dizajn, izvorni kod, pomocne datoteke, itd. Upravljanje promenama ovih artifakata obezbedjuje se da projekat ostane unutar zadatih granica.

18. Šta je to version-control softver?

Version Control softver, ili SCMS (Source Control Management System) je softver kojim se cuva i upravlja procesom promena izvornog koda. Osnovne funkcije koje ovaj softver obezbedjuje je repozitorijum za cuvanje izvnrnog koda, pracenje istorije promena, razresavanja konflikta, kolaboracija izmedju clanova tima, itd. Prema nacinu cuvanja izvornog koda, mogu biti centralizovani (Subversion) ili distribuirani sistemi kao sto je Git. SCMS softver se lako integrise sa drugim alatima kao sto su razvojna okruzenja, softver za upravljanje projektima, sistemi za kontinuiranu integraciju, itd, i njegova korist ne nemerljiva, nevezano za velicinu tima ili projekta.

**LEKCIJA 4**

1. Šta su to "refaktoreri"?

"Refaktoreri" su specijalizovani samostalni programi ili funkcija integrisana u razvojna okruzenja koja se koristi za izmene izvornog koda u sta spadaju izmene simbola (imena klasa, metoda, funkcija, parametara, itd), restruktuiranje koda prema unapred definisanim plravilima (zamena delova izvornog koda prema utvrdjenoj dobroj praksi, paternima, itd), ekstrakciju delova izvornog koda u potprograme, zamenu tipova i broja formalnih argumenata potprograma, itd. Refaktoreri omogucavaju da se ove izmene izvrse na bezbedan nacin eliminisuci potencijalne greske koje bi nastale ako se ove izmene vrse rucno.

2. Opisati rečnike podataka?

Recnik podataka je katalog koji sadrzi strukture i ogranicenja podataka koji su od znacaja za projekat ili imaju neki uticaj na projekat.

3. Navesti vrste CASE alata?

CASE alati se mogu podaliti na:  
\* Alati (tools) - koji omogucavaju specificne pojedinacne zadatke u procesu projektovanja, implementacije, i odrzavanja softvera;  
\* "Radne klupe" (workbench) - podrzavaju jednu ili vise aktivnosti procesa razvoja softvera integracijom vise alata u koherentnu celinu;  
\* Okruzenja - podrzavaju sve ili vecinu aktivnosti procesa razvoja softvera integracijom funkcija prethodne dve grupe.

4. Navesti koji su to alati za konstruisanje softvera?

\* Alati za kodiranje izvornog koda su osnovni alati u procesu razvoja softvera. Ova alati omogucavaju pisanje izvornog koda i veliki broj funkcija kao sto su: automatsko kompletiranje koda na osnovu jezika koji se koristi, proveru gresaka u toku pisanja koda, refaktorisanje, navigaciju kroz izvorni kod, itd.  
  
\* Alati za otklanjanje gresaka, mogu biti samostalne aplikacije ili integrisani u razvojna okruzenja. U ove alate spadaju: debugger-i koji se koriste za pracenje toka izvrsenja programa, stanja i promena vrednosti, itd; alati za staticku proveru izvornog koda; itd.  
  
\* Alati za dizajn su alati koji takodje mogu biti samostalni i tako se najcesce koriste, ili integrisani u razvojna okruzenja, i koriste se za projektovanje i konstruisanje softvera primenom nekog od formalnih modela kao sto su UML dijagrami, ER dijagrami, itd. Modeli koji se kreiraju ovim alatima pojasnjavaju strukturu, ponasanje, i zavisnosti elemenata softverskog sistema koji se projektuje. Mogu imati i opciju automatskog generisanja inicijalnog koda na osnovu modela,a na primer klasa iz UML klasnog dijagrama, DDL koda za kreiranje tabela u bazi podataka na osnovu ER dijagrama, itd.  
  
\* Alati za upravljanje verzijama izvornog koda kojise koriste za cuvanje, upravljanje i pracenje izmena izvnornog koda tokom razvoja. Posto posreduju u procesu izmena izmedju vise programera, imaju i kolaborativnu funkciju. Mogu biti samostalni ili integrisani u razvojno okruzenje. Osnovne funkcije koje obezbedjuju su: repozitorijum za cuvanje izvornog koda koji moze biti centralizovan ili distribuiran, istorija promena, spajanje konrkurentnih promena i razresavanje konflikata, razvoj u vise paralelnih tokova, uporedni prikaz izmena kroz vreme, itd.

5. Navesti neke UML alate?

\* Visual Paradigm - komercijalna licenca i besplatna community sa ogranicenim mogucnostima;  
\* Power Designer - komercijalna licenca;  
\* Visual Studio;  
\* Microsoft Visio;  
\* Argo UML - besplatan alat;  
\* Star UML - komercijalna licenca; itd.

6. Opisati detaljno jedan UML alat?

Visual Paradigm je alat koji je dostupan varijanti sa komercijalnom licencom, i u besplatnoj varijanti za upotrebu u nekomercijalne svrhe. Komercijalna verzija ima nekoliko varijanti: osnovna (Modeler), Standard, Professional, i Enterprise. U zavisnosti od varijane varira i cena i skup mogucnosti koje su ukljucene. Ovo okruzenje podrzava sve znacajnije modele: UML, SysML, BPMN, ER, DFD, TOGAF, ArchiMate, SoaML. Pored toga se moze koristiti i za rad sa korisnickim zahtevima kroz User Story i Use Case modele. Druge mogucnosti su kreiranje inicijalnog koda iz modela, ili reverzni inzinjering, odnosno kreiranje modela iz izvornog koda. Moze se integrisati sa svim poznatijim razvojnim okruzenjima kao sto su Eclipse, NetBeans, IntelliJ, i Visual Sturio. Besplatna verzija nudi samo kreiranje UML modela. Alat je nezavisan od operativnog sistema i moze se koristiti na Linux, MacOS, i MS Windows sistemima.

7. Šta su to softverske skele?

Softverske skele su pomocne klase ili potprogrami koji oponasaju odredjenu funkcionalnost u svrhu testiranja drugih klasa ili potprograma, a kojima je za funkcionisanje neophodna logika koja jos nije implementirana i simulira se kroz softverske skele. Jedna od mogucnosti je rucno kreiranje simulirane funkcionalnosti, ali se mogu koristiti i pomocne biblioteke u kombinaciji sa test radnim okvirima kojima se olaksava simulacija implementacije, na primer koriscenje JUnit i Mockito biblioteke.

8. Šta je to baza podataka greški?

Baza podataka o greskama je kolekcija informacija o greskama formirana na osnovu prethodno identifikovanih i potvrdjenih gresaka. Na osnovu ovako formiranog skupa informacija moguce je kasnije otkrivanje istih ili slicnih gresaka, provera ispravki ranijih gresaka, formulisanje slucajeva testiranja, staticka analiza koda.

9. Navesti alate u Visual Studio-u?

Osnovni alati koje obihvata VS su:  
\* Code-editor;  
\* Inetegrisani debugger;  
\* Forms Designer - alat za vuzuelno kreiranje GUI aplikacija;  
\* Class Designer;  
\* Web Designer; itd.  
  
UML podrska je od verzije 2017 zamenjena novim alatom koji se zove Code Maps.

10. Šta je to GUI builder?

GUI builder je vizuelno okruzenje za kreiranje korisnickog interfejsa po WYSIWYG principu. Ovaj alat sadrzi takozvani dizajner formi u kome se elementi aplikacija (prozori, dijalozi, fragmenti, itd) kreiraju razmestanjem kopononenti. Ovicno su deo nekog razvojnog okruzenja, ali mogu biti i samostalne aplikacije, i povezani su sa code editorom za implementaciju logike koja je povezana sa dogadjajima registrovanim za odredjenu komponentu sa kojom ce korisnik interagovati u aplikaciji.

11. Opisati alat Junit?

JUnit je radni okvir za pisanje jedinicnih testova u programskom jeziku Java. Jedan test slucaj je predstavljen klasom koja sadrzi jednu ili vise test metoda, sa opcionim metodama za inicijalizaciju test slucaja i postavljanja pocetnog stanja, i metodama koje se izvrsavaju nakon izvrsenja testova a koriste se za resetovanje stanja ili oslobadjanje zauzetih resursa. Vise klasa koje implementiraju srodne test slucajeve se mogu grupisati u takozvane test suite grupe. Za pokretanje testova se koristi mehanizam koji se zove test runner, a pokretanje se moze izvrsiti unutar razvojnog okruzenje, ili preko alata za upravljanje zavisnostima i produkciju izvrsne datoteke (maven, ant, gradle). Sam radni okvir se moze prosirivati i kombinovati sa drugim bibliotekama kao sto su na primer biblioteke za programske skele (mock objekti), ili za testiranje neke specificne klase aplikacija.

12. Opisati alat FindBugs?

FindBug je alat koji sluzi za staticku analizu koda i pronalazenje gresaka na osnovu baze podataka o greskama. Alat moze da se koristi samostalno, integrisan u razvojno okruzenje, preko alata kao sto su maven, ant, gradle, itd. Tipove gresaka koje alat moze da prepozna spadaju u neku od sledecih kategorija: losa praksa, ispravnost, sigurnost, maliciozni kod, performanse, stil pisanja, itd.

**LEKCIJA 5**

1. Objasniti od čega se sastoji sistematski prilaz razvojnom testiranju?

On testira svaki relevantni softverski zahtev, svaki problem uocen u fazi dizajna, svaku liniju programa pomocu baznog testiranja, testira tok podataka, i koristi listu provere, u kojoj se nalaze greske iz proslosti.

2. Opisati koje su nesavršenosti razvojnog testiranja?

Pre svega, testovi ne mogu ukazati na sve potencijalne greske, vec samo mogu da potvrde prisustvo gresaka koje su anticipirane i obuhvacene testiranjem. Cesto se veruje da su testovima pokrivene sve mogucnosti nastanka defekata, ali je u realnosti pokrivenost tetovima nedovoljna. Dalje, cesto se vrsi testiranje samo iz jednog ugla, npr. vrsi se bazno testiranje dok su ostali tipovi testiranja zanemareni. Takodje je nedostatak u testiranju koje se vrsi samo na ispravne ulaze, dok se neispravni ulazi zanemaruju, ne testiraju se granicne vrednosti

3. Koje vrste strukturnog testiranja postoji?

Testiranje svake instrukcije  
Testiranje svake grane u programu  
Testiranje svake putanje

4. Uporediti pojedine vrste strukturnog testiranja?

Testiranje svake instrukcije se vrsi kako bi se proverilo da li se sve linije koda izvrsavaju pod odredjenim okolnostima. Postojanje instrukcija koje se ne izvrsavaju ni jednom ukazuje na to da postoji greska zbog koje se ne izvrsava, odnosno greska u logici i dizajnu testiranog koda. Testiranje svake grane vrsi proveru da li ce se za zadate uslove aktivirati svaka moguca grana u programu, odnosno da li je svaki blog zahvacen odgovarajucim grananjem dostizan. Dakle, kod ovog testiranja se vrsi provera uslova koji sumeravaju tok izvrsenja. Testiranje sveke putanje proverva da li se podskup instrukcija koje cine jedinstveni niz instrukcija izvrsava. Ovim testiranjem se otkrivaju greske kao sto je izostanak neke instrukcije iz niza koji se treba izvrsiti, nenamerna zamena mesta instrukcijama.

5. Objasniti ideju osnovnog strukturnog testiranja?

Ideja je da svaku instrukciju programa treba testirati barem jednom.  
U nekim prilazima se testiraju sve putanju algoritma.

6. Kako se određuje broj putanja u programu?

Broj putanja se odredjuje kao suma sledecih cinilaca:  
\* doda se 1 za pravolinijsku putanju,  
\* doda se 1 za svaku kljucnu rec if, else, while, foreach, and, i or.  
\* doda se 1 za svaku case instrukciju kod visestrukog granjanja switch, plus 1 ako nema default grane.

7. Objasniti ideju testiranja toka podataka?

Zasniva se na ideji da je upotreba podataka podlozna greskama koliko i instrukcije programa.

8. Koje vrste stanja podataka postoje?

Podaci u programu mogu da budu u jednom od sledecih stanja:  
\* definisani podaci koji imaju inicijalizovano stanje,  
\* podaci koji su se koristili u nekoj tacki programa,  
\* podaci koji su prethodno bili inicijalizovani ali su eksplicitno ili implicitno klonjeni.

9. Navesti listu alata za podršku razvojnom testiranju?

\* Radni okviri za testiranje kao sto su JUnit, NUnit, itd;  
\* Baze podataka o greskama;  
\* Generatori test podataka;  
\* Programske skele za testiranje;  
\* Alati za pracenje gresaka;  
\* Alati za poredjenje; itd.

10. Šta su to test-data generatori?

Test data generatori su alati za generisanje masovnih i slucajnih skupova podataka koji se koriste kao ulaz u testiranju, sa ciljem da se sa sto vise razlicitih vrednosti izvrse testovi. Polazna hipoteza je da sa vecom kolicinom razilicitih ulaznih podataka moze otkriti i veci broj gresaka.

**LEKCIJA 6**

1. Opisati logičke greške?

Logicke greske u programu su odstupanja od zeljenog ponasanja programa. Ove greske ne prouzrokuju direktno prekide programa, i obicno uticu na stanje i vrednosti podataka. Posledice logickih gresaka su stanja koja kasnije u toku izvrsenja mogu da proizvedu prekid programa, ili da dovedu do nekonzistentnosti u sistemu sto moze imati posledice po korisnika.

2. Opisati debugger?

Debugger je alat koji se koristi za pracenje i analizu izvrsavanja programa i stanja podataka u toku izvrsenja. Moze biti samostalan alat ili integrisan u razvojno okruzenje. Ovi alati za svoj rad, u zavisnosti od platforme, koriste informacije izvrsnog okruzenja kao sto su virtualne masine ili simbole ugradjene u programe prilikom prevodjenja. Osnovne funkcije koje su na raspolaganju su pracenje izvrsenja programa (step-by-step, step-in, step-out, ...), postavljanje prekidnih tacaka i pauziranje toka izvrsavanja na njima, pracenje stanja i vrednosti podataka, itd.

3. Objasniti naučni prilaz debugging-u?

Naucni prilaz u otkrivanju gresaka podrazumeva metodican postupak u kome se na osnovu prikupljenih informacija o ponasanju programa postavlja hipoteza na osnovu koje se eksperimentom potvrdjuje ili osporava moguci razlog za nastajanje greske. Ako se pokaze da je hipoteza tacna, greska se ispravlja kroz proces stabilizacije, a zatim se na osnovu hipoteze mogu proveriti druga potecijalna mesta koja su podlozna istoj ili slicnoj gresci, moze se kreirati zapis u bazi podataka o greskama.

4. Šta je to stabilizacija greške?

Stabilizacija greske je proces koji kroz testiranje identifikuje uslove pod kojima nastaju greske suzavanjem sireg skupa mogucih uzroka. Suzavanjem skupa se vrsi identifikovanjem faktora koji mogu uticati na nastanak greske, zatim se testiranjem kroz izmene onih faktora za koje se pretpostavlja da ne uticu na nastanak greske oni eliminisu iz test slucaja (njihovim promenama se ne eliminise greska), cime se pojednostavljuje test slucaj.

5. Šta je to "breakpoint" ?

Prekidne tacke su oznake u programima koje debugger-u govore da prilikom izvrsenja programa pauzira izvrsenje na mestu na kome je postavljena oznaka. Najznacajnija uloga prekidne tacke je da se u jednoj iteraciji ispitivanja programa brzo dodje do nekog mesta odakle se ispitivanje moze nastaviti nekom od step funkcija, kao i pregled stanja varijabli koje se prate do tacke prekida.

6. Objasniti prednosti i nedostatke debugger-a?

Nedostaci;  
\* stvara zavisnost od njegove primene umesto da se inzinjeri oslanjaju na svoje razmisljanje i deduktiju pri otkrivanju gresaka,  
\* moze da uspori proces, odnosno da vreme otkrivanaj greske bude duze nego primenom neke druge tehnike za otkrivanje gresaka, itd.  
  
Prednosti:  
\* ako se koristi kao alat u kombinaciji sa drugim tehnikama moze da olaksa i ubrza pronalazenje gresaka,  
\* moze da pomogne kod analize nepoznatog ili kompleksnog koda, itd.

7. Navesti osobine kompajlera?

U kontekstu otkrivanja gresaka, kompajleri pri prevodjenju mogu da otkriju greske kao sto su sintaksne greske, prekoracenje opsega vrednosti za tip varijable, dodelu vrednosti pogresnog tipa, itd. Pored toga mogu da prijave u upozorenja, koja su recimo proizvod staticke provere koda, kao sto su varijable koje su deklarisane a nikada nisu koriscene, poziv funkcije ciji se rezultat ne koristi, koriscenje referenci koje nikad nisu inicijalizovane, itd. Medjutim, greske koje kompajleri prijavljuju mogu da budu nejasne i da dovedu u zabludu po pitanju tipa i mesta greske, narocito ako se radi o kaskadnim greskama.

8. Navesti listu alata za debugging?

\* Interaktivni debugger-i,  
\* Alati za pracenje,  
\* Alati za poredjenje,  
\* Skele za testiranje.

**LEKCIJA 7**

1. Koje su prednosti inkrementalne inregracije?

Kod inkrementalne integracije, sa ranim fazama testiranja komponenti, greske se otkrivaju i ispravljaju ranije, cime se smanjuje vreme potrebno za implementaciju kompletnog proizvoda (sto omogucuje da se ispostuju i rokovi), dobija se pouzdaniji proizvod, itd. Druga prednost je ta sto je olaksano pracenje progresa razvoja, a korisnici imaju utisak konstantnog napredovanja jer rezultate vide tokom procesa razvoja sto povecava poverenje, itd.

2. Šta je to eksplozivna integracija?

Eksplozivna integracija je pristup u integrativnom testiranju u kome se sve, ili veci deo, komponenti kombinuje i testira odjednom.

3. Koje su prednosti i mane integracije odozgo-nadole?

Prednost je u tome sto se prvo testiraju klase najviseg prioriteta i najvece slozenosti. Kako se radi o klasama koje cine osnovu sistema, na pocetku se dovija jedan funkcionalni inkrement proizvoda. Nedostatak je veci broj interfejsa prema nizim slojevima koji mogu svojim problemima da uticu i na rad i stabilnost visih nivoa, odnosno da zahtevaju izmene na visim nivoima.

4. Koje su prednosti i mane integracije odozdo-nagore?

Prednost je rano otkrivanje gresaka i inkrementalni pristup. Nedostatak je sto se klase koje cine glavni deo sistema ostavljaju za kraj, neophondan je celokupan detaljni dizajn sistema na samom pocetku kako bi se imala sto jasnija slika onoga sto se nalazi na vrhu hierarhije.

5. Šta je to antirizična integracija?

Antirizicna integracija je pristup u kome se prvo vrsi prioritetizacija na osnovu stepena rizicnosti. Kada se odrede najrizicniji delovi, prvo se oin implmentiraju. U praksi se pokazalo da su klase na vrhu hierarhije uglavnom i najrizicnije, a zatim i sistemske klase koje se nalaze na dnu hierarhije. Ipak, i u sredisnjim nivoima mogu da se jave klase visikog rizika ako se radi o slozenim funkcionalnostima. U zavisnosti od rizika i nivoa na kojima se nalaze rizicne klase, ova metoda moze da se svede na sendvic integraciju.

6. Opisati "sendvič integraciju"?

Sendvic integracija kombinuje "odozgo-na-dole" i "odozdo-prema-gore" pristupe u integracionom testiranju. Nacin kretanja od jednog ka drugom pristupu moze da varira. Moguce je zapoceti pristupom odozgo-na-dole, a zatim ici sa odozdo-prema-gore pristupom, gde je pak moguce ostaviti neke klase za kasniju integraciju kada se vrati na odozgo-na-dole pristup.

7. Kako se obavlja integraciono testiranje?

Integraciono testiranje je nivo testiranja iznad jedinacnog testiranja, i obavlja se u toku razvoja aplikacije. Kada se zavrsi jedinacno testiranje klasa u modulima, prelazi se na (integraciono) testiranje modula koji kombinuju ove klase u koherentne funkcionalne celine, sa ciljem da se validira povezanost i rad izmedju ovih modula.

8. Opisati pristupe integracionom testiranju?

Najcesci pristupi u integracionom testiranju su "odozdo-prema-gore" i "odozgo-na-dole". U prvom pristupu (odozdo-prema-gore) testiraju se prvo moduli na nizim nivoima. Za ovaj pristup je potrebno kreirati posebne programe za simulaciju (drajver) nivoa iznad onog koji se testira. Prednost ovog pristupa je sto se greske ranije otkrivaju i ispravljaju u modulima na nizim nivoima, a nedostatak je sto se glavna funkcionalnost realizuje kasnije. Kod drugog pristupa (odozgo-na-dole) zapocinje se sa testiranjem modula na visim nivoima. Za simulaciju modula na nizim nivoima se koriste programske skele. Prednost kod ovog pristupa je sto se ranije dobija funkcionalni inkrement proizvoda. Nedostatak je u tome sto je potrebno dosta vremena za pisanje programskih skela koje oponasaju module na nizim nivoima.

9. Navesti alate za integraciono testiranje?

\* Citrus  
\* Spock  
\* Arquillian  
\* VectorCAST  
\* Steam  
\* Jasmine  
\* Protector  
\* SITA  
\* TESSY, itd.

10. Uporediti jedinično testiranje i integraciono testiranje?

Jedinicnim testiranjem se testiraju jedinice koda (klase i metode) koje obavljaju pojedinacne funkcije. Integracionim testiranjem se testiraju moduli koji objedinjuju klase u koherendne funkcionalne celine, njihova povezanost i medjusobna interakcija. Obe vrste testiranja se obavljaju za vreme implementacije softvera, pre nego sto se dobije konacni proizvod. Integraciono testiranje je nivo testiranja iznad jedinicnog testiranja i nadovezuje se na njega.

11. Navesti Java alate za integraciono testiranje?

\* JUnit  
\* Test NG  
\* Arquillian  
\* DB Unit  
\* Mickito  
\* REST Assured  
\* Cucumber  
\* Spock, itd.

12. Opisati kako se alat jUnit koristi za integraciono testiranje?

Test kod za integraciono testiranje se moze napisati uz pomoc JUnit klasa kao i regularni jedinicni testovi, ali u kombinaciji sa drugim pratecim bibliotekama. Tako se za integraciono testiranje moze koristiti Mockito biblioteka koja simulira implementaciju i ponasanje objekata, drajveri za ugradjene baze podataka i servere, drajveri koji simuliraju klijente, itd. U zavisnosti od pristupa integracionom testiranju, koriste se i kombinuju ovi i drugi alati i biblioteke koji omogucavaju pokretanje modula koji se testira, sa podmetnutim zavisnostima.

13. Uporediti funkcionalno testiranje sa jediničnim testiranjem?

Funkcionalno testiranje spada u black-box testiranje i odnosi se na testiranje funckionalnosti aplikacije kroz test slucajeve izvedenih iz zahteva. Ovo testiranje je orjentisano na ponasanje aplikacije bez znanja o, i zalazenje u detalje implementacije. Kod ovog testiranja je bitno kakve ce izlaze sistem vratiti za testirane ulazne vrednosti. Sa druge strane jedinicno testiranje je testiranje funckionalnih jedinica kao sto su klase i metode. Ovo testiranje spada u white-box testiranje, i zahteva detaljno poznavanje implementacije. Kod ovog testiranja se otkrivaju greske delova koji sacinjavaju neku funkcionalnu celinu.

14. Koje su osobine integracionog testiranja?

Integraciono testiranje vrsi proveru modula kao jedne zasebne celine i njegovog ponasanja u interakciji sa drugim modulima, nakon njihovog povezivanja, i vrsi se posle jedinicnog testiranja, u toku razvoja. Ovo testiranje mogu da vrse inzinjeri koji su radili na implementaciji ili nezavisni timovi. Integraciono testiranje omogucava da se greske rano otkrivaju, pre nego sto se dodje do konacnog proizvoda. Testiranje moze da se vrsi kroz nekoliko razlicitih strategija: top-down, bottom-up, sendvic, big-bang pristup, itd.

SE211 Pitanja I odgovori sa LAMSa

Lekcija 8

1. Opisati hijerarhijsku strukturu softvera?

N vrhu hierarhije (nivo 1) softver se posmatra kao sistem i tu se određuju granice između sistema i njegovog spoljnog okruženja, funkcionalnost, koristi za korisnika i naručioca softvera, itd. Na sledećem nivou (2) sistem se deli na podsisteme, i tu se određuju odgovornosti, zavisnosti, i komunikacija između podsistema. Na trećem nivou podsistemi se dele na klase koje inkapsuliraju podatke i logiku podsistema. Na četvrtom nivou se specificiraju atributi i metode klasa definisanih na trećem nivou. Na kraju, na nivou 5 se vrši implementacija metoda.

1. Opisati podelu sistema na podsisteme?

Sistem se deli na podsisteme identifikovanjem i određivanjem logičkih celina unutar sistema. Definisanjem podsistema se formuliše njegova uloga, zavisnost, i komunikacija sa drugim podsistemima unutar granica sistema. Prilikom određivanja tokova komunikacije bitno je minimizirati komunikaciju kako bi se smanjila zavisnost i kompleksnost sistema u celini. Takođe je bitno precizno odrediti ulogu i granice podsistema kako bi se izbeglo preplitanje odgovnornosti koje takođe vodi ka kompleksnosti i problemima u implementaciji, testiranju, i integraciji.

1. Šta je to dizajn podataka?

Dizajn podataka se odnosi na određivanje atributa (osobina) i metoda (ponašanja) klasa. Formulisanje atributa obuhvata određivanje imena atributa, nivo pristupa, tip atributa kojim se određuje skup vrednosti atributa. Formulisanje metode klase obuhvata imenovanje metode, određivanje nivoa pristupa, tip izlaza, formalne argumente koji mogu biti ulazno-izlazni i za koje se određuju osobine kao što su tip, podrazumevana vrednost, itd. Pri dizajnu podataka se treba držati dogovorenih konvencija i preporuka oko definisanja imena atributa i metoda, i slično.

1. Opisati dizajn interfejsa?

Dizajn interfejsa se odnosi na specificiranje "ugovora" komunikacije i toka podataka između sistema, podsistema, i klasa, čime se određuje minimalan skup interakcije između modula koji implementira interfejs i korisnika modula. Za dizajn interfejsa se koriste mehanizmi kao što su modifikatori pristupa, potpisi metoda sa specifikacijom ulaza i izlaza iz metoda, interfejsa i apstraknih klasa, itd. Interfejsi mogu biti eksplicitni (kao u primeru jezika koji podržavaju OOP) i implicitni koji se odnose na globalne podatke, kontrolne zastavice, i slično.

1. Koji su to artifakti o.o. analize?

\* Konceptualni model sistema,  
\* Use-case dijagrami,  
\* Sistem sekvencijalni dijagrami.

1. Objasniti vezu između artifakata o.o. analize i o.o. dizajna?

Artifekti OO analize predstavljaju ulaz za OO dizajn. Međutim postoji i povratna sprega, s obzirom da je proces iterativan i u početku se dobijaju osnovni modeli koji se u narednim iteracijama dopunjuju.

1. Opisati detraljni dizajn?

Detaljni dizajn obuhvata dizajn podataka, dizajn algoritama (potprograma) i dizajn interfejsa. Izlaz iz detaljnog dizajna je dokument koji jasno i dovoljno opisuje softverski sistem tako da se na osnovu ovih detaljnih opisa može pristupiti implementaciji sistema.

1. Opisati dokumente dizajna?

Na početku razvoja softvera kreira se dokument sa zahtevima koji sadrži funkcionalne i nefunkcionalne zahteve budućeg softverskog proizvoda. Zahtevi su predstavljeni u formi use-case dijagrama, scenarija, konceptualnih modela, itd. Nakon toga se formira dokument o arhitekturi koji daje opis visokog nivoa sistema, granica sistema, i slično. Na kraju se kreira dokument sa detaljnim dizajnom koji opisuje strukture podataka, interfejse, algoritme i druge detalje na osnovu kojih se može implementirati softverski proizvod.

1. Navesti tehnike dizajna?

\* Rafiniranje korak po korak,  
\* Modularna tehnika,  
\* Dizajnd odozno prema gore,  
\* Dizajnt odozgo prema dole,  
\* Iterativni dizajn, itd.

1. Uporediti dizajn odozgo-nadole i dizajn odozdo-nagore?

Kod tehnike dizajna odozgo prema dole se kreće od vrha definisanjem osnovnih elemenata sa malo detalja, a zatim se u sledećim iteracijama provećava nivo detaljnosti. Kod pristupa odozdo prema gore, kreće se sa što većim nivoom detalja jednog elementa sistema i ide se prema elementima sa kojima je on u interakciji. Kod prvog pristupa problem može biti ako se nema celokupna slika sistema i nisu poznati svi elementi. Kod drugog pristupa problem je što nije moguće u početku imati sve detalje. Kod oba pristupa je potrebno proći kroz više iteracija pa je pitanje izbora jednog ili drugog pristupa pitanje pogodnosti. Često se oba pristupa kombinuju kako bi se dobili bolji rezultati.

1. Navesti vrste reupotrebe softvera?

\* Dizajn šabloni,  
\* Softverske biblioteke,  
\* Radni okviri,  
\* Nasleđivanje u OOP-u,  
\* Korišćenje eksternih softverskih sistema preko servisa, itd.

1. Šta su to softverski okviri, i kako se koriste?

Radni okviri su domenski specifične biblioteke ili skupovi biblioteka koji se koriste kao osnova za izgradnju softvera. Radni okviri ređavaju česte probleme u svom domenu tako da se pri razvoju softvera inžinjer može koncentrisati na razvoj poslovne logike a ne na pomoćne funkcionalnosti. Primeri radnih okvira su JPA za mapiranje objektnod modela u Java programskom jeziku na relacioni model baze podataka i rad sa takvim objektima, Spring MVC za razvoj veb aplikacija, itd.

Lekcija 9

1. Opisati dizajn klasa?

Dizajn klase je (iterativni) proces kojim se vrši opis prepoznatih entiteta iz domena problema koji se rešava, i koji se sastoji od niza koraka u kome se vrši identifikacija objekata i pripadajućih atributa, metoda objekata koji operišu nad atributima (podacima), veze i interakciju sa drugim objektima, identifikovanje i definisanje interfejsa, određivanje vidljivosti atributa i metoda, i hierarhije nasleđivanja.

1. Šta je to vidljivost klase?

Vidljivost klase je nivo vidljivosti (mogućnost pristupa) atributa i metoda klase prema drugim klasama bilo da se radi o spoljnim vezama i interakciji, ili nasleđivanju klasa.

1. Od kojih koraka se sastoji konstruisanje klase?

\* Generalni dizajn klase koji obuhvata definisanje atributa i metoda klase, ideđivanje vidljivosti, određivanje interfejsa, zavisnosti i interakcije sa drugim klasama, odrešivanje nivoa apstrakcije i hierarhije nasleđivanja, itd;  
\* Konstruisanje metoda klase; i  
\* Testiranje klase kao celine

1. Šta je to testiranje klase?

Testiranje klase je testiranje klase kao celine, odnosno testira se validnost implementacije metoda koji su deo implementacije klase, i u kontekstu klasnog dizajna.

1. Navesti razloge za postojanje klase?

\* Modelovanje i predstavljanje realnih objekata;  
\* Redukcija kompleksnosti programa;  
\* Sakrivanje implementacije;  
\* Mogucnost modelovanja apstraktnih objekata kao baze za konkretne primerke;  
\* Enkapsulacija globalnih podataka ako postoji potreba za njima i njihovo skrivanje iza klasnog interfejsa;  
\* Izolovanje kompleksnog koda; itd

1. Navesti razloge za eliminaciju klase?

\* Klasa ima smo atribute a ne i metode (treba razmotriti da li ti atributi mogu da budu deo druge klase);  
\* Klasa ima samo metode treba razmotriti da li te metode pripadaju drugim klasama;  
\* Klasa je svemocna (radi vise poslova) sto se kosi sa principom da klasa treba da ima jednu konkretnu odgovornost (single responsibility); itd.

1. Objasniti mapiranje implementacionog klasnog dijagrama?

Klasni dijagram se direktno mapira na klase u objektno-orjentisanom programskom jeziku. Klasni dijagram sadrži i prikazuje sve elemente klase kao što su ime klase, atributi, metode, vidjlivost atrubuta i metoda, tipovi atributa, metoda, argumenata metoda, povratni tip metoda, i relacije sa drugim klasama. Na osnovu ovoga, manuelno ili uz pomoć CASE alata je moguće generisati inicijalni programski kod klase.

1. Objasniti mapiranje implementacionog sekvencijalnog dijagrama?

Sekvencijalni dijagram prikazuje tokove interakcije i razmene poruka između objekata, redosled poziva, momenat nastajanja i uništavanja objekata, opcione i alternativne tokove, itd. Na osnovu ovih detalja i dinamike interakcije prikazane na dijagramu, moguće je transformisati dijagram u deo implementacije koda klasnih metoda, jer se ovim detaljima upotpunjuju detalji o klasama na taj način što klasni dijagram daje relacije i zavisnosti, a sekvencijalni dijagram upotpunjuje dizajn detaljima o interakciji.

1. Šta je to tabela mapiranja?

Tabela mapiranja prikazuje liniju u kodu i objekate koji emituju i primaju poruke, za svaku poruku iz sekvencijalnog dijagrama.

1. Opisati kreiranje objekta na sekvencijalnom dijagramu?

U toku interakcije mogu da se koriste privremeni (tranzitni objekti) čiji je životni vek ograničen scenarijom prikazanim na sekvencijalnom dijagramu. Kreiranje i početak životnog ciklusa takvog objekta je na dijagramu naznačen tako što je simbolj za objekat postavljen niže u odnosu na glavne objekte, odnosno na mestu u sekvenci poruka na kojoj se kreira instanca. Kraj životnog ciklusa tranzitnog objekta je označen znakom X na kraju linije, a uništavanje objekta može da bude implicitno, ili explicitno (kao rezultat poruke drugog objekta).

1. Šta obuhvata detaljni o.o. dizajn?

Detaljnu specifikaciju klasa, detaljnu specifikaciju relacija i zavisnosti izmedju klasa, i detaljnu specifikaciju interakcija.

1. Šta je to detaljni dizajn atributa?

Detaljan dizajn atributa podrazumeva identifikaciju pripadajucih atributa klase i odredjivanje njihovih imena, odredjivanje vidljivosti atributa, i odredjivanja tipa atributa, odnosno skupa vrednosti koje atribut moze da ima.

1. Šta je to individualni klasni dijagram?
2. Objasniti evoluciju klasnog dijagrama uprocesu razvoja softvera?

Evolucija klasnog dijagrama u procesu razvoja softvera se odnosi na tranziciju klasnog dijagrama kroz razlicite faze dizajna, od konceptualnog modela u početnoj fazi ka dijagramima u kojima se klasni dijagram proširuje detaljima potrebnim za sagledavanje problema i definisanjem rešenja u datoj fazi.

Lekcija 10

1. Opisati PPP?

Proces programiranja pseudokodom je proces gde se vrsi dizajn potprograma korišćenjem apstraktnije alternative samom programskom jeziku. Proces je iterativan, kroz više iteracija se dizajn potprograma profinjava do finalne reprezentacije logike u potprogramu.

1. Šta je to visoki dizajn podprograma?

Visoki dizajn dizajn podrazumeva definisanje problema koji će potprogram rešavati sa dovoljno detalja da se izvrši implementacija potprograma. Ovaj postupak obuhvata određivanje ulaza i izlaza potprograma, preduslove i postuslove, itd.

1. Od kojih koraka se sastoji konstruisanje podprograma?

Ulazak u proces konstruisanja potprograma je pseudo kod. Prvi korak je pisanje deklaracije potprograma, zatim sledi pisanje prvie i poslednje instrukcije i prevodjenje pseudokoda u komentare. Nakon toga se ispod svakog komentara (linije pseudokoda) popunjava implementacioni kod. Na kraju se vrši provera programskog koda i doterivanje.

1. Opisati proveru programa kod konstruisanja podprograma?

Provera obuhvata proveru interfejsa potprogrma, odnosno ulaznih i izlaznih parametara, zatim proveriti varijable koje se koriste gde spadaju nedeklarisane varijable, neinicijalizovane varijable, itd, zatim provera logike potprograma, provera komentara, provera dokumentovanosti potprograma, usaglasenost sa aktuelnim pseudokodom, itd.

1. Opisati izbor strukture podataka?

Kako svaki program radi sa podacima potrebno je čivati te podatke na adekvatan način. Problem izbora adekvatne strukture za smeštanje podataka je problem potrošnje memorije, a može da utiče i na performanse izvršavanja. Kod izbora odgovarajuće strkuture potrebno je voditi računa o tome da ona bude dovoljna za smeštanje podataka, ali i da optimalno koristi memorijski prostor. Primera radi, ako je poznata veličina niza, može se koristiti klasičan niz za smeštanje kolekcije podataka. Sa druge strane ako niz ima tendenciju da raste, onda je bolje koristiti dinamički niz, npr. preko pointera u jeziku C/C++.

1. Objasniti izbor programskog jezika?

Programski jezici se razlikuju ne samo po sintaksi već i po lakoći učenja i korišćenja, i po optimizaciji izvršnog koda koji proizvode. Obično su jezici koji daju brži i optimalniji izvršni kod teži za korišćenje, dok nasuprot tome, lakši jezici proizvode sporiji, manje optimizovaniji i po veličini veći izvršni kod koji troši više resursa kao što su CPU vreme i memorija. U zavisnosti od tipa aplikacije može se odabrati jezik koji najviše odgovara potrebama krajnjeg proizvoda, kao i prema brzini i vremenu za implementaciju. Nekada se radi bržeg potvrđivanja koncepta može koristiti sporiji jezik za izradu prototipa, a onda kada se to pokaže kao ispravno rešenje, piše se novi kod u optimalnom jeziku. Druga opcija je kombinacija, gde se koristi sporiji jezik za pisanje aplikacije a brži samo za kritične delove aplikacije. Na kraju presudnu ulogu može da ima i brzina kompajlera, odnosno optimizacioni mehanizmi tako da bez obzira na jezik, kompajler da optimizovanu verziju aplikacije.

1. Šta je to PDL?

PDL (Programming Design Language) je jezik koji koristi prirodni jezik (engleski, srpski, itd) u kombinaciji sa sintaksom nekog programskog jezika, i koristi se pri dizajnu programa i potprograma.

1. Objasniti dizajn pomoću PDL odozgo-nadole?

Metod odozgo-nadole podrazumeva dizajn programa/potprograma korišćenjem PDL jezika, dok se stvarni izvorni kod u odabranom jeziku dobija profinjavanjem takvog pseudokoda, kroz niz iteracija.

Lekcija 11

1. Koji je kriterijum kvaliteta dizajna?

Kriterijumi koji određuju dobar dizajn su povezanost i kohezija između entieta. Prvi kriterijum određuje meru interakcije između entiteta, dok se drugi odnosi na logičku celovitost unutar entiteta.

1. Kako se definiše kohezivnost unutar nekog entiteta?

Kohezija entiteta se odnosi na logičku povezanosti unutar entiteta i daje meru koliko su elementi unutar entiteta fokusirani na ostvarenje zajedničkog cilja.

1. Navesti razloge za refaktorisanje podprograma?

\* Dupli kod  
\* Predugačak potprogram  
\* Lista parametara je suviše dugačka  
\* Potprogram ima loše ime  
\* Koriste se globalni podaci, itd

1. Navesti razloge za refaktorisanje podprograma?

\* Loša kohezija klase  
\* Loša apstrakcija interfejsa klase  
\* Klasa ima malo odgovornosti, itd

1. Navesti vrste kohezije podprograma?

\* Funkcionalna kohezija (pozeljna)  
\* Sekvencijalna kohezija  
\* Vremenska (temporal) kohezija  
\* Komunikaciona kohezija  
\* Proceduralna kohezija (nepozeljna)  
\* Koincidentna kohezija (nepozeljna)  
\* Logicka kohezija (nepozeljna)

1. Opisati podprogram sa control-flag kao ulaznim parametrom?

Ovo se odnosi na logicku koheziju gde potprogram radi vise poslova u zavisnosti od vrednosti control flag parametra. Ovakav potprogram treba refaktorisati na vise razlicitih potprograma koji obavljaju jedan zadatak.

1. Šta su to "refaktoreri"?

Refaktori su alati koji se koriste za sistematizovano restruktuiranje izvornog koda. Mogu biti zasebni ili integrisani u razvojna okruzenja.

1. Šta su to "metrics reporters"?

Metrički izveštaji su izveštaji koji daju uvid u strukturu izvornog koda, a obicno su generisani iz razvojnog okruzenja (koja podrzavaju metriku). Parametri koji su obihvaceni metrikom mogu da budu kvantitativni (broj instrukcija, linija koda, procedura, itd) i kvalitativni (pokrivenost komentarima, pokrivenost testovima, kompleksnost, itd).

Lekcija 12

1. Objasniti šta su to assertions? Ikoje vrste postoje?

Samoprovere su deo programa (potprogram, makro funkcija, ili ugrađena funkcija jezika) koje na mestu poziva proveravaju uslov, i ako taj uslov nije zadovoljen, prekidaju program, uz navedenu grešku. Dakle, samoprovera se sastoji od uslova koji mora biti ispunjen kako bi se taj deo programa smatrao ispravnim, i greške koja se emituje ukoliko uslov nije ispunjen.

1. Dati jedan primer assert funkcije?

assert idx >= 0 : "Indeks niza mora biti nula ili veci broj!";

1. Nabrojati tehnike tretiranja greški?

\* Vratiti neutralnu vrednost  
\* Vratiti najbližu ispravnu vrednost  
\* Prekinuti rad programa  
\* Prikazati poruku o grešci  
\* Vratiti sledeći ispravni podatak, itd

1. Šta su to barikade u programu?

Barikade se koriste da se greške lokalizuju, odnosno da se spreči širenje grešaka na ostatak programa.

1. Navesti tehnike za tretiranje izuzetaka?

* Obrada izuzetka konstrukcijom podržanom programskim jezikom. Nastali izuzetak se "hvata" i obrađuje rukovaocem izuzetka (exception handler).
* Validacija, sto u stvari nije obrada izuzetka vec se validacijom vrsi prevencija da uopste dodje do izuzetka koji mozda nije obradjen.

1. Uporediti izuzetke u Javi i C++?

Java podrzava try-catch i try-catch-finally konstrukcije dok C++ podrzava samo prvi od navedenih.

1. Objasniti tehniku validacije?

Validacija je tehnika provera ispravnosti kojom se sprecavaju logicke greske u programu ili emitovanje izuzetaka. Ideja je da se izvrsi provera pre kriticne upotrebe koja moze da izazove neocekivani rad programa.

1. Dati jedan primer validacione funkcije?

Provera formata email adrese, greska pri unosu vrednosti za email moze da izazove logicku gresku, ili cak i izuzetak kod slanja email poruka ako korisnik nije uneo ispravan format adrese.

Lekcija 13

1. Opisati ukratko karakteristike kvaliteta softvera?

Karakteristike kvaliteta softvera se dele na eksterne i interne. Eksterne karakteristike su bitne i vidljive korisnicima softvera. Tu spadaju efikasnost, pouzdanost, integritet, tacnost, upotrebljivost, korektnost, itd. Internet karakteristike su od interesa za inzinjere razvoja softvera. Tu spadaju portabilnost, fleksibilnost, lakoca modifikovanja, lakoca testiranja, reupotrebljivost, itd.

1. Koje su tehnike za povećanje kvaliteta softvera?

Tenike za povecanje kvaliteta su:  
\* postavljanje eksplicitnih ciljeva kvaliteta,  
\* utvrdjivanje aktivnosti za obezbedjivanje kvaliteta,  
\* formalni i neformalni pregledi,  
\* odabir tehnike testiranja,  
\* razvoj prototipa, itd.

1. Opisati šta je to konstrukciona verifikacija softvera?

Konstrukciona verifikacija softvera podrazumeva koriscenje nekoliko tenika u toku razvoja softvera kao što su pregled koda (code review), automatizovano testiranje, i upotrebu alata za automatizovano testiranje.

1. Opisati code review?

Code review je tehnika pregleda izvornog koda u svrhu otkrivanja i ispravljanja gresaka u toku razvoja. Ovom tehnikom se moze ispraviti u proseku od 65% gresaka do maksimalnih 85%. Tehnika ima nekoliko formi, od programiranja u paru do formalne i neformalne inspekcije koda.

1. Nabrojati tehnike obezbeđenja kvaliteta softvera?

\* Jedinicno testiranje,  
\* Integraciono testiranje,  
\* Debugging,  
\* Upotreba assert funkcija,  
\* Tehnicki pregledi (formalni i neformalni),  
\* Staticka analiza koda,  
\* Razvoj baziran na testiranju (test-first ili test driven development).

1. Nabrojati standarde relevantne za SQA planove?

SQA plan je formalno definisam standardom IEEEE Std 730-1989.

1. Opisati tehniku formalne inspekcije?

Formalna inspekcija je definisana i planski vodjena aktivnost pregleda izvornog koda softvera, ciji je cilj da se pronadju i isprave greske i poboljsa kvalitet samog izvornog koda. U formalnoj inspekciji ucestvuje nekoliko ljudi sa posebnim ulogama: moderator, inspektor, zapisnicar, i inzinjer koji je kreirao kod cija se inspekcija vrsi. Tipicne aktivnosti formalne inspekcije su: pripremu za inspekciju (pregled koda), sastanak ucesnika inspekcije, preradu na osnovu zakljucaka, i analizu rezultata inspekcije i modifikacije.

1. Šta je to KLOC? I za šta se koristi?

KLOC je termin koji oznacava 1000 linija izvornog koda, i to je broj u odnosu na koji se daje broj pronadjenih gresaka prilikom formalne inspekcije, sto u stvari predstavlja metriku kojom se iskazuje rezultat formalne inspekcije.

1. Nabrojati tehnike kolaborativnog konstruisanja?

\* Programiranje u paru,  
\* Neformalni pregled,  
\* Formalni pregled,  
\* Čitanje dokumentacije.

1. Opisati tehniku parnog programiranja?

Programiranje u paru je tehnika u kojoj dva programera rade zajedno na implementaciji istog dela programa. Ovakvim pristupom se smanjuje broje grešaka i povećava kvalitet izvornog koda. Tehnika je i formalno deo ekstremnog programiranja, ali ze može primenjivati i kod drugih modela.

1. Opisati statičku analizu programa?

Statička analiza programa se obavlja nad izvornim ili objektnim (među) kodom, bez izvršavanja programa. Analiza se obično vrši primenom alata koji su projektovani za ovakav tip analize. Može se vršiti analiza celog ili delova programa, na nivou jedinica ili uz uvažavanje interakcje između jedinica. Ovaj tip analize je posebno korisan kod "mission critical" softvera gde su pouzdanost i bezbednost veoma bitni, kao što je naprimer medicinski softver.

1. Uporediti statičku i dinamičku analizu softvera?

Statička analiza se vrši bez izvršavanja, dok dinamička analiza podrazumeva da se program koji se analizira izvršava. Ni jedna ni druga metoda ne mogu da otkriju baš sve greške, ali se međusobno dopunjuju. Prednost statičke analize je što se testiraju sve moguće putanje, za razliku od dinamičke analize gde se testira samo putanja u zavistnosti od konkretnih vrednost.

Lekcija 14

1. Nabrojati konstrukcione tehnologije?

\* Aplikacioni programski interfejs  
\* Parametarizacija i generalizacija  
\* Dizajn po ugovoru  
\* Defanzivno programiranje  
\* Obrada gresaka i izuzetaka  
\* Izvrsni modeli  
\* Middlware,  
\* Conkurentne primitive  
\* TDD, itd.

1. Objasniti šta je to xUML?

xUML je softverski razvojni model i razvojni jezik visoke apstrakcije. xUML prevodi model u kod manje apstraktnog programskog jezika, koji se zatim moze prevesti u izvrsni kod.

1. Objasniti code-tuning strategiju?

\* Optimizacija kompajlera  
\* Prepisivanje sporog dela programa u nekom drugom brzem jeziku  
\* Optimizacija petlji  
\* Upotreba inline funkcija umesto klasicnih poziva funkcija  
\* Neke se rekurzivne funkcije mogu zameniti petljom (stedi memoriju), itd.

1. Šta sve utiče na brzinu programa?

\* Pozivi potprograma  
\* Operacije sa celim i decimalnim brojevima  
\* Pristup nizovima  
\* Rad sa petljama i nizovima  
\* Rad sa datotekama  
\* Pristup podacima preko mreze, itd.

1. Nabrojati code-tuning tehnike?

\* Zamena if-then-else konstrukcije switch-case konstrukcijom  
\* Kombinovanje petlji  
\* Skracivanje petlji  
\* Izbacivanje uslova van petlje  
\* Permutacija udnjezdenih petlji  
\* Minimiziranje petlje  
\* Zamena operacije mnozenja sabiranjem  
\* Prepisivanje sporog dela u brzem jeziku, itd.

1. Opisati detaljno jednu code-tunin tehniku?

Prepisivanje sporog dela koda u jeziku nizeg nivoa (brzem jeziku). Recimo ako se koristi C++ onda se kriticni delovi koda mogu napisati u asembleru. Skript jezici kao sto je Python ima API kojim se omogucava pisanje funkcija u jeziku C i koje se zatim mogu koristiti iz ovog skript jezika. Ova strategija se sastoji od identifikacije kriticnog koda i njegovim prepisivanjem u drugom jeziku. Mana je sto se uvodi dodatni jezik, sto moze biti problem kod odrzavanja.

Lekcija 15

1. Šta su to zastojne tačke u debugger-u?

Program se izvršava sve dok ne dodje do markirane linije u programu tj do nekebreakpoint. U tom momentu se program zaustavlja, i otvara debugger.Da bi se zadala tačka zastoja, treba otvoriti Code editor, dakle Editor programskog koda, i onda kliknuti mišom na levoj margini, i to rezultira u pojavu jedne krupne tačke, koja ukazuje da je tu Zastojna tačka. A da bi se sklonila zastojna tačka u programu, treba kliknuti na tu tačku u levoj margini

1. Opisati koračanje kroz program pomoću debugger-a?
2. Šta je to table-driven metoda ?

Tabelarno-bazirana metoda (table-driven method), je postupak koji omogućuje da brzo/simultano pregledate tj. pretražite podatke u jednoj tabeli u cilju izbora podataka na osnovu nekih uslova.

1. Objasniti direktan pristup tabeli?

Pored pitanja pristupa tabeli, ako koristite tabelarno-baziranu metodu, je pitanje šta memorisati u tabelu. Ako je rezultat pregleda neke tabele, jednostavno neki podatak, onda se mogu stornirati podaci u tabeli. Medjutim, u ostalim slučajevima rezultat pregleda tabele je neka akcija, i u ovim slučajevima može da se u tabelu memoriše kod koji opisuje akciju. U ovim slučajevima, tabele postaju komplikovanije. Ako se primeni u odgovarajućim okolnostima , tabelarno-bazirani modeli tj. metode tj. tabelarno-bazirani programski kod je jednostavniji od komplikovane logike, jednostavniji za modifikaciju, i efikasniji.

1. Opisati princip kapsulizacije?

Kapsulizacija tj. čaurenje tj. oklopnjavanje klase tj. njenih atributa ili metoda pomoću „private“ ključne reči (publicili private?) je strožiji koncept nego abstrakcija. Abstrakcija rešava kompleksnost pružanjem modela koji ignoriše implementacione tj. kodirajuće detalje. Medjutim, kapsulizacija prevenira/sprečava od posmatranja detaljnih/internih podataka čak i ako to neko pokušava. Ova dva koncepta su povezana, naime, bez kapsulizacije dolazi do raspada koncepta abstrakcije. Znači, kapsulizacija je izuzetno važna.

1. Navesti pravila kapsulizacije?

Sledeće je pravilo za kvalitetnu kapsulizaciju: Ne iznositi pojedine članove podataka u javnost. Naime, svi podaci tj. varijable bi trebalo da su „privatne“ (privatni sektor), a sve metode su obično javne (javni sektor), i ovo je tzv. „public interface-private data“ koncept koji je glavni koncept kod kreiranja klasa. Tj. „data hiding“ odnosno smeštanje svih podataka (data members) u „private“ sektor.

Pored gornja dva primera/pravila treba primenjivati i sledeća pravila: • Minimizirati pristupnost tj. javnost klasa (generalno pravilo) • Ako neka metoda nije neophodno da bude javna staviti je da bude privatna, Npr. klasa može da ima 15 metoda od kojih je 5 javnih a ostalih 10 privatnih, dok svi atributi mogu biti privatni. • Kod baznih klasa (kod nasledja klasa), oni podaci koji mogu da budu privatni umesto zaštićenih staviti da budu „private“ a ne „protected“, dakle nije neophodno u baznoj klasi staviti da su svi atributi „protected“, već razdvojiti na sektor „private“ i „protected“, pri čemu minimizirati sektor „protected“

1. Objasniti princip abstrakcije?

Abstrakcija je sposobnost da se jedna složena operacija-forma predstavi kao pojednostavljena forma

Abstrakcija rešava kompleksnost pružanjem modela koji ignoriše implementacione tj. kodirajuće detalje.

1. Opisati kvalitetni klasni interfejs?

Klasni interfejs se po pravilu sastoji od potpisa javnih metoda klase. Prvi i možda najvažniji korak u kreiranju kvalitetne klase je kreiranje kvalitetnog interfejsa klase. Ovo se sastoji od kreiranja dobre abstrakcije za predstavljanje interfejsa, i obezbedjivanja da se interni detalji sakrivaju iza ove abstrakcije. Abstrakcija je sposobnost da se jedna složena operacija-forma predstavi kao pojednostavljena forma. Klasni interfejs omogućuje abstrakciju programskog koda koji se krije iza interfejsa. Takodje, klasni intefejs treba da ponudi jednu grupu podprograma koji jasno pripadaju zajedno toj grupi, koji imaju zajednički cilj.