# Razvoj softvera – odgovori na pitanja*

Anja Bukurov
Vojislav Stanković

17. juni 2016.

*Materijal je preuzet od dr. Saše Malkov, sa slajdova za predavanje.

# Sadržaj

# 1 Koji od pokazivača p1, p2 i p3 nije ispravno definisan u primeru?

```
int* p1,p2;
int* p3=(int*)1000;
```

Pokazivač p2 nije ispavno definisan.

# 2 U kakvom su odnosu dužine nizova s1 i s2 u primeru:

```
char s1[] = "C++";
char s2[] = { 'C', '+', '+' };
```

Strlen će da vrati jednaku dužinu, ali se s1 završava terminirajućom nulom a s2 ne.

# 3 Šta će ispisati ovaj program?

```
int a = 1;
int* b=&a;
main(){
     *b = a + 1;
     *b = a + 1;
     cout << a << endl;
}
```

Ispisaće 3.

# 4 Koji koncepti programskog jezika C++ se upotrebljavaju da bi se povećala (potencijalno ponovljena) upotrebljivost napisanog koda?

Polimorfizam, nasleđivanje, virtualno nasleđivanje.

# 5 Koji je redosled uništavanja objekata u primeru:

```
int a(3);
main() {
     int* n = new int(10);
     int k(3);
     ...
     delete n;
}
```

Prvo ce se unistiti $n$, pa $k$, i na kraju $a$.

# 6 Da li je neka (koja?) narednih linija neispravna?

```
short* p = new short;
short* p = new short[900];
short* p = new short(900);
short** p = new short[900];
```

short** p = new short[900]; će da vrati grešku pri kompajliranju jer se ne poklapaju tipovi, p je short** i pokušavamo da mu dodelimo nešto što je short*.

# 7 Koje operacije klase Lista se izvršavaju u sledećoj naredbi:

```
Lista l2 = Lista();
```

Konstruktor klase Lista poziva baznu klasu i njene konstruktore u redosledu deklarisanja (ako postoje).
Ako je klasa Lista izvedena iz apstraktne klase, ona incijilazuje pokazivač na apstraknu klasu.
Ako klasa ima ili nasleđuje virtualne funkcije, na incijilazuje pokazivač na virtualne funkcije.
Izvršava bilo koji kod u telu funkcije.

# 8 Koliko puta se poziva destruktor klase A u sledećem programu?

```
main() {
     A a, b;
     A& c = a;
     A* p = new A();
     A* q = &b;
}
```

Dva puta (za a i b), c i q su reference na objekte, a p je kopija objekta a koji se uništava kada se pozove destruktor za a.

# 9 Šta su šabloni funkcija?

Šabloni fukcija u programskom jeziku C++ predstavljaju sredstvo za pisanje polimorfnih funkcija. Polimorfizam se ostvaruje apstrahovanjem nekih tipova i/ili konstanti koji se pojavljuju u deklaraciji i implementaciji funkcije. Najčešće se apstrahuju tipovi argumenata i rezultata, ili konstante koje se odnose na veličinu ili strukturu argumenata ili rezultata, ali se mogu apstrahovati i tipovi i konstante koji se koriste u samoj implementaciji.

# 10 Kako se pišu i koriste šabloni funkcija?

Deklaracija šablona može biti samo globalna. Svaka upotreba šablonske klase predstavlja deklaraciju konkretne šablonske klase.
Poziv šablonske funkcije ili uzimanje njene adrese predstavlja deklaraciju konkrente šablonske funkcije.

Ako se definiše obična negenerička klasa ili funkcija sa istim imenom i sa potupno odgovarajućim tipovima kao što je šablon klase tj. funkcije, onda ova definicija predstavlja defniciju konkretne šablonske klase tj. funkciju za date tipove.

```
Template<classT>//sablon
voidsort(vector<T>&v){...}
voidsort(vector<int>&v){...} //konkretna funkcija
```

Funkcija članica šablonske klase je implicitno šablonska funkcija, Prijateljske funkcije šablonske klase nisu implicitno šablonske funkcije.
Funkcija članica nekog šablona klasa je implicitno i generička funkcija (šablon funkcije), pri čemu su argumenti šablona klasa ujedno i argumenti šablona funkcije.

# 11 Napisati šablon funkcije koja računa srednju vrednost dva broja za bilo koji numerički tip.

```
template< typename T >
   T avg( T x, T y ){
       return (x+y)/2;
   }
```

# 12 Šta su šabloni klasa?

Formalni argumenti šablona klase mogu biti:
Tipovi klase (class T),
Prosti objekti (biće konkretizovani konstantnim izrazima). Oznaka tipa generisane klase treba da sadrži:
Identifikator generičke klase
Listu stvarnih argumenata za generičke tipove i konstante unutar $< >$ iza identifikatora.

# 13 Kako se prevode šabloni klasa?

Konkretne klase se generišu kada se prvi put naiđe na definiciju objekta u kojoj se koristi identifikator te klase.
Pri generisanju klase se generišu i sve funkcije članice.
Mehanizam je statički - instance šablona se prave tekstualnom zamenom u vreme prevođenja.

# 14 Šta je neophodan uslov da bi se pozivi nekog metoda dinamički vezivali? Šta je dovoljan uslov?

Dinamičko vezivanje, se dešava kad metodima objekta pristupamo kroz referencu ili pokazivač na objekat (ili neki od pametnih pokazivača) pod uslovom da je metod virtualan.

Dakle: dovoljan uslov je da mu pristupamo preko reference ili pokazivača na objekat, a neophodan je da je metod virtualan.

```
object o = some_fn1();
o.method(); // nije dinamički

object *o = some_fn2();
o->method() // jeste dinamički, ako je virtualan metod

object &o = some_fn3();
o.mehtod() // jeste dinamički ako je virtualan metod
```

# 15 Šta je virtualna funkcija?

Funkcije članice osnovne klase koje se u izvedenim klasama mogu redefinisati, a ponašaju se poliformno, nazivaju se virtualne funkcije (virtual function).
Virtualna funkcija se u osnovnoj klasi deklariše pomoću ključne reči virtual:, na početku deklaracije. Prilikom deklarisanja/definisanja virtualnih funkcija u izvedenim klasama ne mora se stavljati reč virtual.

Virtualne metode su metode koje se u izvedenim klasama mogu predefinisati.
Piše se nova implementacija metode koja je specifična za tu izvedenu klasu.
Klasa koja sadrži barem jednu poliformnu metodu naziva se poliformna klasa. Virtualna metoda ne mora biti predifinisana u svakoj izvedenoj klasi. Deklaracija virtualne funkcije u izvedenoj klasi mora da se potpuno slaže sa deklaracijom iste u osnovnoj(baznoj) klasi.

```
#include <iostram>
using namespace std;

class Voce{
public:
     virtual void jedem(){
         cout<<"Ja jedem"<<endl;
        }
}
```

```
class Jabuka: public Voce{
public:
void jedem(){
     cout<<"Ja jedem jabuku"<<endl;
     }
}

class Kruska: public Voce{
public:
void jedem(){
     cout<<"Ja jedem krusku"<<endl;
     }
}

int main(){
Jabuka j;
    Kruska k;
    Voce *v1 = &j;
    Voce *v2 = &k;
    v1->jedem();
    v2->jedem();

    return 0;
}
```

U prethodnom primeru rezultat rada programa biće "Ja jedem jabuku, Ja jedem krušku." Virtualna funkcija jedem u baznoj klasi Voce kaže, $v1- > jedem()$, proveri da li u izvedenoj klasi postoji metod jedem, ako postoji izvrši ga, ako ne postoji, izvrši metod bazne klase.

# 16 Šta je čisto virtualna funkcija?

Virtualna funkcija koja nije definisana za osnovnu klasu naziva se čistom virtualnom funkcijom.
Čisto virtualna funkcija kaže kompajleru eksplicitno da u svakoj klasi koja nasleđuje baznu klasu ta funkcija mora biti napisana.

```
class Voce{
public:
     virtual void jedem() = 0;
}
```

# 17 Šta je apstraktna klasa?

Klasa koja sadrži bar jednu virtualnu funkcije naziva se apstraktnom klasom.
Apstraktna klasa ne može imati instance (objekte), već se iz nje samo mogu izvoditi druge klase. Mogu da se formiraju pokazivači i reference na apstraktnu klasu. Apstraktna klasa predstavlja samo generalizaciju izvedenih klasa.
Zajedničke osobine nekoliko klasa se grupišu u jednu osnovnu (apstraktnu ) klasu. Suprotan process aprstrakcije je specijalizacija.
Mogu postojati pokazivači i reference do apstraktne klase.

Apstraktna klasa može imati:

- Podatke članove
- Nevirtualne metode
- "Obične" virtualne metode

Apstraktna klasa može imati konstruktor iako se ne mogu konstruisati objekti apstraktne klase. Klasa koja se izvodi iz apstraktne klase je jedna konkretizacija te klase. Apstrkatna metoda nije definisana za osnovnu klasu.

# 18 Šta su umetnute (inline) funkcije? Kako se pišu?

Funkcija navedena kao umetnuta (obično) je proširena "in line" u svakom pozivu.
Uopšteno, mehanizam inline je namenjen da optimizuje male pravolinijski funkcije, koje se pozivaju često.
'Inline' je sugestija kompajleru, kompajler može da inlajnuje stvari koje nisu deklarisane kao inline, i može da ne inlajnuje stvari koje jesu.

# 19 Šta su umetnuti (inline) metodi? Kako se pišu?

Umetnuti metodi su isto što i umetnute funkcije. Metod je termin koji se koristi generalno u OOP-u, u C++ standardu se zovu 'member function'.

# 20 Koje su osnovne vrste neuspeha pri razvoju softvera? Objasniti svaku ukratko.

Osnovno merilo neuspeha je izgubljena materijalna vrednost - plaćena cena neuspeha

- uložena sredstva
- izgubljeno vreme
- posledice po čitav poslovni sistem

# 21 Koje su osnovne vrste neuspeha pri razvoju softvera? Objasniti svaku ukratko.

- Prekoračenje troškova
- Prekoračenje vremenskih rokova
  - troškovi zbog produženog razvoja
  - troškovi zbog kašnjenja puštanja u rad
- Rezultat nije upotrebljiv (u planiranim razmerama)
  - Sistem je implementiran u skladu sa zahtevima, ali ne odgovara stvarnim potrebama
  - Neispunjenost nefunkcionalnih zahteva
- Odustajanje od projekta
  - usled nekog od prethodno navedenih raloga (ili više njih)
- Katastrofalne greške (bagovi)

# 22 Šta je neupotrebljiv rezultat? Koji su aspekti neupotrebljivosti? Objasniti.

Čak i kada postoji planiranje i kada se projekat implementira po planu, rezultat može da bude neupotrebljiv ili da predstavlja neuspeh.

Aspekti "neupotrebljivosti" :

- Neupotrebljiv korisnički interfejs
  - loše rešeni ergonomski aspekti
  - nepostojanje fizičkog (hardverskog) odziva

  - neintuitivan izgled korisničkog interfejsa
  - problematične kontrole interfejsa
  - spor odziv
  - nepouzdanost
  - ...
- Procedura korišćenja nije ostvariva/dobra u realnim uslovima
  - Uz kupljenu knjigu se dobija besplatna sveska, ali to mora da se zavede kao prodaja. Ako sistem neomogućava promenu cene, to neće biti moguće. Ako sistep ne dopušta da se cena manuelno postavi na 0, takođe neće biti moguće.
  - Pretraživanje knjiga (ili sličnog kataloga) koje podrazumeva unošenje tačnog naslova.
  - Ako IS omogućava uptavi da tačno vidi i kritikuje magacionare zbog neracionalnog zauzeća prostora, a ne omogućava automatsku podršku u vidu saveta, magacioneri počinju da troše nesrazmerno mnogo vremena za planiranje prostora. Iako rezultat jeste mnogo bolja iskorišćenost prostora, gubi se na većem utrošku radnog vremena.

## 23 Koji su najčešći uzroci neupotrebljivosti rezultata razvoja softvera? Objasniti ukratko.

Mogući uzroci:

- Nerealni ili nejasni ciljevi projekta
- Neprecizna procena potrebnih resursa
- Loše definisani zahtevi
- Slaba komunikacija između klijenta, tazvijaoca i korisnika

## 24 Na kojim stranama se nalaze problemi pri razvoju softvera? Objasniti ukratko i navesti po jedan primer.

- Problemi na strani klijenta
- Problemi na strani razvijalaca
- Višestrani problemi

## 25 Koji su najčešći problemi na strani zainteresovanih lica (ulagača) pri razvoju softvera? Objasniti ukratko.

Problemi na strani klijenta:

- Nerealni ili nejasni ciljevi projekta
- Neusklađenost ciljeva i strategije
- Politika ulagača
- Komercijalni pritisak
- Otpor korisnika prema primeni novog softvera

## 26 Koji su najčešći problemi na strani razvijaoca pri razvoju softvera? Objasniti ukratko.

Problemi na strani razvijaoca:

- Slabo vođenje projekta
- Neprecizna procena potrebnih resursa
- Slabo izveštavanje o stanju projekta
- Neupravljani rizici
- Upotreba "nezrelih" tehnologija
- Nesposobnost da se iznese složenost projekta
- Tok praktičnog razvoja bez čvrstih principa i pravila

## 27 Koji su najčešći problemi koji se odnose na ove strane u razvoju softvera (ulagači i razvijaoci)? Objasniti ukratko.

Višestrani problemi:

- Slaba komunikacija između klijenata, razvijaoca i korisnika
- Nepoverenje između klijenata i razvijaoca
- Loše definisani zahtevi

## 28 Objasniti kako pristupi planiranju mogu dovesti do problema.

Loše planiranje je jedan od najčešćih uzroka neuspeha. Ima dva osnovna oblika:

- nedovoljno dobro planiranje
- preterano planiranje

Nedovoljno planiranje se obično ogleda kroz:

- nedovoljnu analizu problema
- loše i/ili neprecizno definisane zahteve

Najčešće posledice su:

- nesrazmerno veliki broj naknadnih korekcija zahteva
- slaba upotrebljivost rešenja
- probijanje rokova

Veoma čest uzrok problema je vid preteranog planiranja. Preterano planiranje se obično ogleda kroz:

- preširoko i nekoncentrisano ulaženje u projekat (suviše duboka i obimna analiza sa ranim detaljnim projektovanjem)
- preširoko ulaženje u implementaciju
- preambicioznost, nesrazmerno realnim mogućnostima

Najčešće posledice su:

- kasno uočavanje propusta
- otežana tranzicija
- probijanje rokova

# 29 Šta je upravljanje rizicima?

"Upravljanje rizicima je proces prepoznavanja, procenjivanja i kontrolisanja svega onoga što bi u projektu moglo da krene naopako pre nego što postane pretnja uspešnom dovršavanju projekta ili implementacije informacionog sistema." (Whitten, 2001)

U upotrebi su i termini "analiza rizika" i "inžinjering rizika".
Upravljanje rizicima nije vezano smao za razvoj softvera već i za svaki drugi složen razvojni/graditeljski proces. U slučaju razvoja softvera

- Najveći značaj ima dobro ušravljanje rizicima u početnim fazama
- Otežano je u slučaju dugačkih razvojih ciklusa

# 30 Koji su osnovni uzroci rizika u razvoju softvera? Navesti bar 7.

- manjak osoblja
- nerealni rokovi i budžet
- razvoj pogrešnih funkcija
- razvoj pogrešnog interfejsa
- preterivanje ("pozlaćivanje")
- neprekidni niz izmena u zahtevima
- slabosti eksterno realizovanih poslova
- slabosti u eksterno nabavljenim komponentama
- slabe performanse u realnom radu
- rad na granicama računarskih nauka

# 31 Koji su najvažniji savremeni koncepti razvoja koji su nastali iz potrebe za smanjivanjem rizika u razvojnom procesu?

- Inkrementalni razvoj
- Određivanje koraka prema rokovima
- Pojačana komunikacija među subjektima
- U žiži su objekti a ne procesi
- Pravljenje prototipova

# 32 Objasniti princip inkrementalnog razvoja.

**Princip** :

- Umesto da se razvoju prisstupa kao velikom celovitom poslu, sa složenim projektom i implementacijom, posao se deli u niz inkrementalnih faza posvećenih manjim celinama posla.

**Osnovna dobit** :

- Svaka od faza razvoja je značajno manja i jednostavnija nego čitav posao, čime se pojednostavljuju planiranje i implementacija.
- Veća je tačnost planiranja troškova i rokova i tačnije izveštavanje o toku razvoja.

**Osnovni rizik** :

- Ako se u prvim koracima potpuno zanemare predstojeći, postoji rizik da će u narednim koracima biti potrebne veće izmene.
- Ako se u prvim koracima uzmu u obzir svi predstojeći, postoji rizik da se njihova složenost približi složenosti čitavog sistema (”naduvavanje koraka”), čime ovakav pristup gubi smisao.
- Često nije moguće sagledati unapred broj, cenu i ukupno trajanje svih koraka.

# 33 Objasniti princip određnja koraka prema rokovima.

**Princip** :

- Za svaki inkrementalni korak se prvo odrede budžet i rokovi,a tek posle toga poslovi koji će korakom biti obuhvaćeni.

**Osnovna dobit** :

- Drastično smanjivanje rizika od prekoračivanje budžeta ili rokova.
- Uspostavljanje ritma redovnog isporučivanja novih verzija sistema
- Redovnija kontrola kvaliteta i viši stepen poverenja između subjekta.
- Postepeno prilagođavanje korisnika novim elemtnima sistema.

**Osnovni rizik** :

- Uspostavljanje tesnih okvira inkrementalnog koraka može otežati pojedine korake i podići ukupnu cenu i trajanje razvoja.
- Neki elementi sistema se ne mogu prirodno podeliti u različite korake.
- Svaki korak zahteva troškove isporučivanja što u zbiru može da postane celika stavka u slučaju velikog broja malih koraka.
- U prvim koracima se obično biraju poslovi koji donose veću dobit.
- Pri kraju razvoja postoji rizik da se ne implementiraju poslovi ”čija je cena veća od dobiti” iako su značajni za sistem kao celinu.

# 34 Objasniti princip pojačane komunikacije među subjektima.

**Princip** :

- U planiranju (i svakog pojedinačnog koraka) se uključuju u razmatranje sve vrste subjekata u što većem broju.

**Osnovna dobit** :

- Dobija se tačnija slika o potrebnim ciljevima iz različitih uglova.
- Smanjuje se rizik tazvoja neupotrebljivog rešenja.
- Subjekti se kroz proces razvoja pripremaju za upotrebu.

**Osnovni rizik** :

- Previše informacija može dovesti do preteranog planiranja, a time i do ”naduvavanja” pojedinačnih koraka.
- Prevelikim razmatranjem mišljenja subjekta koji su navikli na postojeće procese i ne sagledavaju planirane izmene može se smanjiti obim suštinskih funkcionalnih izmena u okviru koraka, a time i nepotrebno povećati broj koraka da se dođe do ”konačnog” rešenja.

# 35 Objasniti princip davanja prednosti objektima u odnosu na procese.

**Princip** :

- U središte pažnje se pri razvoju stavljaju objekti a ne procesi.

**Osnovna dobit** :

- Objekti su obično stabilniji od procesa (tj. izmene u načinu poslovanja se manje odražavaju na objekte nego na procese)
- Takav pristup je prilagođeniji inkremetalnom razvoju
- Smanjuje se rizik od pogrešnih odluka u ranim fazama razvoja.
  - U ranim koracima se više pažnje posvečuje objektima.
  - U kasnijim koracima se više pažnje posvećuje procesima.

**Osnovni rizik** :

- Potpuno zanemarivanje procesa u ranim fazama razvoja može voditi pogrešnoj arhitekturi sistema, što se kasnije veoma teško (skupo) menja.
- Sasvim detaljno razmatranje procesa u ranim koracima preti da dovede do "naduvavanja" prvih koraka.

# 36 Objasniti princip pravljenja prototipova.

**Princip** :

- U okviru analize problema se pravi prototip koji odražava način funkcionisanja i upotrebe softvera.

**Osnovna dobit** :

- Olakšava se ne-tehničkim subjektima da u ranim fazama razvoja uoče određene nedostatke.
- Smanjuje se rizik od pogrešnih odluka u ranim fazama razvoja.

**Osnovni rizik** :

- Prototipovi obično odražavaju funkcionalne aspekte i elemente korisničkog interfejsa, ali ne i unutrašnju strukturu softvera.
- Posledica je da su samo delemično odgovarajući objektno orijentisanim metodologijama.
- Nedovoljno široko napravlje prototip i nedovoljno široko razmatranje prototipa mogu da prikriju nedostatke u drugim aspektima IS.
- Previše pažnje posvećene prototipu preti da "naduva" fazu njegove izrade.

# 37 U kojim okolnostima su nastale objektno orijentisane razvojne metodologije?

- Postojanje metodologija koje strukturirano pristupaju analizi i opisivanju procesa
- Potreba za strukturiranim opisivanjem podataka
- Potreba za opisivanjem entiteta koji menjaju stanja
- Podizanje nivoa apstraktnosti posmatranja elemenata sistema
- Programiranje upravljano događajima
- Vizuelni korisnički interfejsi

- Povećana modularnost softvera
- Skraćivanje razvojnog ciklusa
- Tranzicija modela
- Višestruka upotrebljivost softvera
- i drugo

Za razliku od strukturnih, koje su u središte pažnje stavljale uređivanje procesa i algoritama, objektno orijentisane metodologije u redište pažnje stavljaju uređivanje objekata kojima se opisuje sistem. Razvoj OO metodologija je počeo u vreme kada su slabosti prethodnih metodologija bile uglavnom poznate. U njih su ugrađeni neki od predstavljenih savremenih koncepata razvoja.

# 38 Objasniti osnovne koncepte pristupanja objektno orijentisanih razvojnih metodologija problemu razvoja.

- U žižu stavljaju objekte, a ne procese.
- Sve OOM se odlikuju skraćivanjem trajanja razvojnih ciklusa
  - RUP (i druge) propisuju inkrementalni razvoj
  - Agilne metodologije propisuju određivanje koraka prema rokovima i troškovima
- Pojačana komunikacija među subjektima u svim fazama razvoja.

# 39 Šta je objekat? Objasniti svojim rečima i navesti jednu od poznatih definicija.

Objekat je apstrakcija nečeg konkretnog u domenu problema, koja opisuje sposobnost sistema da o tome čuva informaciju, interaguje sa time ili oboje. (Coad, Yourdon, 1990)

Objekat je koncept, apstrakcija ili nešto sa jasnim granicam i smislom u odnosu na konkretan problem. Objekat ima sve svrhe: da pomogne razumevanju stvarnog saveta i pruži praktičnu osnovnu za računarsku implementaciju. (Rumbaugh at al., 1991)

Objekti se opisuju odgovorima koje mogu da daju na pitanja:

- Ko sam ja?
- Šta mogu da uradim?
- Šta znam?
  (Wirf-Brock, 1990)

Objekti imaju stanje, ponašanje i identitet. (Booch, 1994)

Objekat ima životni ciklus:

- nastajanje
- postojanje
- nestajanje

# 40 Šta je klasa? Atribut? Metod?

Klasa je apstrakcija skupa objekata koji imaju "dovoljno sličnosti". Kao kriterijum sličnosti se prevashodno uzima u obzir ponašanje. Stanje i znanje objekta ne utiče značajno na njihovu klasifikaciju. Objekat prestavlja konkretan primerak klase.

Klasa je opis skupa objekata koji dele iste atribute, operacije, metode, odnose i sematniku. Svrha klase je da deklariše kolekciju metoda, operacija i atributa koja u potpunosti opisuje strukturu i ponašanje tih objekata. Objekta je primerak koji potiče iz klase, strukturiran je i ponaša se u skladu sa svojom klasom.

Atributi su sredstvo za opisivanje stanja i znanja
Metodi su sredstvo za opisivanje ponašanja

Klase se formalno definišu kao skupovi svih objekata koji:

- Imaju propisane atribute odgovarajućeg imena i tipa.
- Mogu da primene propisane metode odgovarajuće imena i tipa.
- Zadovoljavaju odgovarajuće formalne uslove semantika atributa i metoda.

U većini savremenih programskih jezika se raspoznavanje da li neki objekat pripada nekoj klasi odvija pragmatično, bez primene ad-hok pravila. Pripradnost objekta klasi po pravilu se određuje u trenutku pravljenja objekta i ne može se menjati tokom života objekta.
Napomena: Postoje jezici kod kojih se pripadnosti klasi uspostavlja dinamički, proveravanjem semantičkih pravila.

# 41 Koji su osnovni koncepti na kojima počivaju tehnike objektno orijentisanih metodologija?

- enkapsulacija
- interfejs
- polimorfizam
- nasleđivanje
  - specijalizacija i generalizacija
  - hijerarhije klasa

# 42 Objasniti koncept enkapsulacije.

Struktura objekta je njihova interna stvar

- ne sme se izlagati spoljašnjem svetu
- atributima se sme pristupati samo posredno

Postoje slučajevi kada neki atributi predstavljaju osnovu ponašanja objekata ili klasa

- tada im se može omogućiti javni pristup
- tada je ipak preporučljivo enkapsuliranje

**Svrha**

- Apstrahovanje strukture metodima
- Viši nivo međusobne nezavisnosti klase od modula ukojima se upotrebljava

# 43 Objasniti koncept interfejsa.

Objekat (klasa) pruža spoljašnjim korisnicima samo skup metoda putem kojih mogu komunicirati sa njim.

Takav skup metoda se naiva javni interfejs objekta (klase)

Interfejs se oblikuje tako da omogući obavljanje jednog celovitog posla

**Svrha**

- Suština objekta je u njihovom ponašanju. Interfejs omogućava to ponašanje.
- Sužavanjem interfejsa na suštinu funkcionisanja objekta prikriva se sva složenost implementacije i pruža viši nivo nezavisnosti objekta od okruženja.

Napomena: Ako objekat ima više interfejsa, znači da ima više funkcija, pa je potrebno razmotriti njegovo razlaganje na više objekata.

# 44 Objasniti koncept polimorfizma.

Polimorfizam podrazumeva da se jednom napisan kod može upotrebljavati za različite vrste objekata.

Postoje tri osnovne vrste polimorfizma: hijerarhijski, parametarski i implicitni.

- Svi OO programski jezici omogućavaju mhijerarhijski polimorfizam.
- Neki savpremeni OOPJ omogućavaju parametarski. polimorfizam
- Samo retki jezici podržavaju implicitni polimorfizam.

**Svrha:** Polimorfizam omogućava pisanje apstraktnijeg koda, koji ima visoku upotrebljivost.

# 45 Objasniti koncept nasleđivanja i odgovarajuće odnose.

Nasleđivanje klase je ekvivalentno uvođenju jednosmerne parcijalno uređene relacije "jeste" između klasa.

Klasa A "jeste" klasa B akko svaki objekat klase A ima sve osobine koje imaju i objekti klase B.

Ako A "jeste" klasa B kaže se i da je:

- A "izvedena" klasa iz B ili A "je potomak" klase B
- B "osnovna" klasa za A ili B "je predak" klase A

Relacija "jeste" je parcijalna relacija poretka (refleksivna i tranzitivna).

Predstavlja osnovu za građenje hijerarhija klasa.

**Svrha** :

- Nasleđivanje se koristi za eksplicitno označavanje sličnosti među klasama (objektima).
- Predstavlja osnovu za hijerarhijski polimorfizam: ako se to može uraditi sa svakim objektnom klase B, onda se to može uraditi i sa svakim objektom klase A koja je izvedena iz B.

Nasleđivanje se posmatra u dva smera, kao specijalizacija ili generalizacija:

- klasa A je poseban (specijalan) slučaj klase B
- klasa B je opštiji (generalan) slučaj klase A

# 46 Kroz koje faze je prošao razvoj OO metodologija?

- Početni koraci (-1997)
- Oblikovanje UML-a (1995-2005)
- Post-UML koraci (2000-)

## 47 Koje su karakteristike prve faze razvoja OO metodologija?

- Veliki broj različitih notacija i metodologija
- Nijedna kompletna i dovoljno široka
- Booch, 1991.
- Coad, Yourdon, 1991.
- Martin, Odell, 1992.

## 48 Koje su karakteristike druge faze razvoja OO metodologija?

- Nekoliko dubljih metodologija koncetrisanih na različite faze razvoja
- Pokušaji ujednačavanja notacije
- Akcenat na notaciji

## 49 Koje su karakteristike treće faze razvoja OO metodologija?

- Ujednačena notacija
- Široko shvatanje procesa razvoja
- Potpuno posvećenje metodologijama
- RUP i druge savremene metodologije
- Agilne metodologije

## 50 Šta je UML?

Objedinjeni jezik za modeliranje (Unified Modeling Language). Sam jezik ne predstavlja metodologiju, ali je oblikovan prema paralelno razvijanoj metodologiji - "Objedinjeni pristup" (Unified Approach). Sam jezik je razvijen, publikovan i šire prihvaćen pre odgovarajuće metodologije.

## 51 Koje vrste dijagrama postoje u UML-u? Objasniti.

Dijagrami se dele u tri grupe:

- dijagrami ponašanja
- dijagrami interakcije
- strukturni dijagrami

Slika 1: Vrste dijagrama

## 52 Navesti strukturne dijagrame UML-a.

- Dijagram klasa (class diagram)
- Dijagram komponenti (component diagram)
- Dijagram objekta (object diagram)
- Dijagram profila (profile diagram)

- Dijagram složene strukture (composite structure diagram)
- Dijagram isporučivanja (deployment diagram)
- Dijagram paketa (package diagram)

## 53 Koja je uloga i šta su osnovni elementi dijagrama klasa?

Ilustruje elemente statičkog modela, kao što su klase, njihov sadržaj i međusobne odnose.
Sadrži:

- nazive klasa
- atribute klasa
- modele klasa
- specijalizaciju i generalizaciju
- odnose
- asocijaciju
- agregaciju
- kompoziciju

## 54 Koja je uloga i šta su osnovni elementi dijagrama komponenti?

Ilustruje komponente koje čine aplikaciju, sistem ili organizaciju.
Sadrži:

- nazive komponenti
- njihove međusobne odnose
- javne interfejse

## 55 Koja je uloga i šta su osnovni elementi dijagrama objekata?

Predstavlja objekte i njihove odnose u jednom trenutku vremena. Koristi se kao dopuna dijagrama klasa i komunikacije za opisivanje dinamičkih sistema.
Sadrži:

- nazive klasa
- imena i vrednosti atributa
- odnose

Dijagram složene strukture predstavlja internu strukturu klase, objekta, komponente ili slučaja upotrebe.
Sadrži:

- složene komponente i njihove elemente
- tačke interakcije sa drugim elementima sistema

## 56 Koja je uloga i šta su osnovni elementi dijagrama isporučivanja?

Predstavlja elemente fizičke arhitekture sistema.
Sadrži:

- čvorove (servere)
- softverske ili hardverske podsisteme
- međusobne veze podsistema
- može da ilustruje i zastupljenost komponenti u podsistemima

## 57 Koja je uloga i šta su osnovni elementi dijagrama paketa?

Ilustruje kako su elementi logičkog modela organizovani u pakete, kao i međuzavisnosti paketa.
Sadrži:

- nazive i granice paketa
- klase u paketima
- međusobne odnose klasa
- međusobne zavisnosti paketa
- može se koristiti i u domenu slučajeva upotrebe

## 58 Navesti dijagrame ponašanja UML-a.

- Dijagram aktivnosti (activity diagram)
- Dijagram stanja (state machine diagram)
- Dijagram slučajeva upotrebe (use case diagram) - u ove dijagrame se mogu ubrojati i dijagrami interakcije

## 59 Koja je uloga i šta su osnovni elementi dijagrama aktivnosti?

Predstavlja poslovne procese višeg nivoa, tokove podataka i eventualno složene logičke elemente sistema.
Sadrži:

- procese
- tokove podataka
- čvorišra i grananja
- uslovne tačke
- početne i završne tačke
- može da sadrži i ”linije autora”

## 60 Koja je uloga i šta su osnovni elementi dijagrama stanja?

Opisuje kako se stanja objekta menjaju u zavistnosti od inerakcija u koje objekat ulazi.
Sadrži:

- sva moguća stanja objekta
- posebno označeno početno i završno stanje
- prelaske između stanja
- strelica od prethodnog prema narednom stanju
- nazivi događaja koji menjaju stanje objekata
- odgovarajuća objašnjenja

## 61 Koja je uloga i šta su osnovni elementi dijagrama slučajeva upotrebe?

Predstavlja slučajeve upotrebe, aktere i njihove međusobne odnose.
Sadrži:

- slučajeve upotrebe
- aktere
- pakete
- podsisteme
- međusobne odnose

## 62 Navesti dijagrame interakcija.

- Dijagram komunikacije (communication diagram) - raniji naziv Dijagram saradnje (collaboration diagram)
- Dijagram iterakcija (iteraction diagram ili interaction overview diagram)
- Dijagram sekvence (sequence diagram)
- Dijagram vremena (timing diagram)

## 63 Koja je uloga i šta su osnovni elementi dijagrama komunikacije?

Predstavlja objekte, njihove međusobne odnose i poruke koje razmenjuju. Pažnja se po pravilu posvećuje strukturnoj organizaciji objekta koji učestvuju u razmeni poruka.
Sadrži:

- objekte (može i aktere)
- poruke
- komentare i napomene

# 64 Koja je uloga i šta su osnovni elementi dijagrama interakcije?

Varijanta dijagrama aktivnosti u kojoj je akcenat na upravljanju procesima ili sistemom. Svaki čvor/aktivnost u dijagramu može da predstavlja neki drugi dijagram interakcija ili aktivnosti.
Sadrži:

- objekte
- manje dijagrame aktivnosti ili interakcija
- slučajeve upotrebe
- tok odvijanja procesa (protoka podataka)
- grananja i spajanja
- početak i kraj

# 65 Koja je uloga i šta su osnovni elementi dijagrama sekvenci?

Predstavja slučajeve upotrebe, aktere i njihove međusobne odnose.
Sadrži:

- slučajeve upotrebe
- aktere
- pakete
- podsisteme
- međusobne odnose

Dijagram vremena predstavlja promene stanja ili uslova objekata tokom vremena. Uobičajeno se upotrebljava za predstavljanje promena stanja u zavisnosti od spoljnih događaja.
Sadrži:

- protok vremena
- spoljašnje događaje
- promene stanja

# 66 Šta je uzorak za projektovanje? Čemu služi?

Švaki uzorak opisuje koji se stalno ponavlja našem okruženju i zatim opisuje suštinu rešenja problema tako da se to rešenje može upotrebiti milion puta a da se dva puta ne ponovi na isti način."(Christopher Alexander)

Christopher Alexander je govorio o uzorcima za zidanje gradova ali to što je rekao važi i za uzorke objektno orijentisanog projektovanja.
Softverska rešenja su izražena pomoću objekta i interfejsa umesto zidova i vrata ali suština obe vrste uzoraka je rešavanje problema u svom kontekstu.

# 67 Koji su osnovni elementi uzoraka za projektovanje? Objasniti ih.

Svaki uzorak ima četiri bitna elementa:

- Ime uzorka
- Problem
- Rešenje
- Posledice

# 68 Objasniti ime, kao element uzoraka za projektovanje.

U nekoliko reči opisuje, njegova rešenja i njegove posledice.

Davanjem imena uzorku uvećava se rečnik projektovanja.

Tako se projektovanje podiže na viši nivo apstrakcije.

Rečnik uzoraka omogućava razmenu mišljenja o uzorcima, diskutovanje, pisanje i čitanje.

Lakše je razmišljati o projektovanju i prenositi drugima taj model i njegove ocene.

Pronalaženje dobrih imena je jedan od najtežih poslova pri izradi kataloga.

# 69 Objasniti predmet, kao element uzoraka za projektovanje.

Opisuje slučaj u kome se uzorak koristi.

Opisuju se i problem i njegov kontekst.

Moguć je opis specifičnih problema (primera).

Problem se može opisivati strukture klasa ili objekata čije osobine nagoveštavaju kruto projektovanje.

Ponekad problem sadrži spisak uslova potrebnih da bi se uzorak primenio.

# 70 Objasniti rešenje, kao element uzoraka za projektovanje.

Opisuje elemente koji čine dizajn, njihove odnose, odgovornosti i saradnju.

Ne opisuje određen konkretan projekat ili implementaciju, pošto je uzorak kao šablon koji se može primeniti u mnogim različitim situacijama.

Daje apstraktan opis problema projektovanja i uputstvo kako se on rešava opštim uređenjem elemenata (klasa i objekata).

# 71 Objasniti posledice, kao element uzoraka za projektovanje.

Obuhvataju rezultate i ocene primene uzorka.

Često se ne pominju u opisima odluka o projektovanju, ali su veoma bitne za procenu alternetiva i za razumevanje prednosti i nedostataka primene uzorka.

# 72 Navesti šta sve obuhvata opis jednog uzorka za projektovanje.

**Ime uzorka i klasifikacija** :

- Ime uzorka sažeto izlaže suštinu uzorka.
- Dobro ime je od suštinskog značaja pošto ono ulazi u rečnik projektovanja.
- Klasifikacija uzorka izvedena je po šemi koja će biti kasnije navedena.

**Namena** :

- Kratak iskaz koji odgovara na sledeća pitanja:
  - Šta je uzrok a projektovanje radi?
  - Kakvo mu je obrazloženje i namena?
  - Na koje konkretno pitanje ili problem projektovanja se uzorak odnosi?

**Poznat takođe kao** :

- Ostala poznata imena uzorka, ako postoje.

**Motivacija** :

- Scenario koji ilustruje problem projektovanja i način na koji strukture klasa i objekata u uzorku rešavaju taj problem.
- Scenario pomaže da se shvati apstraktniji opis uzorka.

**Primenjivost** :

- Na koje situacije se uzorak može prmeniti?
- Koji su primeri lošeg projektovanja koje uzorak može da ispravi?
- Kako prepoznati te situacije?

**Struktura** :

- Grafički prikaz klasa u uzorku u notaciji UML
  - dijagrami klasa
  - dijagrami interakcije

**Učesnici** :

- Klase i/ili objekti koji učestvuju u uzorku za projektovanje kao i njihove odgovornosti.

**Saradnja** :

- Kako učesnici sarađuju da bi izvršavali svoje odgovornosti.

**Posledice** :

- Kako uzorak zadovoljava svoju namenu?
- Koji su nedostaci i koristi od korišćenja uzoraka?
- Koji aspekt strukture sistema može nezavisno da se menja?

**Implementacija** :

- Kojih zamki, saveta ili tehnika bi trebalo da budete svesni prilikom implementiranja uzorka?
- Ima li nekoh pitanja zavisnih od programskog jezika?

**Primer koda** :

- Fragmenti koda koji ilustruju kako bi se uzorak mogao implementirati.

**Poznata korišćenja** :

- Primeri uzorka koji se nalaze u stvarnim sistemima.

**Povezani uzorci** :

- Koji uzorci za projektovanje su u tesnoj vezi sa ovim uzorkom?
- Koje su značajne razlike?
- Uz koje druge uzorke bi trebalo koristiti ovaj uzorak?

# 73 Šta su klasifikovani uzorci za projektovanje? Navesti po jedan primer od svake vrste uzorka.

Klasifikacija uzoraka je važna. Daje nam standardna imena i definicije za tehnike koje koristimo. Ako ne naučimo uzorke u softveru, nećemo biti u stanju da ih poboljšamo, i biće teže smisliti nove.

- Gradivni
- Strukturni
- Uzorci ponašanja

# 74 Objasniti namenu gradivnih uzoraka za projektovanje.

- Apstrahuju proces pravljenja objekata. Pomoću njih sistem postaje nezavistan od načina pravljenja objekata
- Različiti domeni
  - Uzorak za pravljenje sa domenom klase - koristi nasleđivanje za menjanje klase koja se instancira.
  - Uzorak za pravljenje sa domenom objekta - delegira pravljenje nekom drugom objektu.
- Uzorci za pravljenje su važni kada sistemi više zavise od sastavljanja objekata nego od nasleđivanja
  - naglasak se sa fiksnog kodiranja fiksnog skupa ponašanja prebacuje na definisanje manjeg skupa osnovnih ponašanja koja se mogu sastavljati u veći broj složenijih
  - pravljenje objekata sa nekim određenim ponašanjem zahteva više od pukog instanciranja klase
- U ovim uzorcima postoje dve teme koje se stalno ponavljaju:
  - svi ovi uzorci enkapsuliraju znanje o tome koje konkretne klase sistem koristi
  - kriju kako se prave primerci ovih klase i kako se sastavljaju
- U celom sistemu se o objektima zna jedino za njihove onterfejse, na osnovu toga kako su definisani u apstraktnim klasama.
- Uzorci za pravljenje omogućavaju veliku fleksibilnost u smislu
  - šta se pravi,
  - ko to pravi,
  - kako se pravi i
  - kada.
- Omogućavaju da se konfiguriše sistem objektima "proizvodima" raznovrsne strukture i funkcija.
- Konfigurisanje može da bude statično (tj. određeno u vreme prevođenja) ili dinamičko (u vreme izvršavanja).

# 75 Navesti bar četiri gradivna uzorka za projektovanje.

Najvažniji gradivni uzorci:

- Apstraktna fabrika (Abstract Factory)
- Graditelj(Builder)
- Proizvodni metod (Factory Method)
- Prototip (Prototype)
- Unikat (Singleton)

# 76 Objasniti namenu strukturnih uzoraka za projektovanje.

Bave se načinom na koji se klase i objekti sastavljaju u veće strukture. Dele se na

- Strukturne uzorke klasa i
- Strukturne uzorke sa domenom objekata

Strukturni uzorci klasa koriste nasleđivanje za sastavljanje interfejsa ili implementacija

- Jednostavan primer je višestruko nasleđivanje kojim se dva ili više klasa kombinuju u jednu
  - Rezultat je klasa u kojoj su kombinovana svojstva roditeljskih klasa.
  - Ovaj uzorak je posebno koristan za kombinovano korišćenje nezavisno razvijenih biblioteka klasa.
- Drugi primer je klasni oblik uzorka Adapter
  - Adapter prilagođava jedan interfejs drugom interfejsu tako što pravi ujednačenu apstrakciju različitih interfejsa.
  - Klasni adapter to postiže privatnim nasleđivanjem klase koju prilagođava.
  - Adapter zatim izražava svoj interfejs klase koju prilagođava.

Strukturni uzorci sa domenom objekata opisuju načine za postizanje nove funkcionalnosti kombinovanjem objekata umesto sastavljanjem interfejsa ili implementacija. Dodatna fleksibilnost sastavljanja objekata potiče od mogućnosti da se sastav menja u vreme izvršavanja što je nemoguće kod statičkog sastavljanja klasa.

# 77 Navesti bar pet strukturnih uzorka za projektovanje.

Najvažniji strukturni uzorci su:

- Adapter (Adapter)
- Most (Bridge)
- Sastav (Composite)
- Dekorater (Decorator)
- Fasada (Facade)
- Muva (Flyweight)
- Proksi (Proxy)

# 78 Objasniti namenu uzoraka ponašanja.

Bave se algoritmima i raspodelom odgovornosti među objektima.

- Ne opisuju samo uzorke objekata ili klasa već i uzorke njihove međusobne komunikacije.
- Opisuju prirodu složenog toka kontrole koji se teško prati u vreme izvršavanja.
- Pažnja prelazi sa samog toka kontrole na način međusobnog povezivanja objekata.

Dele se na:

**Klasne uzorke ponašanja** - koriste nasleđivanje za distribuiranje ponašanja. Primeri su:

- Šablonski metod
- Interpretator

**Objektne uzorke ponašanja** :

- Koriste sastavljanje objekata a ne nasleđivanje.
- Neki od njih opisuju kako grupa ravnopravnih objekata sarađuje na zadataku koji nijedan od njih ne može sam da obavi.
  - Ovde je važno pitanje kako ravnopravni objekti saznaju jedan za drugoga.
  - Ravnopravni objekti bi mogli da čuvaju međusobne eksplicitne reference ali bi to pojačalo njihovovezivanje. U krajnjem slučaju bi svaki bi svaki objekat znao za sve ostale.
- Primeri: Posrednik

# 79 Navesti bar sedam uzoraka ponašanja.

Najvažniji uzorci ponašanja su:

- Lanac odgovotnosti (Chain of Responsibility)
- Komanda (Command)
- Interpretator (Interpreter)
- Iterator (Iterator)
- Posrednik (Mediator)
- Podsetnik (Memento)
- Posmatrač (Observer)
- Stanje (State)
- Strategija (Strategy)
- Šablonski metod (Template Method)
- Posetilac (Visitor)

# 80 Objasniti kada se i kako primenjuje uzorak Proizvodni metod (Factory Method).

Definisati interfejs za kreiranje objekta, ali dati potklasi da odluči koju klasu će da instancira. Proizvodni metod omogućava da klasa prepusti instanciranje potklasi.

Koristite Proizvodni metod kada:

- klasa ne može da predvidi klasu objekta koji mora stvoriti
- klasa želi da njena potklasa precizira objekte koje stvara
- klasa prenosi odgovornost na jednu od nekoliko pomoćnih potklasa, i kada želite da se lokalizuju informacije kojih je pomoćna potklasa delegat.

# 81 Skicirati dijagram klasa uzoraka za projektovanje Proizvodni metod (Factory Method).

Slika 2: Dijagram klasa uzoraka za projektovanje Proizvodni metod

# 82 Objasniti kada se i kako primenjuje uzorak Strategija.

Definitiši familiju algoritama, enkapsuliraj svaki, i napravi ih
Koristite obrazac Strategija kada:

- se mnoge srodne klase razlikuju samo u njihovom ponašanju. Strategija obezbeđuje način da podesite klasu sa jednim od mnogih ponašanja.
- vam trebaju različite varijante nekog algoritma. Strategija može da se koristi kada se ove varijante sprovode kao hijerarhije klasa algoritama.
- algoritam koristi podatke o kojima klijenti ne treba da znaju. Upotrebiti obrazac Strategija da se izbegne izlaganje složenih struktura podataka specifičnih za algoritam.
- klasa definiše mnoga ponašanja, koja se pojavljuju kao višestruke uslovne naredbe u njenim operacijama. Umesto mnogih uslovnosti, pomeriti srodne uslovne grane u zasebnu klasu Strategije.

# 83 Skicirati dijagram klasa uzoraka za projektovanje Strategija.

Slika 3: Dijagram klasa uzoraka za projektovanje Strategija

# 84 Objasniti kada se i kako primenjuje uzorak Dekorater.

Pridaje dodatne odgovornosti objektu dinamički. Dekorater obezbeđuje fleksibilnu alternativu za proširenje funkcionalnosti potklasa.

Koristite obrazac dekorater da:

- Dodate odgovornosti u pojedinačnim objektima i transparentno, a da ne utičete na druge objekte.
- Za obaveze koje se mogu povući.
- Kada je proširivanje potklase nepraktično. Ponekad je moguć veliki broj nezavisnih ekstenzija, ali će proizvesti eksploziju potklasa da bi podržao svaku kombinaciju. Ili je definicija klase sakrivena ili na drugi način nedostupna za izvođenje potklase.

# 85 Skicirati dijagram klasa uzoraka za projektovanje Dekorater.

Slika 4: Dijagram klasa uzoraka za projektovanje Dekorater

# 86 Objasniti kada se i kako primenjuje uzorak Složeni objekat (Sastav, Composite).

Sastaviti objekte u drvolike strukture da predstavite deo/čitavu hijerarhiju. Obrazac Composite omogućava korisnicima da tretiraju pojedine objekte i kompozicije objekata ravnopravno.

Koristite Composite obrazac kada:

- želite da predstavite deo/čitavu hijerarhije objekata.
- želite da korisnici mogu da ignorišu razliku između kompozicija objekata i individualnih objekata. Klijenti će tretirati sve objekte kompozitnih struktura ravnopravno.

# 87 Skicirati dijagram klasa uzoraka za projektovanje Složeni objekat (Sastav, Composite).

Slika 5: Dijagram klasa uzoraka za projektovanje Složeni objekat

# 88 Objasniti kada se i kako primenjuje uzorak Unikat (Singleton).

Osigurati da klasa ima samo jednu instancu, i obezbediti globalnu tačku pristupa za nju.

Koristite obrazac Singleton kada :

- mora biti tačno jedna instanca klase, a ona mora biti dostupna korisnicima iz poznate pristupne tačke.
- kada jedina instanca treba da bude proširiva izvođenjem potklase, i korisnici treba da budu u mogućnosti da koriste izvedenu instancu bez menjanja svog koda.

# 89 Skicirati dijagram klasa uzoraka za projektovanje Unikat (Singleton).

| -static uniqueInstance<br>-singletonData |
|---|
| +static instance()<br>+SingletonOperation() |

# 90 Objasniti kada se i kako primenjuje uzorak Posetilac (Visitor).

Predstaviti operaciju koja će da se izvrši na elementima objekta. Posetilac omogućava da definišete novu operaciju bez promene klase elemenata na kojem je izvršavate.

Koristite obrazac Posetilac kada:

- struktura objekata sadrži mnoge klase objekata sa različitim interfejsima, i želite da izvršite operacije nad ovim objektima koji zavise od njihovih konkretnih klasa.
- mnoge različite i nepovezane operacije treba da se izvode na objektu iz strukture objekata, a želite da izbegnete "zagađivanje" njihovih klasa sa ovim operacijama. Posetilac omogućava da zadržite povezane operacije zajedno, definisanjem u jednoj klasi. Kada je struktura objekata deljena od strane dosta aplikacija, koristite Posetioce da stave operacije samo u one aplikacije kojima trebaju.
- se klase koje određuju strukturu objekata retko menjaju, ali mi često želimo definisati nove operacije nad strukturom. Promena klasa strukture objekata zahteva redefinisanje interfejsa svim posetiocima, što je potencijalno skupo. Ako se klase strukture objekata menjaju često, onda je verovatno bolje definisati operacije u tim klasama.

# 91 Skicirati dijagram klasa uzoraka za projektovanje Posetilac (Visitor).

Slika 6: Dijagram klasa uzoraka za projektovanje Posetilac

Slika 7: Dijagram klasa uzoraka za projektovanje Posetilac

# 92 Objasniti kada se i kako primenjuje uzorak Posmatrač (Observer).

Definisati zavisnost jedan-na-mnoge objekata tako da kada neko promeni stanje objekata država, svi koji zavise od njega su obavešteni i automatski se ažuriraju.

Koristite obrazac Posmatrač u bilo kojoj od sledećih situacija:

- Kada apstrakcija ima dva aspekta, jedan zavisnim u odnosu na drugog. Enkapsuliranje ovih aspekata u odvojenim objektima omogućava da ih menjamo koristimo samostalno.
- Kada promena u jednom objektu zahteva promenu u drugima, a vi ne znate koliko objekata mora da se menja.
- Kada bi objekat trebalo da bude u stanju da obavesti druge objekte bez pretpostavke o tome ko su ovi objekti. Drugim rečima, ne želimo da ovi objekti budu čvrsto vezani.

# 93 Skicirati dijagram klasa uzoraka za projektovanje Posmatrač (Observer).

Slika 8: Dijagram klasa uzoraka za projektovanje Posmatrač

# 94 Objasniti kada se i kako primenjuje uzorak Apstraktna fabrika (Abstract Factory).

Obezbediti interfejs za kreiranje porodice srodnih ili zavisnih objekata bez navođenja njihove konkretne klase.

Koristite obrazac Apstraktna Fabrika kada:

- sistem treba da bude nezavisan od toga kako su njegovi proizvodi kreirani, od sastava, i načina predstavljanja.
- sistem treba da bude konfigurisan sa više od jedne porodica klasa.
- familija srodnih klasa je dizajnirana da se koriste zajedno, i treba da sprovede ovo ograničenje.
- želite da obezbedite biblioteku klase proizvoda, a želite da otkrijete samo njihove interfejse, a ne njihove implementacije.

# 95 Skicirati dijagram klasa uzoraka za projektovanje Apstraktna fabrika (Abstract Factory).

Slika 9: Dijagram klasa uzoraka za projektovanje Apstraktna fabrika

# 96 Šta je Agilni razvoj softvera?

Agilni razvoj softveraa (ARS) je familija metodologija razvoja softvera koja je nastala krajem poslednje decenije 20. veka (kraj 80tih, početak 90tih). Naziv je oblikova 2001. kada je formulisan *Manifest* agilnog razvoja softvera. Upotrebljava se termin agilne metodologije.

Agilni razvoj softvera ima mnogo sličnosti sa OO metodologijama, ali sa drugačijim pristupom planiranju. Uglavnom se pretpostavlja upotreba tehnika OO projektovanja i programiranja. Propisuje skup principa i skup tehnika za njihovo ostvarivanje. Promoviše dinamičan i diciplinovan timski rad. Brzo reaguje na svaku promenu u okruženju.

Agilni razvoj NIJE

- odsustvo sistematičnost ponašanje članova tima potpuno odsustvo dokumentacije
- anarhično ili svojevoljno ponašanje članova tima
- potpuno odsustvo dokumentacije (softver bez dokumentacije ne vredi ništa)
- selektivna primena samo nekih od principa ARS-a
- najlakši metod razvoja softvera
- univerzalno rešenje za sve probleme
- pravi način rada za neiskusne ili nedovoljno stručne programere ili timove

# 97 Navesti osnovne pretpostavke Manifesta agilnog razvoja.

Manifest prepoznaje osnovne 4 prepostavke na kojima počiva agilni razvoj. Određuje 12 osnovnih principa agilnih metodologija. Konkretne agilne metodologije mogu imati dodatne pretpostavke, mogu imati dodatne principe, određuju metode i tehnike kojima se principi ostvaruju.

Četiri osnovne pretpostavke:

- Pojedini i saradnja ispred procesa i alata
- Funkcionalan softver ispred iscrpne dokumentacije
- Saradnja sa klijentom pre nego pregovaranje
- Reafovanje na promene pre nego praćenje plana

# 98 Objasniti pretpostavku agilnog razvoja da su pojedinci i saradnja ispred procesa i alata.

- ljudi su najvažniji deo uspešnog razvoja
- dobar proces ne može uspeti sa lošim ljudima
- loš proces i najbolje ljude čini neproduktivnim
- grupe dobrih ljudi će biti neuspešne ako nema aradnje

- dobri alati pomažu razvoj
- previše pažnje posvećene alatima je jednako loše kao potpuno odsustvo alata
- izgradnja tima je bažnija od izgradnje okruženja

## 99 Objastniti pretpostavku agilnog razvoja da je funkcionalan softver ispred iscrpne dokumentacije.

- softver bez dokumentacije je propast
  - kod nije idealno sredstvo za komunikaciju
  - neophodna je razumljiva dokumentacija
- dokumentacija mora biti ažurna
- previše dokumentacije može biti gore nego premalo
- previše vremena ide na održavanje i sinhronizaciju
- uvek je dobro pisati propratnu dokumentaciju
- ali ona mora da bude kratka i na visokom konceptualnom nivou
- uvek imati na umu da cilj nije dokumentacija nego softver
- sastavni deo, ne primarni cilj
- "ne praviti dokumentaciju osim ako je odmah i značajno potrebna" (R. Martin, tzv. "prvi zakon o dokumentaciji")

## 100 Objasniti pretpostavku agilnog razvoja da je saradnja sa klijentom ispred pregovaranja.

- softver ne može da se naručuje kao nameštaj
  - u iole složenijem slučaju praktično je neizvedivo da se napiše opis zadataka i prepusti nekome da napravi softver
- uvek je veliki problem izražavanje želje klijenta na jasan i dovoljno eksplicitan način
  - klijent ne zna šta želi niti zna šta mi možemo da uradimo za njega
- uspeh projekta zahteva redovnu komunikaciju razvojnog tima i klijenta
- precizno ugovaranje zahteva, rokova, faza i troškova pre početka projekta je suštinski nemoguće u većini slučajeva
- redovna isporuka softvera doprinosi komunikaciji i saradnji sa klijentom

## 101 Objasniti pretpostavku agilnog razvoja da je reagovanje na promene ispred praćenje plana.

- sposobnost reagovanja na promene često određuje uspešnost projekta
- planovi moraju da budu prilagodljivi promenama
- promene će sigurno nastupiti, pitanje je samo kada i koje
- planiranje ne bi trebalo da ide daleko u budućnost
  - sa tokom razvoja menjaju se i ciljevi i prioriteti

  - definišemo neke grube okvire pa posle razdrađujemo
- detaljno planiranje je neophodno, ali za kratak period
- detaljni planovi su korisni ali se teško održavaju na duži period
- naravno, vizija i definisani glavni ciljevi moraju da postoje mada se i oni mogu menjati

## 102 Navesti bar 8 principa agilnog razvoja softvera.

Manifest agilnog razvoja propisuje principe agilnog razvoja softvera:

- Najviši prioritet je zadovoljiti klijenta kroz brzo i neprekidno isporučivanje vrednog softvera.
- Uvek otvoreno prihvatiti promene, čak i u kasnim fazama razvoja. ARS uvažava promene kao sredstvo postizanja kvaliteta za klijenta.
- Obično u poslednjih par dana pre isporuke softvera nema promene
- Isporučivati funkcionalan softver što češće, sa intervalom od par nedelja do par meseci.
- Poslovni ljudi i razvijaoci moraju sarađivati svakodnevno na projektu.
- Zasnivati projekat na motivisanim pojedincima. Pružiti im okružinje i potrebnu podršku i imati poverenja da će obaviti posao.
- Najefikasniji način za razmenu informacija timu je razgovor licem u lice. Timovi bi trebalo bar da rade istoj zgradi.
- Funkcionalan softver je osnovno merilo napretka.
- ARS promoviše uzdržani razvoj. Sponzori, razvijaoci i korisnici bi trebalo da budu u stanju da neprekidno održavaju ujednačen ritam. (trebalo bi biti zabranjeno raditi prekovremeno)
- Neprekidno posvećivanje pažnje tehničkoj doteranosti i dobrom dizajnu podiže agilnost. (uzorci u projektovanju spadaju u ovu priču)
- Jednostavnost. Umetnost maksimizovanja količina posla koji se ne obavlja je od suštinskog značaja. (Uraditi što manje da softver radi kako treba, ali da ima dobar dizajn)
- Najbolje arhitekture, zahtevi i projekti potiču iz samoorganizovanih timova. (Tim treba da bude dobar, skladan, i šta je potrebno da se uradi za rezultat, a tim pusiti da razmišlja svojom glavom)
- U redovnim intervalima tim mora da salgeda svoj rad i mogućnosti unapređivanja svog ponašanja i efikasnosti.

## 103 Navesti bar 3 metodologije agilnog razvoja softvera.

- Agilno modeliranje
- Agilan objedinjen proces (AUP)
- Ekstremno programiranje (XP)
- Otvoren objedinjen proces (OpenUP)
- Scrum

## 104 Šta je ekstremno programiranje?

Jedna od najpoznatijih agilnih metodologija.

Čine da jednostavne, međusobno nezavisne metode.

Značajno zastupljen u savremenoj praksi.

## 105 Navesti bar 8 metoda ekstremnog programiranja.

- Klijent je član tima
- Korisničke celine (user stories)
- Kratki ciklusi
- Praćenje toka razvoja
- Testovi prihvatljivosti
- Programiranje u paru
- Razvoj vođen testovima
- Kolektivno vlasništvo
- Neprekidna integracija
- Uzdržan ritam
- Otvoren radni prostor
- Igra planiranja
- Jednostavan dizajn
- Metafora

## 106 Objasniti metod ekstremnog programiranja "Klijent je član tima".

Pored programera u timu mora da postoji i jedan predstavnik klijenta. Idealno bi bilo da je klijent stalno tu.

Klijenti su osobe ili grupe koje definišu ciljeve i prioritete.

Klijenti i razvijaoci moraju sarađivati.

U ekstremnom programiranju klijenti imaju ulogu člana tima.

Najbolje je da budu fizički blizu zbog ostvarivanja nepostredne komunikacije.

## 107 Objasniti metod ekstremnog programiranja "Korisničke celine (user stories)".

Korisnička celina je nešto što predstavlja jednu celinu koja se pravi. Jedna mogućnost koju će korisnik vašeg softvera moći da uradi. (Npr. u editoru teksta mogućnost da korisnik automatski formatira kod)

Radi okvirnog planiranja potrebno je sagledavati zahteve

- ali ne i potpuno precizne elemente zahteva
- neophodno je znati gde i kakvih detalja ima, ali ne i same detalje
- detalji se menjaju tokom vremena
- prve (grube) procene (rokova i troškova) služe samo za upravljanje prioriteta i nikoga ne obavezuju

Uobičajeno:

- zahtevi se iznose kao celine, u svega par reči
- okvirne procene se daju odmah ili veoma brzo

Korisnička celina služi kao referenca na nešto što će se detaljnije razmatrati kada dođe na red za implementaciju.

## 108 Objasniti metod ekstremnog programiranja "Kratki ciklusi".

Za ekstremno programiranje je uobičajena redovna:

- iteracije
  - mali korak u razvoju
  - manji nivo isporuke
  - mi smo uradili nešto i dali to klijentu, obično deo softvera koji klijent može da testira
  - uobičajeno relativno često, na primer na svake sve nedelje
- izdanja
  - na svakih nekoliko iteracija pravimo izdanje
  - veći nivo isporuke
  - obično obuhvata nekoliko iteracija, na primer 6
  - po pravilu predstavlja dorađeniji kod, koji ide u upotrebu

Plan iteracije obuhvata

- budžet i trajanje - određuju se na samom početku planiranja iteracije
- korisničke celine koje ulaze u iteraciju
  - po izboru klijenta, a uz konsultacije sa razvojnim timom
  - ne određuju se prioriteti celina u okviru iteracije
  - skup korisničkih celina ne bi trebalo da se menja tokom rada na iteraciji

Tok iteracije

- razvijaoci prave poslove i rade po redosledu koji sami odrede
- na pola trajanja iteracije snima se stanje i procenjuje se da li će iteracija biti dovršena u roku - ako neće, obaveštava se klijent i donose se odgovarajuće odluke
- na kraju svake iteracije sistem se isporučuje - može ali ne mora da ide u produkciju
- klijenti daju povratne informacije za svaku iteraciju

Plan izdanja

- uobičajeno se radi o značajnoj celini koja ide u produkciju
- sastoji se od korissničkih celina sa procenama i prioritetima
- najviše okvirno definiše sadržaje pojedinačnih iteracija
- skup korisničkih celina i njihov raspored po iteracijama se uobičajeno menja tokom rada na izdanju
- čak i boj iteracija se može promeniti
- nisu poželjne veće promene broja iteracija

## 109 Objasniti metod ekstremnog programiranja "Testovi prihvatljivosti".

Detalji o korisničkim celinama se dokumentuju u obliku testova prihvatljivosti

- Određuje ih klijent
- Piše se neposredno pre ili čak paralelno sa implementacijom iste celine
- Ako je moguće, pišu se na nekom skript jeziku, koji omogućava da se izvode automatizovano i sa ponavljanjem
- Nekad se automatizuju, vrlo često postoje okruženja koja automatizuju korisnički interfejst

Kada se test uspešno prođe, on se dodaje u kolekciju položenih testova

- Oni se ponavljaju svaki put pri izgradnji sistema (nekoliko puta dnevno)
- Tako se obezbeđuje da kada se zahtev jedanput zadovolji on više ne bude doveden u pitanje kasnijim razvojem

## 110 Objasniti metod ekstremnog programiranja "Programiranje u paru".

Sav produkcioni kod se piše u parovima od po dva programera koji rade zajedno na istoj radnoj stanici

- jedan član tima piše kod
- drugi u hodu proverava i unapređuje kod
- kada jedan piše a drugi gunđa, manje bagova pravimo
- neophodna je intenzivna saradnja
- uloge se često menjaju - i po više puta u toku jednog sata

Članovi parova se menjaju bar jedanput dnevno

- tokom iteracije svaki član tima mora
  - da radi u paru sa svim ostalim članovima
  - da radi na svim ili skoro svim delovima iteracija
  - ovim se povećava komunikacija između članova tima
  - jako brzo se razmenjuju informacije između ljudi
  - svaki član je upoznat sa svim delovima softvera

Posledice

- Veoma brzo širenje informacija o projektu među članovima tima
- Širenje znanja i veština među članovima tima
- Studije pokazuju da se ne smanjuje efikasnost programera, već se značajno smanjuje broj grešaka
- Specijalnosti i dalje ostaju na pojedincima, ali su i drugi upoznati da rezultatima, odlukama i razlozima za njihovo donošenje

## 111 Objasniti metod ekstremnog programiranja "Razvoj vođen testovima".

Produkcioni kod se piše sa ciljem da zadovolji testove jedinica (unit tests)

- Počinje se od pisanja testova koji ne prolaze zato što ne postoji odgovarajuća funkcionalnost
- Zatim se piše kod koji omogućava da testovi prođu
- Iteracije pisanja testova i koda se veoma brzo smenjuju, često na svaki minut
- Testovi i kod evoluiraju zajedno, tako da testovi budu tak nešto ispred koda

Posledice:

- Zajedno sa kodom dobija se i veoma kompletna kolekcija testova
- Kolekcija testova omogućava programeru da proverava da li jedinica koda radi ispravno ili ne
- Sprečava se nastajanje grešaka u kodu prilikom naknadnih izmena
- Pomaže se refaktorisanje
- Metod vrši pritisak da se razdvajaju jedinice koda, čime se dobija bolji dizajn projekta

## 112 Objasniti metod ekstremnog programiranja "Kolektivno vlasništvo".

Svaki par ima pravo da proveri bilo koji modul i da ga uporedi.

Nijedan programer nije pojedinačno odgovoran za bilo koji konkretan modul ili tehnologiju.

Svako radi na svim nivoima sistema, od baze podataka, preko srednjeg sloja, pa sve do korisničkog interfejsa

- to ne znači da se ne poštuju specijalnosti
- svakako će najviše raditi u svojoj specijalnosti, ali će moći da se uključi i u druge celine i da uči o drugim specijalistima

## 113 Objasniti metod ekstremnog programiranja "Neprekidna integracija".

Intenzivna upotreba sistema za upravljanje verzijama omogućava da svakako po više puta dnevno uzima i postavlja nove verzije koda

Timovi obično koriste neograničavajuću kontrolu koda

- svako može da preuzme i menja bilo koji modul
- ako je neko menjao u međuvremenu, programer je dužan da uklapa svoje izmene (merge)
- da bi se izbegla veća uklapanja koda, integrisanje se obavlja veoma često

Postupak:

- par radi na jednom poslu sat ili dva
- prave testove i produkcioni kod
- u nekom pogodnom trenutku, mnogo pre nego što je posao dovršen, kod se integriše
  - pre svake integracije proverava se da li prolaze testovi
  - ako je potrebno, obavlja se uklapanje koda
- nakon integracije se ponavlja izgradnja čitavog sistema
  - doslovnog čitavog
  - u zavistnosti od iteracije/izdanja, ako je potrebno, čak se ponavlja rezanje CD-ova, instalacija softvera,...
- zatim se ponavljaju svi testovi jedinica koda, kao i svi testovi prihvatljivosti iz kolekcije položenih

Posledice

- izgradnja sistema više puta dnevno
- testiranje izgrađenog sistema više puta dnevno
- mala verovatnoća naknadnog nastajanja greška i njihovo lako uočavanja
- značajno jednostavniji postupak debagovanja
  - testovi ukazuju da postoji problem
  - testovi ukazuju na njegovu prostornu i vremensku lokaciju
  - istorija verzija omogućava sužavanje prostorne i vremenske lokacije nastajanja problema

## 114 Objasniti metod ekstremnog programiranja "Uzdržan ritam".

"Razvoj softvera nije sprint, nego maraton." (R. Martin)

Agilni razvoj promoviše ustaljen ritam rada

Ekstremno programiranje zabranjuje prekovremeni rad

Jedini izuzetak je poslednja nedelja izdanja - "ako je tim dovoljno blizu cilja, sprint je dopušten"

Posledice:

- Bolji odnosi u timu
- Rasterećenost članova tima
- Manji broj grešaka
- Efikasnija iskorišćenost radnog vremena

## 115 Objasniti metod ekstremnog programiranja "Otvoren radni prostor".

- Tim bi trebalo da radi zajedno u jednoj velikoj otvorenoj prostoriji.
  - Na svakom stolu su dve ili tri radne stanice.
  - Za svakom radnom stanicom su po dve stolice.
  - Na zidovima su table i panoi.
- Uobičajeni ton je tiha komunikacija.
  - Svako po potrebi može da dozove bilo koga.
  - Svi znaju ako je neko zapao u probleme.
  - Svi mogu da intenzivno komuniciraju.
- Posledice
  - Iako bi se inicijalno moglo posumnjati u efikasnost rada u takvom okruženju, u praksi se pokazuje da takav radni prostor značajno podiže efikastnost.

## 116 Objasniti metod ekstremnog programiranja "Igra planiranja".

- Suština planiranja u kontekstu ekstremnog programiranja je u podeli odgovornosti između klijenta i razvijenog tima.
  - klijenti odlučuju šta je značajna karakteristika softvera
  - razvijaoci odlučuju koliko ta karakteristika košta
- Pri planiranju iteracije
  - razvijaoci određuju obim (vremena i novca) tj. daju gornju procenu mogućnosti
  - klijenti u to uklapaju korisničke celine
- Sa brzim iteracijama klijenti i razvijaoci se brzo navikavaju i ulaze u dobar ritam.

# 117 Objasniti metod ekstremnog programiranja "Jednostavan dizajn".

Bez komplikovanja unapred, pravimo samo ono što nam treba baš sad

Cilj je da dizajn bude što jednostavniji i što izražajniji

- Pažnja se posvećuje samo onome što je planirano za tekuću iteraciju.
- Ne razmatra se ono što će (možda) doći kasnije.

Dizajn sistema se menja od iteracije do iteracije.

Razvoj obično ne počinje od infrastrukture

- izbor baze podataka obično nije prva stvar
- ni izbor srednjeg sloja obično nije prva stvar
- obučno je primarno da prva grupa korisničkih celina propali na najjednostavniji mogući način
- infrastruktura se zatim dodaje prema potrebama

Tri osnovna pravila

- Razmotriti prvo najjednostavnije rešenje koje bi moglo da uradi posao.
  - primer: ako nešto može da se reši pomoću datoteka, nema razloga da se odmah upliću baza podataka ili objekti srednjeg sloja
  - primer: ako nešto može bez paralelnog izračunavanja, onda tako valja i rešiti
  - cilj je sa što manje rada i u što većem kraćem vremenu doći do cilja - dizajn će se unapređivati kasnije, ako bude potrebno.
- Neće biti potrebno
  - kada god se pretpostavlja da će biti nešto biti potrebno možda ili za neko vreme, odgovor je "neće biti potrebno"
  - nešto se dodaje samo ako je sasvim izvesno da je potrebno sada ili da će biti potrebno u vrlo skoroj budućnosti
  - cilj je izbegavati komplikovan dizajn radi nečega što možda nikad neće zatrebati
- Jedanput i samo jedanput
  - ekstremno programiranje ne dopušta ponavljanja u kodu
  - kada god se naiđe na ponavljanje, ono se mora otkloniti pravnjanjem metoda ili klase
  - čak i slučajevi kada su neki delovi koda veoma slični se moraju apstrahovati
  - najbolji put za otklanjanje redudantnosti je apstrahovanje
  - ako su neke dve stvari slične, sigurno postoji apstrakcija koja ih objedinjuje
  - otklanjanjem redudanci se promoviše pravljanje apstrakcija i smanjivanje spregnutosti (decoupling) koda

VAŽNO:

- Razmatranje najjednostavnijih rešenja i jednostavnog dizajna nikako ne znači da kod sme biti loše dizajniran.
- "Jednostavno" ne znači "na brzinu" niti "nepažljivo".

# 118 Objasniti metod ekstremnog programiranja "Refaktorisanje".

"Kod je kvarljiv" R. Martin

Jednom napisan kod nije gotov. Jednog dana ćemo ga menjati. Ako želimo da izmenimo nešto trivijalno potrebno je da analiziramo da li je ta promena zaista potrebna.

Ako postepeno menjamo dizajn kod, on se kvari.

Kada se kod dograđuje, čak i idealnom dizajnu vremenom može da opadne kvalitet.

Ekstremno programiranje se suprotstavlja opadanju kvaliteta dizajna čestim refaktorisanjem.

Refaktorisanje je preduzimanje niza malih transformacija koda kojima se unapređuje struktura koda bez promene ponašanja sistema

- svaka pojedinačna transformacija je sasvim jednostavna, skoro trivijalna
- posle svake transformacije se testira izmenjena jedinica koda
- ponavljaju se transformacije sve dok se ne dođe do čistog i dobro strukturiranog dizajna

Refaktorisanje se primenjuje

- neprekidno, u hodu
- svaki put kada se uoči da neka izmena narušava postojeći dizajn
- paralelno sea razvojem testova i produkcionog koda (refaktorisanje će biti posebna tema na jednom od narednih časova)

## 119 Objasniti metod ekstremnog programiranja "Metafora".

Ako ne znamo šta želimo projektom da postignemo, ne treba ni da ga razvijamo

Metafora je velika slika čitavog sistema

- Predstavlja viziju sistema kao celine.
- Iz nje (ne)posredno potiču svi konkretni moduli i zahtevi.
- Metafora odgovara konceptu vizije u drugim metodologijama.

Često se formalizuje u vidu rečnika pojmova koji identifikuju najvažnije koncepte sistema i problema koji bi on trebalo da reši.

- Taj rečnik pojmova je često simboličnog ili apstraktnog karaktera.

## 120 Šta je "Razvoj vođen testovima"?

Je jedan stil vođenja razvoja koji kaže sledeću stvar: Ako hoćemo nešto novo da isprogramiramo, pre nego što isprogramiramo, prvo napišimo deo koda koji će da proverava da li naš kod radi kako treba.
Prvo napišemo testove, onda napišemo kostur našeg koda (koji u tom trenutku ne radi, ali omogućava da se naš kod prevede) pa tek onda pišemo kod.
Kada pričamo o razvoju vođenom testovima prvenstveno pričamo o testovima jedinica koda.
Pisanje testova nije dokaz korektnosti koda.

## 121 Navesti i objasniti vrste testova softvera.

**Testovi jedinica koda** (Unit Test) - Testiranje pojedinačnih delova koda

**Integralni testovi** (Integration Test) - Testiranje povezanih celina koda

**Testovi sistema** (System Test) - Testiranje sistema kao celine

**Testovi prihvatljivosti** (Acceptance Test) - Testiranje sistema iz ugla korisnika. Često se izvodi pomoću skriptova koji simuliraju ili emuliraju korisnički interfejs.

## 122 Navesti i objasniti ukratko osnovne principe razvoja vođenog testovima.

- Test Driven Development
Odnosi se primarno na testove jedinica koda. Osnovni principi:

- Testovi prethode kodu
- Sistematičnost (ne sme da postoji nijedan deo koda koji nije pokriven)

# 123 Objasniti princip razvoja vođenog testovima "Testovi prethodne kodu" i način njegove primene.

Pre pisanja bilo kakvog koda, prvo se prave odgovarajući testovi.

Nijedna funkcija programa se ne razvija sve dok ne postoji test koji ne uspeva zbog njenog odsustva.

Nijedna linija koda se ne dodaje sve dok ne postoji test koji zbog nje ne uspeva.

Prvo se dodaju testovi koji ne prolaze, pa tek onda funkcionalnost koju oni zahtevaju i proveravaju.

Primena

- Svaka iteracija počinje pisanjem testova - oni često najpre ne mogu ni da se prevedu, zato što ne postoje odgovarajući metodi ili klase
- Zatim se napiše kostur koda koji omogućava da se testovi prevedu ali da ne prolaze - npr. svi metodi vraćaju neki konstantan rezultat
- Zatim se piše kod koji omogućava da testovi prođu.

Mali koraci

- Treba testirati što manje celine
- Ako postoje testovi a postoji i bag, onda testovi nisu dobri
- Iteracije pisanja testova i koda se veoma brzo smenjuju, često na svaki minut.
- Poželjno je da testovi i kod evoluiraju zajedno, tako da testovi budu tek nešto ispred koda.
- Nije dobro napisati veliki broj testova bez odgovarajućeg koda.

# 124 Objasniti princip razvoja vođenog testovima "Sistematičnost".

- Svaka karakteristika softvera se mora pokriti testovima.
- Dobro je da pokrijemo opšte slučajeve i neke specijalne slučajeve.
- Svi granični slučajevi moraju biti obuhvaćeni testovima.
- Svaka linija koda mora biti testirana.
- Naknadno uočavanje bagova obično je posledica nedovoljno sistemaičnih testova
  - da su bili dovoljno sistematični, testovi bi pokazali problem, umesto da se kasnije manifestuje kao bag
- Tehnike
  - Ako se softver razvija
    - od vrha prema dnu
      - testira se pomoću umetaka (stub)
      - privremeni delovi koda koji "implementiraju" interfejs ili druge metode tako da ne rade ništa
      - jedina namena im je da omoguće da se program prevede, iako neće raditi ispravno
    - od dna prema vrhu
      - testira se pomoću izvođača (driver)
      - privremeni delovi koda koji koriste manje funkconalne celine u odsustvu većih funkcionalnih celina čiji razvoj tek sledi

## 125 Navesti osnovne uloge testova.

Testovi su vid verifikacije

Pomaže refaktorisanje

Više uglova posmatranja koda

Testovi su vid dokumentacije

Debagovanje

- Svaki put kada se uoči neka neispravnost koja nije prepoznata postojećim testovima to znači da postojeći testovi nisu dovoljno sistematični i dodaje se novi test ili više novih testova.
- Debagovanje se svodi na zadovoljavanje testova.
- Za razliku od privremenih provera, koje se često koriste pri debagovanju, testovi ostaju kao trajne provere (i potvrde) ispravnosti.

## 126 Objasniti ulogu testova kao vida verifikacije softvera.

- Kolekcija testova omogućava programeru da proverava da li jedinica koda radi ispravno ili ne.
- Za svaku funkciju programa postoje odgovarajući testovi koji proveravaju njenu ispravnost
- Kolekcija testova predstavlja prepreku srljanju. Ukazuje na nepotpunosti ili neispravnost.
- Omogućava nam da prepoznamo da li smo možda nešto loše izmenili
- Pomaže se rano prepoznavanje grešaka kako na nivou koda tako i na konceptualnom nivou.
- Metod vrši dodatni pritisak da se razdvajaju jedinice koda, čime se dobija bolji dizajn projekta.

## 127 Objasniti ulogu testova u okviru refaktorisanja.

- Lako se prepoznaje nastajanje grešaka u kodu prilikom naknadnih izmena.
- Ako popravljamo dizajn koda, ponašanje programa mora da ostane isto.
- Onemogućava se promena ponašanja sistema pri transformisanju koda.

## 128 Objasniti ulogu testova u kontekstu ugla posmatranja koda.

- Pravljenje testova stavlja programera na mesto korisnika koda koji se piše.
- U prvom planu su interfejs jedinice koda i apstrakcija njenog ponašanja.
- Dobija se lako upotrebljiv kod.
- Dobija se lako proverljiv kod, što je kasnije veoma teško postići ako nije od početka dobro oblikovan.

## 129 Objasniti ulogu testova kao vida dokumentacije.

- Predstavljaju oblik specifikacije zahteva.
- Opisuju uslove funkcionisanja koda.
- Opisuju način upotrebe interfejsa jedinice koda - svaki test predstavlja primer upotrebe interfejsa.
- Predstavljaju karakteristične primere - obično testovi počivaju na tzv. graničnim slučajevima.
- Zajedno sa kodom dobija se i veoma kompletna kolekcija testova.
- Ovaj vid dokumentacije je uvek ažuran - prevodi se i proverava sa kodom. (najvažnija stvar)

## 130 Šta može biti jedinica koda koja se testira?

Jedinica koda može biti

- operacija, funkcija, metod
- struktura podataka
- klasa
- više manjih integrisanih jedinica koda
- softverski paket
- podsistem
- spoljašnji podsistem
- testiranje da li interfejs odgovara specifikaciji

## 131 Šta može biti predmet testiranja jedinice koda?

Funkcionalni zahtevi

Zavisnost postuslova od preduslova.

- Proverava da li jedinica koda radi ispravno za različite očekivane kategorije ulaza.
  - da li izračunava ispravan izlaz?
  - da li na ispravan način menja stanje sistema?

Robusnost

- Provera ispravnosti ponašanja u slučaju neispravnih ulaznih podataka.
- Provera da li dolazi do problema, da li dolazi do izuzetka ili je ponašanje prihvatljivo.

Integracija

- Rezultat integracije manjih jedinica koda je veća jedinica koda.
- I ona može i mora biti predmet testiranja.

Interfejs spoljašnjeg podsistema

- Provera da li interfejs odgovara specifikaciji.
- Testiramo i delove softvera koje koristimo pri izradi našeg softvera.

## 132 Navesti bar 5 biblioteka za testiranje jedinica koda u programskom jeziku C++.

- CppUnit
- CppUnitLite
- Boost.Test
- NanoCppUnit
- Unit++
- CxxTest
- Google Test
- CATCH
- CPUnit

## 133 Opisati ukratko osnovne mogućnosti biblioteke CppUnit.

Relativno jednostavna upotreba

Relativno prilagodljiv podsistem

Omogućava složene testove i okruženja za testove

Podražava izuzetke

Relativno bogat skup provera

Podržava različite načine izveštavanja o testovima

Podržava biblioteke testova

Veliki broj korisnika i solidna javna podrška

## 134 Koji su osnovni elementi koje programer pravi pri pravljenju testova uz primenu biblioteka CppUnit? Kako?

U okviru projekta

- napravi se novi cilj - program za izvršavanje testova
- može po jedan za svaki produkcioni cilj, ali može i jedan univerzalan
- doda mu se glavna funkcija tako da pokreće testove
- za svaku jedinicu produkcionog koda piše se po jedna ili više test jedinica koda
- svaki put kada se provodi ciljni produkcioni kod prevodi se i program za izvršavanje testova

**Makroi za testove** :

- Skup makroa za proveravanje uslova
- U slučaju greške izbacuju izuzetke

**Test-slučaj** (Test Case)

- Klasa sa testovima
- Nasleđuje
  - CppUnit::TestCase
- Konstruktor prosleđuje naziv
- Glavni metod je void runTest()

**Izvršavanje testova**

- Biblioteka obuhvata alate za izvršavanje testova
- Mogu se lako dodavati i prilagođavati okruženju
- Osnovni izvršavač testova, za konzolu, je CppUnit::TextTestRunner
- Dodaju mu se objekti slučajeva testova: addTest( CppUnit::TestCase* )
- Pokreće se metodom run()

## 135 Šta je Test suit?

Za lakše rukovanje većim skupovima testova postoje kompleti testova (Test Suite). Makroi prave okvir kompleta u okvir klase CppUnit::TestCase

- CPPUNIT_TEST_SUITE( VektorTest );
- CPPUNIT_TEST( constructorTest );
- CPPUNIT_TEST_SUITE_END();

Izvršavanje kompleta testova

- Makroi za prijavljanje okvira implementiraju statički metod za pravljenje objekta kompleta testova: suite()
- Izvršavaču se ne dodaje objekat nego statički napravljen komplet: runner.addTest( VektorTest::suite() );

# 136 Navesti osnovne vrste pretpostavki koje podržava biblioteka CppUnit.

**CPPUNIT_ASSERT** (*uslov*)
- Pretpostavka da je uslov zadovoljen

**CPPUNIT_ASSERT_MESSAGE** (*poruka, uslov*)
- Pretpostavka da je uslov zadovoljen
- Korisnički definisana poruka koja se ispisuje ako uslov nije ispunjen

**CPPUNIT_FAIL** (*poruka*)
- Eksplicitan neuspeh, sa datom porukom

**CPPUNIT_ASSERT_EQUAL** (*očekivana, stvarna*)
- Pretpostavka da su date dve vrednosti jednake
- Moraju da budu
  - istog tipa
  - da se mogu proslediti u izlazni tok
  - mogu se porediti operatorom ==

**CPPUNIT_ASSERT_EQUAL_MESSAGE** (*poruka, očekivana, stvarna*)
- Kao...ali sa datom porukom

**CPPUNIT_ASSERT_DOUBLES_EQUAL** (*očekivana, stvarna, delta*)
- Pretpostavka da su date dve vrednosti jednake, sa dopuštenom greškom delta

**CPPUNIT_ASSERT_THROW** (*izraz, TipIzuzetka*)
- Pretpostavka da izračunavanje datog izraza izbacuje izuzetak datog tipa TipIzuzetka

**CPPUNIT_ASSERT_NO_THROW** (*izraz*)
- Pretpostavka da izračunavanje izraza ne izbacuje izuzetak

**CPPUNIT_ASSERT_ASSERTION_FAIL** (*pretpostavka*) - Pretpostavka da data pretpostavka ne uspeva
- koristi se za testiranje pretpostavki

**CPPUNIT_ASSERT_ASSERTION_PASS** (*pretpostavka*)
- Pretpostavka da data pretpostavka uspeva - koristi se za testiranje pretpostavki

# 137 Šta su testovi prihvatljivosti?

Testovi prihvatljivosti su konceptualno drugačiji od testova jedinica koda

- Odnose se na testiranje ponašanja sistema kao celine
- Ne implementiraju se istim alatima
- Uobičajeno se definišu nekim specifičnim skript jezikom - relativno često se upotrebljava XML
- Obuhvataju sve aspekte upotrebe sistema
  - definisanje podataka
  - izvršavanje operacija
  - proveravanje rezultata izvršavanja operacija

## 138 Po čemu se testovi prihvatljivosti razlikuju od testova jedinica koda?

Testovi prihvatljivosti testiraju da li ceo program radi ono što treba, a testovi jedinica koda, upravo da li deo koda radi ono što treba.

## 139 Ko od učesnika u razvoju softvera piše testove jedinica koda? A testove prihvatljivosti?

Testove jedinica koda pišu programeri a testove prihvatiljivosti članovi razvojnog tima ali ne nužno programeri, obično su to oni ljudi koji nisu direktno uključeni u samo pisanje programa.

## 140 Kakav je odnos refaktorisanja i pisanja programskog koda?

Tehnika koja omogućava da popravljamo kod, da ispravljamo greške, da dizajn bude bolji, lakši za održavanje i slično. Ako je dobro strukturiran, lako ćemo prepoznati šta treba izmeniti i uneti tu izmenu.

Imamo delove koda koji su nekako povezani, a refaktorisanjem menjamo način kako su povezani, a da se pritom ne menja ponašanje programa. Dakle, menjamo samo strukturu.

Refaktorisanje ne menja vidljivo ponašanje, već samo strukturu.

Pisanje novog koda ne menja postojeći kod već samo dodaje novo ponašanje.

Programiranje se sastoji od naizmeničnog refaktorisanja i pisanja novog koda.

## 141 Šta su osnovni motivi za refaktorisanje koda?

Refaktorisanje podiže nivo kvaliteta dizajna softvera.
- to je i osnovni motiv za njegovo preduzimanje

Refaktorisanje čini softver lakše razumljivim.

Refaktorisanje pomaže u traženju grešaka.

Refaktorisanje podiže brzinu pisanja koda.

## 142 Kada se pristupa refaktorisanju koda?

Treba refaktorisati redovno.

Potencijalno svaki put pre dodavanja novog koda.

" Ako se nešto ponovi pre po treći put"

Kada se dodaju nove funkcije.

Kada je potrebno pronaći bag. Refaktorišemo samo one delove za koje smo sigurni, inače treba izbegavati refaktorisanje.

Kada se proverava ispravnost i kvalitet koda.

# 143 Na osnovu čega se odlučuje da je potrebno refaktorisati neki kod?

Veoma je nezahvalno definisati uslove koje bi neki kod trebalo da zadovoljava da bi ga trebalo refaktorisati. Umesto toga, izdvajaju se kandidati u kojima je potrebno razmotriti da li se refaktorisanjem može dobiti pomak u kvalitetu dizajna koda.

- "Ako kod zaudara, potrebno ga je izmeniti." (" If it stinks, change it!")
  - U pitanju je subjektivna kategorija.
  - Osetljivost programera na određene probleme nije ista
    - Programer sa većim iskustvom će bolje umeti da prepozna moguća unapređenja.
    - Programer može neke transformacije koda smatrati za trivijalne, pa zato odgovarajuća transformisanja koda odlagati kao manje značajna.
  - Da li će se zaista pribeći refaktorisanju zavisi kako od moguće dobiti, tako i od sposobnosti programera da prepozna dobiti i pogodne tehnike refaktorisanja
    - ako nije jasno da je nekom tehnikom moguće ostvariti dobit, ili nije jasno da se tehnika može ispravno primeniti u datom slučaju, možda je bolje ne primeniti je.

# 144 Nabrojati bar 10 slabosti koda (tzv. zaudaranja) koje ukazuju da bi trebalo razmotriti refaktorisanje?

Zaudaranja su različite vrste signala koje nam govore da treba menjati kod.

- Ponavljanje koda
- Velika klasa
- Dugačka lista argumenata
- Divergentne promene
- Distribuirana apstrakcija
- Velika zavisnost od drugih klasa
- Grupisanje podataka
- Poplava primitivnih podataka
- Naredba switch
- Paralelne hijerarhije nasleđivanja
- Lenje klase
- Spekulativno uopštavanje
- Privremeni podaci
- Lanci poruka
- Posrednik
- Nepoželjna bliskost
- Alternativne klase sa različitim interfejsima
- Nepotpuna biblioteka
- Klasa podatak
- Nepoželjno nasleđe
- Komentari

## 145 Zašto je dobro eliminisati ponavljanja iz koda? (refaktorisanje)

Jedan od najčešćih i osnovnih problema. Ako se isti ili sličan kod ponavlja više puta, potrebno je razmotriti apstrahovanje.

- Izdvajane metoda
- Povlačanje metoda hijerarhiju
- Pravljenje šablonskih metoda
- Zamena algoritma
- Izdvajanje klase

## 146 Zašto dugački metodi mogu predstavljati problem? (refaktorisanje)

Ako je metod suviše dugačak, sva je prilika da ga je potrebno razložiti na više manjih. Pre svega radi lakšeg održavanja

- razumevanje
- debagovanje
- menjanje
- Izdvajanje metoda
- Zamena privremenih vrednosti upitima
- Uvođenje parametarskog objekta
- Zamena metoda objektom
- Dekompozicija uslova

## 147 Zašto velika klasa može da predstavlja problem? (refaktorisanje)

Velika klasa može postojati sa razlogom. Ako se pravi mnogo objekata neke klase i to za različite namene, to može biti signal da je klasu potrebno podeliti na više klasa

- Izdvajanje klase
- Izdvajanje podklase
- Izdvajanje interfejsa
- Pojavljanje praćenih podataka

## 148 Šta su divergentne promene? Zašto su problematične? (refaktorisanje)

Termin divergentne promene označava slučajeve u kojima se neka klasa menja iz više različitih razloga. To obično znači da razlozi promena nisu dovoljno apstrahovani.

- Izdvajanje klasa

## 149 Šta je distribuirana apstrakcija? Zašto je problematična? (refaktorisanje)

U nekim slučajevima se zbog neke promene moraju praviti brojne male izmene u mnogo različitih klasa. To obično znači da je odgovarajuća apstrakcija distribuirana u više klasa, što nije dobro.

- Premeštanje metoda
- Premeštanje podataka
- Umetanje klase

## 150 Zašto velika zavisnost neke klase ili metoda od drugih klasa može da predstavlja problem? (refaktorisanje)

Za klasu ili metod kažemo da mnogo zavisi od drugih klasa ako koristi veliki broj metoda druge klase za čitanje vrednosti. To obično znači da je neka odgovornost podeljena između više klasa na neodgovarajući način.

- Premeštanje metoda
- Izdvajanje metoda

## 151 Zašto naredba switch može da predstavlja problem? (refaktorisanje)

Problem sa upotrebom naredbe switch je u tendenciji da se ponavljaju kriterijumi i grananja. Često se to može prevazići hijerarhijama klasa.

- Izdvajanje metoda
- Premeštanje metoda
- Zamena kodiranog podataka potklasom
- Zamena uslova polimorfizmom
- Zamena parametara eksplicitnim metodom
- Uvođenje praznih objekata

Vrlo često može da se zameni hijerarhijom klasa.

## 152 Šta je spekulativno uopštavanje? Zašto može da prestavlja problem?(refaktorisanje)

Ako je kod apstrahovan zato što se pretpostavlja da bi raznolikost mogla nastupiti ali zapravo ne postoji. Često je suvišna apstrahovanost otežava održavanje pa ju je potrebno ukloniti.

- Sažimanje hijerarhije
- Umetanje klase
- Uklanjanje argumenta
- Preimenovanje metoda

## 153 Zašto privremene promenljive mogu da predstavljaju problem? (refaktorisanje)

Ako se neka privremena promenljiva koristi samo u nekim slučajevima, to može otežati razumevanje koda.

- Izdvajanje klasa
- Uvođenje praznog objekta

## 154 Zašto lanci poruka mogu da predstavljaju problem? (refaktorisanje)

Lanci poruka ukazuju na veliki broj posrednika u obavljanju nekog posla. Potrebno je razmotriti uklanjanje suvišnih posrednika.

- Skrivanje delegata
- Izdvajanje metoda
- Premeštanje metoda

## 155 Zašto postojanje klase posrednika može da predstavlja problem? (refaktorisanje)

Skrivanje internih detalja je jedna od osnovnih karakteristika OO programiranja. Međutim, može se otići predaleko. Ako čitava klasa predstavlja samo posrednika, bez značajne dopune u ponašanju, potrebno je razmotriti njeno uklanjanje.

- Uklanjanje posrednika
- Umetanje metoda
- Zamena delegiranja nasleđivanjem

## 156 Šta je nepoželjna bliskost? Zašto je problematična? (refaktorisanje)

Nepoželjna bliskost nastupa kada neka klasa suviše često pristupa privatnim delovima neke druge klase.

- Premeštanje metoda
- Premeštanje podataka
- Promena dvosmernog odnosa u jednosmeran
- Izdvajanje klase
- Skrivanje delegata
- Zamena nasleđivanjem delegiranjem

## 157 Kakav je odnos agilnog razvoja softvera i pisanja komentara? Zašto komentari mogu da budu motiv za refaktorisanje?

Komentari u implementaciji metoda sugerišu da kod bez njih nije dovoljno jasan, tj. da nije dovoljno dobro dizajniran.
" Komentari imaju dobar miris, ali često se koriste kao dezodorans za kod koji zaudara."

- Izdvajanje metoda
- Preimenovanje metoda
- Uvođenje pretpostavke

## 158 Šta bi trebalno da sadrži opis svakog od refaktorisanja u katalogu?

Sistematizacija refaktorisanja podrazumeva ujednačeno opisivanje tehnika. Svaka tehnika ima

- ime - isto kao kod uzoraka
- sažet opis slučaja u kome se primenjuje
- motivaciju - koju korist možemo očekivati od tehnike koju primenjujemo
- opis tehnike izvođenja - ovo je veoma važno
- primer

## 159 Navesti bar 5 grupa tehnika refaktorisanja.

Sva refaktorisanja su grupisana po nameni u:

- Metode komponovanja koda
- Premeštanje koda između objekata
- Organizovanje podataka
- Pojednostavljivanje uslovnih izraza
- Pojednostavljivanje pozivanja metoda
- Razrešavanje uopštavanja
- Velika refaktorisanja

## 160 Navesti bar 5 tehnika refaktorisanja.

- Preimenotavi metod
- Inline-ovati metod
- Inline-ovati klasu
- Enkapsulirati polje
- Enkapsulirati promenljivu

## 161 U kojim slučajevima refaktorisanje može biti značajno otežano?

Složen dizajn Refaktorisanje velikog, nefaktorisanog koda je veoma teško.

## 162 Kako i zašto može biti otežano refaktorisanje u prisustvu baze podataka?

Menjanje strukture podataka ima široko uticaja. Migracija podataka je veoma neugodna. Treba menjati i bazu kod istovremeno. Jedan od prinicpa Agilnog razvoja je ne praviti bazu pre nego što je potrebna. Prvo pokušai sa datotekama.

## 163 Zašto može biti otežano refaktorisanje spoljnog interfejsa neke klase?

Dok se promene odnose na implementaciju nekof sistema, sve je relativno jednostavno. Kada se počne menjati interfejs, potrebna je prava mera. Ako se menja javni interfejs, najčešće je potrebno privremeno sačuvati stari interfejs....

## 164 U kojim slučajevima refaktorisanje može da ne predstavlja dobro rešenje?

Ako su problemi višestruki i međusobno zavisni. Ako se ne mogu primenjivati manji koraci be remećenja postojećeg ponašanja. Ako postojeći kod ne radi. Ako smo blizu kraja roka.

## 165 Kakav je odnos refaktorisanja i performansi softvera?

Podizanje nivoa strukture koda često ima za posledicu spuštanje nivoa performansi. Performasne se ne smeju zanemariti. Ipak

- stepen negativnog uticaja na performacije je obično daleko niži od ostvarenog pozitivnog uticaja na dizajn
- pisanje strukturiranog koda je daleko efikasnije nego pisanje optimizovanog koda
- troškovi razvoja često su važniji od troškova eksploatacije

Više metoda

- dobro strukturiranje sa definisanjem ciljnih performansi svake od komponeneti
- nakon dostizanja funkcionalnosti pristupa se optimizaciji u meri u kojoj je potrebno za dostizanje ciljnih uslova
- kontinualno staranje o performansama
- tokom čitavog razvoja svi se staraju da uvek ponude optimalna rešenja
- u praksi je obično bolji prvi metod
- funkcionalnost pre performansi
- performanse se planski podižu u skladu sa dizajnom
- neoptimizovan program se lakše održava i menja

## 166 Šta su bagovi?

Propusti u razvoju softvera se često nazivaju greškama ili bagovima.
Problemi sa terminima: Često se razvoj softvera poistovećuje sa programiranjem. Postojanje grešaka ili bagova se često neopravdano povezuje sa propustima u programiranju. To može biti i propust u projektovanju, planiranju.
Greška u razvoju softvera ima mnogo šire značenje. Obuhvata sve propuste koji mogu da nastanu u svim fazama razvoja softvera.
Termini propust i greška se obično eksplicitno koriste u širem konekstu razvoja, a termin bag u užem konteksta programiranja.

Propust (greška, bag) u razvoju softvera je sve ono što stvara probleme u funkcionisanju softvera kao završnog proizvoda.
"Sve ono što ima za posledicu da se softver ne ponaša u skladu sa specifikacijom"
"Sve ono što ima za posledicu da se softver ne ponaša u skladu sa specifikacijom ili očekivanjem korisnika"

## 167 Navesti jednu specifikaciju bagova i objasniti je.

Jedna od uobičajenih klasifikacija je prema načinu ispoljavanja:

- nekonzistentnosti u korisničkom interfejsu
- neispunjena očekivanja
- slabe performanse
- padovi sistema (program) ili oštećenja podataka

## 168 Šta su nekonzistentnosti u korisničkom interfejsu i kakve uzroke i posledice imaju?

Nedoslednosti u korisničkom interfejsu mogu da naprave mnogo neugodnih posledica.
Nije samo greška u softveru već i u planiranju.
Primer: svi programi za MS Windows koriste prečicu $Ctrl + F$ za traženje. Program Outlook koristi tu prečicu za prosleđivanje poruke (forward).
MS Word za OS/2. (Ako hoćemo da izađemo iz programa a nismo sačuvali izmene, OS/2 kaže neću da sačuvam po defaultu, a Microsoft word kaže da hoću).

Najčešći uzroci: Propusti pri projektovanju korisničkog interfejsa ili čak ustanovljavanja osnovnih koncepata softvera.

## 169 Šta su neispunjena očekivanja i kakve uzroke i posledice imaju?

Dobijanje neočekivanog (pogrešnog) rezultata je među najneugodnijim bagovima.
Podvrste:

- tip 1: davanje pogrešnog pozitivnog rezultata
- tip 2: davanje pogrešnog negativnog rezultata
- tip 3: davanje neispravnog rezultata izračunavanja

Najčešći uzroci:

- greške u komunikaciji, kada razvojni tim ne razume ispravno potrebe klijenata
- greške u projektovanju, kada projektanti naprave propuste
- greške u kodiranju, kada programeri naprave greške u samom kodu

## 170 Objasniti problem slabih performansi i moguće uzroke.

Slabe performanse mogu da prilično frustriraju korisnika zbog stalnog ili povremenog iščekivanja rezultata usled slabog odziva sistema.
Vode potpunoj neupotrebljivosti programa/sistema.
Najčešći uzroci:

- greške u proceni opterećenja
- greške u proceni raspoloživih resursa
- greške u projektovanju rešenja
- greške u kodiranju rešenja

## Padovi softvera i oštećenja podataka

Padovi softvera i oštećenja podataka su najopasniji vid bagova.
Mogu ostaviti trajne posledice po sistem i/ili podatke.
Najčešći uzroci:

- greške u projektovanju načina upotrebe podataka
- greške u kodiranju
- greške u povezivanju komponenti sistema

## 171 Koje okolnosti posebno pogoduju nastanku bagova? Objasniti dve.

Nedovoljna stručnost razvojnog tima

- neobučenost članova tima
- nerazumevanje zahteva
- izostajanje posvećenosti kvalitetu
- pristup "kodiraj pa razmišljaj"
- ...

Nepotrebno povećan nivo stresa u timu

- kratki ili čak nemogući rokovi
- prekovremeni rad
- nizak nivo komunikacije u samom timu
- ...

## 172 Koje okolnosti smanjuju verovatnoću nastajanja bagova? Objasniti dve.

Visoka stručnost tima

- stalno usavršavanje članova tima
- dobra komunikacija sa klijentom
- sistematičnost

Posvećenost kvalitetu

- testovi prihvatljivosti
- testovi jedinica koda
- pisanje robusnog softvera
- upravaljanje rizicima

# 173 Koje okolnosti olakšavaju pronalaženje uzroka bagova? Objasniti dve.

Informisanost

- čuvanje istorije verzija programskog koda
- čuvanje komunikacije članova tima, kako međusobne tako i sa klijentom
- redovno pisanje neophodne dokumentacije

Sistematičnost i redovnost

- često i plansko građenje koda
- posvećivanje pažnje upozorenjima koja se dobijaju od prevodilaca
- uvođenje i poštovanje pravila kodiranja i komentarisanja

# Debagovanje.

Otklanjanje grešaka uobičajeno se naziva *debagovanje*.
Čine ga:

- uočavanje da postoji greška
- razumevanje greške
- lociranje greške
- ispravljanje greške

Najčešće je najteži deo posla ispravno razumevanje i račno lociranje greške. Jednom kada se greška locira, njeno ispravljanje obično nije poseban problem.

## "Neformalno" debagovanje.

Jednostavan i površan pristup:

1. Pokušati sa nekom jednostavnom popravkom
2. Ponavljati korak 1 dok se problem ne reši

Motivacija: neke greške, koje se uoče tokom pisanja koda, često mogu da se relativno lako otklone

- sintaksne greške
- propuštanje da se izmene vrednosti nekih promenljivih
- pogrešne granice blokova koda
- neispravni indeksi
- i slično...

Niska efikasnost: ovakav pristup "radi" samo u jednostavnim slučajevima u složenijim slučajevima proizvodi nove probleme.

### Empirijski naučni metod

Postupak sličan uobičajenom istraživačkom metodu u prirodnim naukama:

- Posmatrati uočen problem
- Postaviti hipotezu o uzroku problema
- Na osnovu hipoteze napraviti predviđanje ponašanja
- Eksperimentalno proveriti ispravnost predviđanja
- Ponavljati prethodne korake, iz popravljanje ili zamenjivanje hipoteze, sve dok se ne potvrdi ispravnost hipoteze ili ne ponestanu mogućnosti za njeno dalje unapređivanje

Uopšteno posmatrano, ovo je najbolji pristup debagovanju.
Problem je što nije dovoljno jasno šta je sledeće što je potrebno da se uradi.
Problem se može posmatrati na mnogo različitih načina, mogu da se posmatraju različiti aspekti problema.

### Heurističko debagovanje

Heurističko debagovanje počiva na primeni nekog skupa pravila.
Predstavlja nadogradnju empirijskog metoda.
Glavni ciljevi primene pravila su:

- usmeravanje posmatranja prema uzroku problema
- izbegavanje pravljenja previda pri posmatranju
- sužavanje skupa kandidata za iskazivanje hipoteza

## 174 Navesti bar 6 osnovnih pravila debagovanja.

1. Razumeti sistem
2. Navesti sistem na grešku
3. Najpre posmatrati pa zatim razmišljati
4. Podeli pa vladaj
5. Praviti samo jednu po jednu izmenu
6. Praviti i čuvati tragove izvršavanja
7. Proveravati i naizgled trivijalne stvari
8. Zatražiti tuđe mišljenje
9. Ako nismo popravili bag, onda on nije popravljen

## 175 Objasniti pravilo debagovanja "Razumeti sistem".

"Da bi se razumeo sistem potrebno je dobro razumeti sistem"
Razumevanje sistema nije isto što i razumevanje problema. Predstavlja preduslov za razumevanje problema.
Osnovni aspekti razumevanja sistema:

- Čitanje uputstva za sistem i komponenti koje ga čine (!!! oprezno, uputstva mogu da budu neispravna!!! - potencijalno problematična ažurnost dokumentacije)
- Detaljno čitati uputstva, ne ostajati na okvirnom upoznavanju sa konceptima
- Razumeti šta je očekivano normalno ponašanje

- Razumeti tokove podataka i aktivnosti
- Poznavai alate
- Obratiti pažnju na detalje

## 176 Objasniti pravilo debagovanja "Navesti sistem na grešku".

Grešku je potrebno ponoviti iz više razloga:

- Da bi se mogla posmatrati
- Da bi se mogla posvetiti pažnja uzorcima
- Da bi se moglo proveriti da li je greška otklonjena

Greška se može ponoviti na više načina:

**Ponoviti postupak -** pouzdan način da se greška ponovi je da ponovite postupak pred publikom dokumentovati korake koji vode problemu

**Početi od početka -** nekada je neposredan uzrok lako ponoviti, ali je priprema okruženja za njegovo ponavljanje komplikovana

**Simulirati problematičnu sekvencu -** ako je teško ili sporo manuelno ponavljati postupak, automatizovati ponavljanje

**Ne simulirati greški nego uslove u kojima se ispoljava** - simuliranje mehanizma suviše bliskih uzorku može često zaobići neispravan deo koda

**Ne odbacivati problem kao "nemoguć"**

**Čuvati napravljene alate -** mogu zatrebati ponovo

Šta ako se problem ispoljava samo povremeno? Nrp. jednom u 5, 10 ili 500 ponavljanja.
To znači da nam nisu dovoljno dobro poznate okolnosti ispoljavanja.
Potrebno je:

- razmotriti razne nekotrolisane uslove:
  - neinicijalizovane podatke
  - slučajno generisane podatke
  - ulazne podatke
  - vezanost za trenutak ili trajanje izvršavanja
  - sinhronizaviju niti
  - spoljašnje uređaje

- kontrolisanje uslova često vodi nemogućnosti ispoljavanja greške (to znači da jedan od kontrolisanih uslova u nekom specifičnom stanju (različitom od kontrolisanog) proizvodi problem)
- nekada takve uslove nije moguće kontrolisati, ali ih je moguće učiniti proizvoljnijim (i to može da pomogne u razumevanju uslova nastajanja problema, ali može i da proizvede greške)

Šta ako smo sve pokušali, ali problem se i dalje ispoljava samo povremeno?
Važno je imati na umu da problem nije "svojeglav", već ima vrlo precizan uzrok, koji se sigurno može pronaći. Pokušaćemo da ispunimo ciljeve ponavljanja bez samog ponavljanja.
Kako?
Što je teže ponoviti problem, to je potrebno pažljivije posmatrati slučajeve kada se ponovi, prikupljati i anilizirati informavije, uočiti uslove koji su uvek povezani ili nikada nisu povezani sa ponavljanjem problema.
Da bi se (relativno) pouzdano ustanovilo da je neki problem rešen, potrebno je sekvencu ponoviti statistički značajan broj puta.

# 177 Objasniti pravilo debagovanja "Najpre posmatrati pa tek onda razmišljati".

Razmišljanje bez dovoljno informacija je suštinski promašaj u pristupu. Može da vodi iskrivljenom tumačenju podataka.
Dok se greška ne vidi, ne donositi nikakve zaključke.
Posmatrati detalje, svako posmatranje problema donosi saznanja o uslovima ili mehanizmu nastajanja problema.
Praviti alate koji ističu problematične uslove ili problematičan deo koda, razmatrati neke vidove takvih alata još u fazi projektovanja, npr. trajno zapisivanje informacija o problemima.
Omogućiti automatsko razlikovanje poruka radi lakšeg pretraživanja i analiziranja. Dodati takve alate tokom debagovanja, ne bežati od privremenog zamenjivanja delova koda.
Koristiti spoljašnje alate: softverski debageri, hardverski alati.

Čuvati se efekta posmatrača

- "Svako posmatranje menja sistem zato što su i posmatranja sastavni deo sistema"
- Testiranje i menjanje koda može da ima neugodne posledice, od promene ponašanja pa do prikrivanja grešaka
- Sve izmene i testove praviti u sasvim malim koracima, kako bi se smanjila mogućnost prikrivanja problema

Pretpostavljati samo radi boljeg fokusiranja pri traženju

- Ako se donese pretpostavka o uzroku greške, ne koristiti je za "otklanjanje greške" već samo za fokusiranje traženja greške na neki aspekt problema
- Izuzetak: neke greške su i česte i lake za otklanjanje - samo tada može imati smisla pokušati sa otklanjanjem pre pouzdanog potvrđivanja uzroka problema, ali i tada samo ako postoji pouzdano sredstvo za proveru da li je otklonjen, tj. ako je moguće ponoviti problem

# 178 Objasniti pravilo debagovanja "Podeli pa vladaj".

Ako je sistem složen (a praktično uvek jeste) nije moguće sagledati sve elemente sistema jednako detaljno. Potrebno je obratiti pažnju samo na delove koji povezuju uzrok problema i uočene posledice.
Kako?

- Sužavati oblast traženja uzastopnim aproksimacijama
  - postavljati hipoteze o mestu ili uslovima pojavljivanja problema, zatim testovima ustanovljavati tačnost hipoteze
  - birati hipoteze tako da što uspešnije dele mogući prostor problema
- Pronaći prave okvire prostora za traženje problema
  - pre sužavanja potrebno je postaviti deovoljno široke početne okvire
  - neophodno je pouzdano proveriti da su početni okviri dovoljno široku
  - ovo je posebno problematično ako je okvir potencijalno širi od jednog računarskog sistema
- Prepoznati na kojoj strani je greška
  - hipoteza deli prostor na dva dela
  - mora biti jasno u kom od tih delova je problem
  - ako nije, onda hipoteza nije korisna
- Koristiti test-uzorke koji se lako uočavaju
  - često nije lako uočiti greške na "živim" podacima

  - potrebno je napraviti veštačke uzorke za koje je poznato kako bi trebalo da izgleda rezultat
  - poželjno je da uzorci budu što manji i da mehanizam provere ispravnosti ponašanja bude što jednostavniji
- Početi od greške pa ići prema uzrocima
  - obično je bolje traženje započeti od mesta ispoljavanja greške
  - mogućih uzroka obično ima mnogo i nije isplativo proveravati jedan po jedan
- Popraviti prepoznate greške i nastaviti traženje
  - bagovi često idu zajedno
  - ako smo pronašli jedan, to ne znači da je i jedini
  - obavezno pokušati ponavljanje problema
- Otkloniti šum
  - ako je moguće, potrebno je privremeno "smiriti" delove sistema koji proizvede visok nivo dinamičnosti, kako ne bi odvlačili pažnju
  - jedan bag može biti uzrok drugim bagovima
  - jedan način da se prikriju drugi bagovi je privremeno isključivanje delova koda koji nisu neposredno vezani sa mestom ispoljavanja posmatranog problema

# 179 Objasniti pravilo debagovanja "Praviti samo jedno po jednu izmenu".

Ako se načine dve izmene, teško je ispravno oceniti njihov efekat.

- možda će jedna i rešiti problem, ali druga napraviti novi?
- možda je samo jedna dovoljna?
- možda nijedna nije dobra?
- možda one i rešavaju problem ali nisu ispravne?

Saveti?

- Izolovati ključni faktor
  - praviti samo dobro izolovane i lokalizovane izmene, to je osnovni preduslov za ispravno zaključivanje o rezultatima izmene
  - više izmena koje nisu lokalizovane u kodu značajno otežavaju uočavanje posledica
  - ako je zaista uočeno u čemu je problem, jedna izmena je dovoljna, a ako nije, onda je potrebno dalje posmatrati
- Biti uzdržan
  - kada se uoči da se nešto dešava, potrebno je zastati i dobro sačekati sa reagovanjem dok se ne uoče svi simptomi i ne donese nedvosmislen zaključak
  - u suprotnom se sasvim verovatno radi o nagađanju i reagovanju na osnovu nagađanja
- Menjati i isprobavati samo jednu stvar
  - ako pokušaj popravke ili testiranje nije pomogao, prvo vratiti kod u početno stanje, pa tek onda praviti nove izmene koda
- Porediti sa ispravnim slučajem
  - poređenje uslova pri kojima nastaje problem i uslova pro kojima nema problema je jedan od najkorisnijih postupaka pri razrešenju bagova

  - porediti:
    * kod
    * tragove izvršavanja
    * dnevnike grešaka
    * i sve ostale raspoložive informacije

- Ustanoviti šta je izmenjeno od poslednje poznate situacije u kojoj je ponašanje bilo ispravno
  - nekada naizgled nebitna izmena može da ima značajan uticaj na sistem
  - ako je sistem nekada radio ispravno, pravi put je u lokalizovanju intervala u kome je počelo pojavljivanje problema
  - lokalizovanje izmena nastalih u tom intervalu

## 180 Objasniti pravilo debagovanja "Praviti i čuvati tragove izvršavanja".

Nekada su naizgled beznačajni faktori od presudnog uticaja na ispravnost rada sistema. Primer: službenik za podršku često ne može da ustanovi zašto diskete prestaju da rade posle prve upotrebe dok ne vidi kako se tačno rukuje disketama

- Zapisujte šta radite, kojim redom, kao i šta su uočene posledice. Pisani tragovi o aktivnostima imaju veliki značaj za kasnije lokalizovanje događaja u vremenu
- Važno je razumeti da svaki od detalja može biti onaj koji je važan, pisani trag nikad nije suviše detaljan
- Povezivati događaje. Međusobnim povezivanjem simptoma i ishoda (bilo uspešnih ili neuspešnih) olakšava se uočavanje uzorka o problemu
- Dnevnici sa tragovima izvršavanja su važni za testiranje, na primer, ostavljanje tragova pri aktiviranju delova koda ili obavljanju određenih poslova, otvaranju datoteka i sl.
- Zapisujte, koliko god se to činilo teško "najkraća olovka je duža od najdužeg pamćenja"

## 181 Objasniti pravilo debagovanja "Proveravati i naizgled trivijalne stvari".

"Proveriti da li je uopšte uključeno"
Rešenja problema su često sasvim jednostavna ... ali je često veoma teško to uočiti.
Dovoditi u pitanje pretpstavke. Ako smo pretpostavili da je neka komponenta ispravna, to ne znači da ona to i jeste. Nikada nije dobro biti ubeđen da su pretpostavke ispravne, posebno ako su u središtu nekog neobjašnjivog problema.
Početi od početka. Da li su uslovi za obavljanje posla ispunjeni?
Primeri:

- pita se nije ispekla - da li je rerna uključena?
- automobil ne može da se pokrene - pre nego što ga rastavimo, možda bi valjalo proveriti da li ima goriva?
- ako ne inicijalizujemo podatke eksplicitno, najgore je što će sistem možda ponekad raditi ispravno

Proveriti alate

- ako smatramo da alat nešto radi na neki način, to ne znači da to nije potebno i proveriti, posebno ako postoji problem koji ne razumemo
- ako alat pokazuje da je sve u redu, a mi vidimo da nije, možda alat nije ispravan ili ne umemo da ga koristimo

## 182 Objasniti pravilo debagovanja "Zatražiti tuđe mišljenje".

Ne ustručavati se da tražite tuđu pomoć.
Uključite sveže snage. Mnogo vremena provedenog uz neki sistem stvara implicitne (podsvesne) pretpostavke koje ometaju objetkivno rasuđivanje. Neopterećen nebitnim detaljima neko "sa strane" će često lakše uočiti uzrok problema.
Za (skoro) svaku oblast postoje eksperti. Umesto da gubimo sate (dane? nedelje?) mnogo je lakše i jevtinije zatražiti pomoć eksperta. Eskperti znaju gde je potrebno "udariti čekićem".

Pomoć je dostupna na raznim stranama. Sasvim je moguće da je neko već imao sličan problem, štaviše, možda je već dokumentovan. Potražiti "vodiče kroz probleme" (*eng.* **troubleshooting guides**). Potražiti diskusione grupe na Internetu.

Ne valja biti (suviše) ponosan. Greške se dešavaju, ponos treba da pomogne da se istraje na rešavanju problema, ali ponos ne treba da sprečava da se potraži pomoć. Ipak, nekada ni drugi nisu u pravu - računati i na to. Izveštavati o simptomima, a ne o pretpostavkama, ako uključujemo nekoga u rešavanje problema, potrebno ga je snabdeti samo pouzdanim informacijama.

## 183 Objasniti pravilo debagovanja "Ako nismo popravili bag, onda on nije popravljen".

Ako simptomi nestanu sami od sebe, to ne znači da je problem rešen - uobičajeno je da se pojavi ponovo kada nam to bude najmanje odgovaralo.
Proveriti da li je zaista popravljen. Ako smo pratili ranija pravila, onda znamo kako da ponovima problem, pa, šta se dešava kada pokušamo da ga ponovimo?

Kada pomislimo da smo rešili problem, neophodno je da proverimo da li smo ga zaista rešili načinjenim popravkama ili je nešto drugo u pitanju?
Pokušajmo da ponovimo problem bez naše ispravke, ako se tada pojavljuje, a sa ispravkom ne, onda smo ga rešili. Inače su u pitanju izmenjene okolnosti i sasvim je verovatno da je naše "rešenje" beznačajno. Naravno, ovo nekada nema smisla.

Problemi nikada ne nestaju sami od sebe. Ako je problem nestao "sam od sebe" to samo znači da nismo dovoljno dobro upoznali okolnosti pod kojima se pojavljuje. Te nepoznate okolnosti su se promenile, sasvim je verovatno da će se u nekom trenutku opet promeniti i da će se problem ponovo pojaviti.

Popraviti uzroke problema. Ako pregori osigurač, zamenjivanjem osigurača ćemo samo privremeno rešiti problem. Potrebno je pronaći uzroke pregorevanja, ili će i nov osigurač pregoreti.
Popraviti razvojni proces. Ako je problem nastao, znači da smo napravili grešku u procesu, greška u procesu zahteva popravke procesa.

## 184 Navesti najvažnije tehnike za prevenciju nastajanja bagova.

**Unutrašnje tehnike i alati** - sve ono što se ugrađuje u programski kod samo radi pomoći u prevenciji i otklanjanju grešaka

**Spoljašnje tehnike i alati** - različiti alati koji se upotrebljavaju tokom razvijanja ili debagovanja programa

Unutrašnje tehnike

- pravljenje pretpostavki (assert)
- komentarisanje značajnih odluka i mesta u kodu
- testiranje jedinica koda

Spoljašnje tehnike i alati

- Debager
- Alati za praćenje verzija programskog koda
- Alati za podršku i praćenje komunikacije
- Alati za automatizovanje pravljenja dokumentacije

# 185 Objasniti *pisanje pretpostavki* kao tehniku za prevenciju nastajanja bagova.

Pretpostavke (asserts) su delovi koda koji tokom izvršavanja programa proveravaju da li su na datom mestu i u datom trenutku ispunjene neke pretpostavke koje moraju važiti. Obično prekidaju ili privremeno obustavljaju izvršavanje koda sa odgovarajućim obaveštenjem.

assert(...)

- većina biblioteka ima različite oblike makroa *assert* koji proverava da li je dati uslov zadovoljen
- ako nije, prekida izvršavanje programa
- ako nije verzija za debagovanje, onda se provera ne uključuje u kod

Primer:

```
#include <assert>
...
int fact( int n ){
assert( n<100 );
int r = 1;
while( n > 1 )
r *= (n--);
return r;
}
```

- navoditi sasvim jednostavne uslove
- ako nije zadovoljen neki složen uslov, kako ćemo znati koji njegov deo nije zadovoljen
- složene uslove je bolje podeliti na više jednostavnijih uslova
- proveravati da li argumenti potprograma zadovoljavaju neophodne uslove
- ako ne zadovoljava, znači da imamo grešku u implementaciji funkcije
- u složenim potprogramima proveravati međurezultate

Biblioteka QT ima sledeće elemente:

- *Q_ASSERT*( bool test )
- *Q_ASSERT_X*( bool test, const char* where, const char* what )
- *Q_CHECK_PTR*( ptr )

# 186 Objasniti tehniku *ostavljanja tragova pri izvršavanju* kao prevenciju nastajanja bagova.

Osnovna pravila su već pominjala ovu tehniku. U zavisnosti od načina implementacije dnevnika, pravi se marko ili šablon (ili više makroa/šablona) koji ispisuju tekuće stanje u dnevnik.

Primer:

```
void TRACE(const char* s){
cerr << gettime() << s << ... << endl;
}
```

## 187 Objasniti *komentarisanje koda* kao tehniku za prevenciju nastanaka bagova.

Komentarisanje je vid dokumentovanja koraka koji su doveli do toga da program izgleda kako izgleda. Komanetari moraju da opisuju:

- namenu nekog dela koda
- motivaciju da rešenje bude takvo kakvo je (posebno važno u slučaju naknadnog menjanja)
- načine povezivanja sa drugim delovima koda (posebno način upotrebe potprograma)

Komentari ne smeju da opisuju:

- ono što je očigledno
- ono što ne pripada konkretnom nivou apstrakcije

## 188 Objasniti *testiranje jedinica koda* kao tehniku za prevenciju nastanaka bagova.

*Obrađivano u pitanjima 129,130...*

## 189 Navesti osnovne tehnike upotrebe debagera.

- Izvršavanje korak po korak
- Postavljanje tačaka prekida
- Praćenje vrednosti promenljivih
- Praćenje lokalnih promenljivih
- Praćenje stanja steka
- Praćenje na nivou instrukcija i stanja procesora

## 190 Objasniti tehniku upotrebe debagera *Izvršavanje korak po korak.*

U izvršavanju korak po korak, korak može biti jedna linija izvornog koda ili jedna mašinska instrukcija. Ukoliko naiđemo na tačku prekida pri izvršavanju koraka, izvršavanje programa će stati na tački prekida.

## 191 Objasniti tehniku upotrebe debagera *Postavljanje tačaka prekida.*

Tačka prekida je lokacija u izvršnom kodu u kojoj operativni sistem staje sa izvršavanjem i podiže debager. To nam omogućava da analiziramo situaciju i prosledimo komande debageru.

## 192 Objasniti tehniku upotrebe debagera *Praćenje vrednosti promenljivih.*

Kucanjem *info variables* u GDB-u možemo da pratimo imena i vrednosti svih globalnih i statičkih promenljivih.

## 193 Objasniti tehniku upotrebe debagera *Praćenje lokalnih promenljivih.*

Kucanjem *info locals* u GDB-u možemo da pratimo imena i vrednosti svih lokalnih promenljivih u stek okviru u kom se nalazimo, uključujući i statičke promenljive te funkcije.

## 194 Objasniti tehniku upotrebe debagera *Praćenje stanja steka.*

Pri selektovanju stek okvira možemo pratiti imena i vrednosti promenljivih koje su dostupne na tom steku. Kucanjem *info args* možemo da pratimo imena i vrednosti argumenata tog stek okvira.

## 195 Objasniti tehniku upotrebe debagera *Praćenje na nivou instrukcija i stanja procesora.*

## 196 Šta je dizajn softvera? Šta je arhitektura softvera? Objasniti sličnosti i razlike.

**Dizajn softvera** je struktura softverskog sistema. Ali, to je i disciplina razvoja softvera koja se bavi stukturom softverskog sistema i dokumentacija koja opisuje strukturu softverskog sistema. Dokumentacija je zapravo prikaz dizajna. Koristi se za strukturu na relativno niskom nivou. Bavi se detaljima softvera.

**Arhitektura sistema** je struktura softverskog sistema, posmatrana na relativno visokom nivou. Vidimo samo najkrupnije elemente i njihove međusobne odnose. Ali, to je i disciplina razvoja softvera koja se bavi pravljenjem softverskog sistema i dokumentacija koja opisuje arhitekturu softverskog sistema.

Svaka arhitektura je dizajn, ali nije svaki dizajn arhitektura. Pri pravljenju arhitekture softverskog sistema preduzimaju se specifične aktivnosti i uzimaju u obzir specifični aspekti problema:

- veliki značaj širine posmatranja
- manji značaj pojedinosti

## 197 Šta čini dizajn softvera? Navesti osnovne pojmove dizajna

Strukturu softverskog sistema čine svi njegovi elementi. Zato i dizajn softvera čine svi njegovi elementi:

- komponente i njihov intrfejs
- implementacija
- stukture podataka
- infrastruktura
- pomoćni alati

Dva osnovna pojma su:

- apstrakcija
- dekompozicija

# 198 Šta je apstrakcija? Objasniti ulogu apstrakcije u dizajnu softvera.

Apstrakcija je razmatranje opštih karakteristika problema i rešenja uz zanemarivanje pojedinosti. Podrazumeva rešavanje problema na konceptualnom nivou. Ima sličnosti sa mehanizmom indukcije - posmatranjem sličnih slučajeva težimo da napravimo opšti model. Kaže se da je apstrakcija *uopšteno rešenje*. Familija sličnih problema se uopštava u jedinstveno rešenje koje će biti istovremeno rešenje za sve. Približava problem arhitekturi. Daje grubu sliku rešenja. Arhitektura počiva na apstrakciji.

# 199 Šta je dekompozicija? Objasniti ulogu dekompozicije u dizajnu softvera.

Dekompozicija je postepeno preciziranje sistema kroz prepoznavanje manjih celina (komponenti) koje ga čine. Rekurzivnom primenom dekompozicije se dolazi do fizičkog dizajna. Predstavlja vezu između različitih nivoa apstrakcije. Dekompozicija apstraktnog sistema vodi njegovoj konkretnoj strukturi. Ima sličnosti sa mehanizmom dedukcije - posmatranjem i razlaganjem opšteg slučaja težimo da napravimo konkretniji model. Kaže se i da je dekompozicija *razloženo rešenje*. Imamo veliki problem i delimo ga na manje celine radi lakše implementacije sistema.

# 200 Navesti i ukratko objasniti 6 osnovnih pojmova dizajna softvera.

**Modularnost**

- složene celine se dele na manje module
- na višem nivou apstrakcije se posmatra celina, a na nižem dekomponovani moduli koji čine celinu.

**Enkapsulacija**

- razdvajanje posmatranja *ponašanja* komponenti i njihove *implementacije*
- na višem nivou apstrakcije se posmatra ponašanje, a na nižem implementacija

**Razdvajanje nadležnosti**

- jedan od kriterijuma pri dekompoziciji
- svaka komponenta može da ima jasno prepoznatu i zaokruženu odgovornost
- MVC arhitektura

**Interfejsi**

- predstavljaju tačke vezivanja komponenti
- mogu se posmatrati kao deklaracije ponašanja komponenti

# 201 Navesti i ukratko objasniti bar 6 ključnih principa dizajna softvera.

**Princip jedinstvene odgovornosti**

- jedan modul bi trebalo da ima jednu namenu

- komponenta bi trebalo da ima samo jedan razlog za menjanje
- jedna komponenta - jedna odgovornost
- što je više odgovornosti veća je šansa da će morati da se menja
- nije dobto grupisati stvari koje nisu vezane

**Razdvajanje odgovornosti**

- različite komponente bi trebalo da imaju različite odgovornosti
- nijedna odgovonost ne bi trebalo da bude podeljenja na više komponenti

**Princip minimalnog znanja**

- jedna komponenta ne mora da zna kako druga radi
- dovoljno je da zna kako da je koristi

**Princip inverzije zavisnosti**

- apstrakcija ne sme da zavisi od pojedinosti
- pojedinosti moraju da zavisi od apstrakcije

**Princip acikličnih zavisnosti**

- izbegavati cikličnu zavisnost komponenti

**Princip stabilne zavisnosti**

- smer zavisnosti bi trebalo da se poklapa sa smerom porasta stabilnosti

**Princip stabilne apstrakcije**

- paket (komponenta) bi trebalo da bude apstraktan koliko je stabilan
- biraju se komponente koje su najmanje podložne promenama da budu najapstraktnije

**Izbegavanje ponavljanja**

- nijedna funkcionalnost se ne bi smela implementirati više puta

**Izbegavanje suvišnog posla**

- predvideti samo ono što je danas potrebno
- ono što će možda biti potrebno - nije potrebno

Dobar je onaj projekat koji omogućava da promene budu što lokalnije. Promena će sigurno biti. Ako ih nema znači da se ne softver ne koristi.

## 202 Objasniti odnos pisanja koda i dizajniranja.

Praktično svaki aspekt razvoja softvera menja dizaj softvera ili utiče na dizajn softvera. Ako pretpostavimo da izvorni kod predstavljanje dizajn softvera

- izgradnja (prevođenje) koda je proces implementacije
- cena izgranje je izuzetno niska (sve niža i niža sa bržim i jeftinijim računarima)
- cena dizajniranja je relativno niska (u odnosu na druge vrste projekta i dobijenu složenost)
- dizajn softvera lako i brzo postaje veoma veliki i složen
- programiranje je mnogo više od kodiranja, obuhvata i projektovanje
- razvoj softvera je i zanatksa i inženjerska disciplina

## 203 Objasniti ulogu i mesto dizajniranja u razvoju softvera u savremenoj praksi.

- Što je dizajn nižeg nivoa, to se ređe pravi pre samog programskog koda
  - dizajn softvera se sve ređe pravi i dokumentuje pre kodiranja
  - ako se pravi, onda samo umereno detaljno
- Što je nizajn višeg nivoa, to je neophodnije njegovo pažljivo planiranje i dokumentovanje pre kodiranja
  - arhitektura softvera se najčešće pravi i dokumentuje pre kodiranja
  - preskakanje arhitekture obično ima skupe posledice
- granica između dizajna i arhitekture je obično nivo komponente
- Planiraju se
  - komponente
  - njihovi interfejsi
  - njihovi međusobni odnosi
- To uobičajeno spada u arhitekturu
- Ne planira se detaljno implementacija komponenti
- To uobičajeno spada u dizajn
- Što je sistem složeniji, to se arhitektura pravi na više različitih nivoa apstrakcija

## 204 Navesti i ukratko objasniti obaveze softverskog arhitekte.

- Razumevanje svih aspekata razvoja
  - praćenje procesa
  - uzimanje u obzir svih praktičnih ograničenja
- Razumevanje sopstevne uloge
  - preuzimanje odgovornosti
  - ne zalaženje u (iz ugla arhitekture) nebitne detalje
- Komuniciranje sa ulagačima - arhitektura mora da zadovolja potrebe ulagača
- Izvođenje rešenja iz poslovnih potreba

  - prepoznavanje i razumevanja potreba
  - arhitektura je šlika"poslovnog okruženja
- Uticanje na zahteve
  - prepoznavanje kompromisa i predočavanje ulagačima
  - pomaganje ulagačima da donesu ispravne odluke
- Upravljanje rizicima i promenama - iterativno unapređivanje i prilagođavanje arhitekture
- Ponovljena upotreba - upotreba postojećih dobara štedi vreme i novac
- Dobro odmeravanje učešća
  - arhitektura mora da dodnese odluke u njegovom domenu
  - ne sme da izađe van svog domena
- Preciziranje arhitekture prema tehnološkim ograničenjima
  - arhitektura se uvek implementira u nekom tehnološkom kontekstu
- Uvažavanje opštih okvira
  - svaka konkretna arhitektura je samo deo šireg poslovnog konteksta
  - ona mora da se slaže sa ostalim elementima okruženja

## 205 Navesti i ukratko objasniti osnovne aspekte arhitekture i ključne uticaje na arhitekturu.

- Šta? - zahtevi koje softver mora da zadovolji
- Zašto? - ključne odluke na osnovu kojih se pravi arhitektura
- Kako? - dizajn i implementacija rešenja
- Međusobno suprotstavljene tri sile:
  - želje - sve ono što ulagači žele
  - izvodljivost - sve ono što tehnologija može da pruži
  - održivost
    * sve ono što u datim uslovima može da se postigne
    * vreme, novac i drugi resursi
- Nešto manje važna jaka četvrta sila:
  - estetika - svi oni neformalni (iracionalni?) aspekti rešenja koji ga čine kvalitetnijim

## 206 Šta su kohezija i spregnutost u kontekstu razvoja softvera?

**Kohezija** je stepen međusobne povezanosti elemenata jednog modula.

**Spregnutost** je stepen međusobne povezanosti elemenata dvaju različitih modula.

Kohezija i spregnutost (eng. coupling) su dve zajedničke mere koje opisuju složenost softverskog sistema. Dobro dizajniran sistem se odlikuje:

- visokom kohezijom
- niskom pregnošću komponenti

## 207 Navesti vrste kohezije u kontekstu razvoja softvera.

- funkcionalna
- sekvencijalna
- komunikaciona
- proceduralna
- vremenska
- logička
- koincidentna

## 208 Objasniti funkcionalnu koheziju u kontekstu razvoja softvera.

Funkcionalna kohezija predstavlja povezanost delova komponente na osnovu međusobne funkcionalne zavisnosti, a u cilju ostvarivanja funkcije za koju je komponenta odgovorna

- svi delovi komponente su prikupljeni sa jednim istim osnovnim ciljem
- svaki deo je potreban - nijedan deo nije višak

Funkcionalna kohezija je ideal kome se teži pri projektovanju komponenti.

## 209 Objasniti sekvencijalnu koheziju u kontekstu razvoja softvera.

Kohezija je sekvencijalna ako su eementi komponente projektovani tako da izalz jedne predstavlja ulaz druge

- ne postoji potupna funkcionalna zavisnost
- potencijalno delovi sekvence obavljaju različite poslove
- potencijalno otvoren problem odgovornosti u odnosu na celovit posao

## 210 Objasniti komunikacionu koheziju u kontekstu razvoja softvera.

Kohezija je komunikaciona ako su elementi komponente prikupljeni zato što kotiste iste podatke

- ne postoji funkcionalna zavisnost
- obično delovi obavljaju različite poslove
- obično nisu dobro razdvojene odgovornosti
  - jedna komponenta ima više odgovornosti
  - jedna odgovornost podeljena na više komponeneti

Ne znači da komponente međusobno komuniciraju već da koriste iste podatke.

## 211 Objasniti proceduralnu koheziju u kontekstu razvoja softvera.

Kohezija je proceduralna ako su elementi komponente prikupljeni zato što se koriste pri obavljanju nekog celovitog posla

- npr. otvaranje datoteke, proveravanje ispravnosti, ...
- verovatno ne postoji funkcionalna zavisnost
- često delovi obavljaju potpuno raznorodne poslove, koji se, doduše, često obavljaju jedan za drugim
- obično nisu dobro razdvojene odgovornosti
  - jedna komponenta ima više odgovornosti
  - jedna odgovornost podeljena na više komponeneti

## 212 Objasniti vremensku koheziju u kontekstu razvoja softvera.

Kohezija je vremenska ako su elementi komponenete prikupljeni zajedno zato što se koriste u "istom"periodu vremena

- npr. različiti elementi inicijalizacije sistema
- verovatno ne postoji funkcionalna zavisnost
- često delovi obavljaju raznorodne poslove
- obično nisu dobro razdvojene odgovornosti
  - jedna komponenta ima više odgovornosti
  - jedna odgovornost podeljena na više komponeneti

## 213 Objasniti logičku koheziju u kontekstu razvoja softvera.

Kohezija je logička ako su elementi komponente prikupljeni zato što imaju logički sličnu (ili čak istu) ulogu u sistemu

- npr. različiti načini učitavanja podataka sa ulaza, ...
- najčešće ne postoji funkcionalna zavisnost
- delov obično nobavljaju potpuno raznorodne poslove koji mogu predstavljati alternativu jedni drugima
- obično nisu dobro razdvojene odgovornosti
  - jedna komponenta ima više odgovornosti
  - jedna odgovornost podeljena na više komponeneti

## 214 Objasniti koincidentnu koheziju u kontekstu razvoja softvera.

Kohezija je koncidentna ako su elementi komponente međusobno nespregnuti ili su veoma slabo spregnuti

- npr. komponenta obuhvata sve jedinice koda pisanog nekim alatom, ...
- ne postoji nikakava ili postoji samo sasvim niska funkcionalna zavisnost
- deloovi obavljaju potpuno raznorodne poslove

Koincidentna kohezija je najniži nivo kohezije.

## 215 Navesti osnovne karakteristike spregnutosti u konekstu razvoja softvera.

- vrsta sprege
- nivo sprege
- širina sprege
- način ostvarivanja sprege
- intenzitet spregnutosti

## 216 Zašto je spregnutost komponenti softvera potencijalno problematična?

Spregnutost je neminovna. Da bi komponente sistema sarađivale one moraju da budu spregnute. Ako komponenete komuniciraju, na bilo koji način, onda među njima postoji sprega. Ako su komponente potpuno nezavisne , onda one ne mogu da sarađuju.

Spregnutost nije problem sama po sebi, problem mogu predstavljati neke karakteristike spregnutosti.

## 217 Navesti i ukrakto objasniti vrste spregnutosti u konekstu razvoja softvera.

- sprega logike
- sprega tipova
- sprega specifikacije

## 218 Objasniti spregu logike u kontekstu razvoja softvera.

Ako komponenete dele informacije ili pretpostavke jedna o drugoj, onda je u pitanju sprega logike. Primeri:

- komponente obavljaju različite delove jednog istog posla
- komponenta A pretpostavlja da implementacija metoda B::M počiva na nekom konkretnom algoritmu
  - npr. A i B implementiraju komplementarne algoritme (kodiranje/dekodiranje, pisanje/čitanje, ...)
- komponenete dele pretpostavke o nekim konkrentim podacima
  - npr. A zapisuje datoteku pod nekim imenom, a B čita

Sprega logike je visoko problematična i nepoželjna.

## 219 Objasniti spregu tipova u kontekstu razvoja softvera.

Sprega tipova označava da jedna komponenta koristi neki tip definisan u okviru druge komponente. Može biti:

- određena - ako komponenta A pravi instance tipa B
- neodređena - ako komponenta A ne pravi instance tipa B tj. može u prakci dobijati na upotrebu instance podtipove

Primeri:

- komponenta A koristi klasu implementiranu u komponenti B

Dobro implementirana sprega tipova ne predstavlja problem, posebno ako je neodređena.

## 220 Objasniti spregu specifikacije u kontekstu razvoja softvera.

Sprega specifikacije je još apstraktnija nego sprega tipova. Pretpostavlja se da nije poznat tip koji se koristi već samo neke pretpostavke o njegovom interfejsu. Primeri:

- upotreba parametarskog ili polimorfizma
- upotreba referenci na metode objekata
- apstraktne bazne klase hijerarhija
- implementacija interfejsa

Dobro implementirana sprega specifikacije ne predstavlja problem.

## 221 Navesti i ukratko objasniti nivoe spregnutosti u konekstu razvoja softvera.

Spregnutost može biti (od najjače prema najslabijoj):

- po sadržaju
- preko zajedničkih delova
- spoljašnja spregnutost
- preko kontrole
- preko markera
- preko podataka
- preko poruka

### Spregnutost preko poruka

Spregnutost je preko poruka ako jedna komponenta koristi interfejs druge komponenete putem koga mu ne prenosi nikakave podatke.

Pojednostavljen primer:

```
...A::metod1(...) {
   ...
   transakcija->izvrsni();
   ...
}
```

Problem je zavisnost na nivou interfejsa.
Ovo je najniži nivo spregnutosti i u načelu je sasvim prihvatljiv.

Ostali nivoi opisani su u narednim pitanjima.

## 222 Objasniti spregnutost po sadržaju u kontekstu razvoja softvera.

Spregnutost po sadržaju predstavlja otvoreno i neposredno pristupanje i/ili menjanje sadržaja jedne komponente od strane druge komponente.

Pojednostavljen primer:

```
...A::metod1(...) {
   ...
   objB->podatak
   ...
}
```

Problemi:

- narušena enkapsulacija
- dovedene su u pitanje odgovornosti komponenti
- veoma je teško održavati komponentu od čijih internih aspekata implementacije zavisi drugia komponenta

Spregnutost po sadržaju je najjača i najnepoželjnija vrsta spregnutosti.

## 223 Objasniti spregnutost preko zajedničkih delova u kontekstu razvoja softvera.

Spregnutost preko zajedničkih delova je na delu ako dve ili više komponenti neposredno pristupaju nekim podacima.

Pojednostavljen primer:

```
struct ZajednickiDelovi {...};
...
A::metod1(...){
   ...
   zajednickiDelovi->podatak
   ...
}
...
B::metod2(...){
   ...
   zajednickiDelovi->podatak
   ...
}
```

Problemi:

- razlikuje se od spregnutosti po sadržaju samo po tome što su deljeni podaci odvojeni od ponašanja i stavljeni na upotrebu drugim komponentama
- enkapsulacija samo načelno nije narušena, a zapravo nije ni uspostavljena
- veoma je teško održavati ovako spregnute komponente

Spregnutost preko zajedničkih delova je skoro jednako nepoželjna kao spregnutost po sadržaju.

# 224 Objasniti spoljašnju spregnutost u kontekstu razvoja softvera.

Spoljašnja spregnutost je na delu ako više komponenti upotrebljava isti od spolja nametnut koncept( interfejs, format podatala, komunikaconi protokol, ...)

Pojednostavljen primer:

```
...
A::metod1 {
...
external::oper1(...);
...
external::oper2(...);
};

...
B::metod2(...){
...
external::oper3(...);
...
external::oper4(...);
}
```

Problemi:

- ponavljanje koda
- ako se dati koncept izmeni, moraju se menjati sve komponente koje ga upotrebljavaju
- podeljene odgovornosti
- poželjno je razmotriti učauravanje elemenata spoljašnjeg koncepta u nivou komponentu (ili komponente), koju bi tazim koristile ovako spregnute komponente

Spoljašnja spregnutost je često sasvim prihvatljiva ako su spoljasnji koncepti relativno stabilni i pouzdani.

# 225 Objasniti spregnutost preko kontrole u kontekstu razvoja softvera.

Spregnutost je preko kontrole ako jedna komponenta ultimativno upravlja radom druge komponente

- upravljana komponenta nije u stanju da funkcioniše bez spoljašnje kontrole ili
- upravljana komponenta očekuje sva ili neka uputstva za rad po kojima zatim samostalno postupa

Pojednostavljen primer:

```
class Kontrolisana {
...
oper1(...);
...
oper2(...);
};
...
Kontroler::metod(...){
...
kontrolisana->oper1(...)
...
kontrolisana->oper2(...)
}
```

Problemi:

- potencijalno nejasna podela odgovornosti
- ako je kontrolor odgovora za viši nivo operacija, onda to mora biti njegova jedina odgovornost

Spregnutost preko kontrole je relativno jaka, ali može biti veoma korisna u nekim slučajevima

- npr. kontrolisana komponenta implementira elementarne postupke, a različite stategije obezbeđuju složene načine upravljanja radom komponente

# 226 Objasniti spregnutost preko markera u kontekstu razvoja softvera.

Spregnutost je preko markera ako više komponenti međusobno razmenjuje neki složenu stukturu podataka (marker) koju upotrebljavaju na različite načine

- ako neposredno koriste iste podatke markera, onda je u pitanju spregnutost preko zajedničkih delova

Pojadnestovljen primer:

```
...
parser->fillRequest( request );
environment->prepareRequest( request );
processor->performTransaction( request );
response = reporter->prepareReport( request );
```

Problemi:

- potencijalno otvoreno pitanje odgovornosti markera
  - šta je njegova odgovornost?
  - zašto ne sve ili neke operacije?
  - ...
- marker praktično nije dovoljno enkapsuliran
- postoji opasnost od sukoba nadležnosti

Slično kao u slučaju spregnosti preko kontrole, u nekim slučajevima je ovakvo rešenje sasvim prihvatljivo

- ako marker nema složeno ponašanje
- ako marker nije spregnut na drugi način osim ovim putem
- ako postupak obrade u kome marker učestvuje nije stabilan ili realno zavisi od više različitih komponenti

# 227 Objasniti spregnutost preko podataka u kontekstu razvoja softvera.

Spregnutost podataka ako jedna komponenta kosirti interfejs druge komponente putem koga mu prenosi pojedinačne podatke.

Pojednostavljen primer:

```
...
simulacija::promenaSmera(...){
   ...
   automobil->skreniLevo( 5 );
   ...
}
```

Problem je zavisnost na nivou interfejsa.

Ovaj vid spregnutosti e među najnižim i u načelu je sasvim prihvatljiv.

## 228 Objasniti pojam širina sprege u kontekstu razvoja softvera.

Širina sprege dveju komponenti odgovara broju elemenata komponente S (objekata, podataka, metoda, poruka, ...) koje upotrebljava komponenta B. Poželjno je da sprega bude što uža:

- uska sprega znači da su jasni podela odgovornosti među komponentama i razlog njihove spregnutosti
- ako je sprega široka, to može da znači da
  - odgovornost između ove dve komponente nisu nedovoljno diferencirane - možda je moguće suziti spregu implementiranjem nekih složenijih celina koda u okviru komponente A
  - odgovornosti komponente A su preširoke - možda je potrebno podeliti komponentu A na više manjih

## 229 Objasniti pojam smer sprege u kontekstu razvoja softvera.

Sprega može biti:

- jednosmerna - komponenta A upotrebljava elemente B, ali ne i obratno
- dvosmerna - komponenta A upotrebljava elemente komponente B, ali i komponenta B upotrbljava elemente komponente A
- cirkularna - ne postoji neposredno dvosmerna sprega dveju komponenti , ali pstoji tranzitivna cirkularna spregnutost više komponenti

Poželjno je da sprega bude jednosmerna. Dvosmerna i cirkularna sprega značajno uvećavaju složenost sistema

- usložnjava se podela odgovornost među komponentama
- dovodi se u pitanje enkapsulacija

## 230 Objasniti pojam statička spregnutost u kontekstu razvoja softvera.

- Ostvaruje se pre izvršavanja programa
- Eksplicitno je iskazna u programskom kodu
- Primeri:
  - nalseđivanje klasa
  - podatak klase A je objekat klase B
  - argument metoda klase A je objekat klase B
  - u metodima klase A se pravi ili upotrebljava objekat klase B

## 231 Objasniti pojam dinamička spregnutost u kontekstu razvoja softvera.

- Ostvaruje se u toku izvršavanja programa
- Nije eksplicitno iskazana kodom programa
- Može da zavisi od konfiguracije, ulaznih parametara, ...
- Primer:
  - ako klasa A sadrži referencu na klasu B (statički), konkretan objekat (dinamički konfigurisan) može pripadati bilo kojoj klasi koja nasleđuje B

## 232 Objasniti odnos statičke i dinamičke spregnutosti u kontekstu razvoja softvera.

- Statička sprega je jača i nepoželjna od dinamičke
  - deklaracija (interfejsa ...) i implementacija neke statički spregnute komponente imaju neposrednog uticaja na implementaciju ostalih spregnutih komponenti
  - promena dinamički spregnute komponente je lokalizovana - ako odgovara nasleđenom interfejsu, onda je sve u redu, a ako ne odgovara, potrebno je samo popraviti tu komponentu

## 233 Objasniti pojam intenzitet spregnutosti u kontekstu razvoja softvera.

Intenzitet spregnutosti je složena karakteristika

- logička sprega ima veći intenzitet od sprege tipova i specifikacija
- viši nivo spregnutosti ima veći intenzitet
- šira sprega ima ima veći intenzitet
- cirkularna sprega ima veći intenzitet od dvosmerne, a dovsmerna od jednosmerne
- statička sprega ima veći intenzitet od dinamičke

## 234 Na koji način se može pristupiti merenju i računanju intenziteta spregnutosti?

Mera spregnutosti neke komponetnte sa sistemom se može izraziti na više načina

- broj komponenti koje se referišu iz te komponente
- broj elemenata drugih komponenti koje se referišu iz te komponente
- broj komponenti iz kojih se referiše ta komponenta
- broj elemenata te komponente koje se referišu iz drugih komponenti
- karakteristike posmatranih sprega

Intenzitet spregnutosti se može izraziti numerički na više načina. Na primer:

- Intenzitet spregnutosti dveju komponenti:

  coefSprege(A,B) = coefVrste(A,B) + coefNivoa(A,B) + ...

- Spregnutost jedne komponente sa sistemom:

  coefSprege(A) = sumB(coefSprege(A,B))

- Ukupna spregnutost sistema:

  coefSpregeSistema = sumA(coefSprege(A))

## 235 Navesti i objasniti dva osnovna pravila u vezi spregnutosti komponenti u kontekstu razvoja softvera.

1. Što je komponenta složenija, to je važnije da bude što manje spregnuta.
2. Spregu je poželjno uvoditi na što jednostavnijim komponentama.

## 236 Navesti nekoliko uobičajenih načina spregnutosti komponenti.

Neki oblici spregnutosti su relativno česti. Mogu se prepoznati u uzorcima za projektovanje. Primeri:

- arhitektura klijent-server
- hijerarhija pripadnosti
- cirkularna spregnutost
- sprega putem intefejsa
- sprega putem parametara metoda

## 237 Objasniti karakteristike spregnutosti u slučaju arhitekture klijent-server.

Jedna od najčešćih i tipičnih oblika sprege:

- komponenta server pruža usluge
- komponenta klijent koristi usluge

Oba sprega je:

- jednosmerna
- poželjno uska (server pruža samo jedan tip usluga)
- nivo sprege je obično relativno nizak - preko poruka ili preko parametara
- sprega tipova ili specifikacija
- nizak intenzitet sprege preporučuje ovu vrstu sprege kao idealnu

## 238 Objasniti karakteristike spregnutosti u slučaju hijerarhije pripadnosti.

Relativno čest oblik sprege:

- komponenta rofitelj sadrži više dece, koja zapravo obavljaju funkciju ili delove funkcije
- komponente deca obavljaju svaka svoj deo odgovornosti
- roditelj je odgovoran za koordinaciju

Primer: prozor - dekorateri.

Ova sprega je:

- obično jednosmerna
- relativno uska
- nivo sprege je obično relativno nizak - preko poruka ili preko parametara
- sprega tipova ili specifikacija
- nivo se često može dalje spustiti, npr. primenom uzorka graditelj

## 239 Objasniti karakteristike spregnutosti u slučaju cirkularne spregnutosti.

Relativno čest oblik sprege:

- komponenta roditelj sadrži više dece
- svako dete zna ko mu je roditelj i pristupa mu radi određinih poslova

Primer: prozor-kontrole

Ova sprega je:

- dvosmerna
- potencijalno široka
- nivo sprega je obično relativno nizak - preko poruka ili preko parametara
- sprege tipova ili specifikacija
- nivo se često može dalje spustiti, npr. primenom zajedničkih nadtipova i/ili uzorka graditelj

## 240 Objasniti karakteristike spregnutosti u slučaju sprege putem interfejsa.

Tipičan oblik sprege u OO razvoju:

- komponenta B deklariše (i implementira) interfejs
- komponenta A upotrebljava komponentu B posredstvom interfejsa

Ova sprega je:

- potencijalno jednosmerna
- širina zavisiod kvaliteta dizajna interfejsa
- nivo sprege je obično relativno nizak - preko poruka ili preko parametara
- sprega tipova ili specifikacije

## 241 Objasniti karakteristike spregnutosti u slučaju sprege putem parametara metoda.

Čest oblik sprege u OO razvoju:

- komponenta A deklariše metod koji kao parametar prima komponentu B
- komponenta BB, kao specijalizacija komponetnte B, se predaje na upotrebu komponenti A kao parametar deklarisanoj metodi

Ova sprega je:

- potencijalno jednosmerna
- širina zavisi od kvaliteta dizajna interfejsa
- nivo sprege je obično preko metoda
- obično sprega tipova ili specifikacija

# 242 Objasniti odnos koncepta klase i pojma kohezije u konekstu OO razvoja softvera.

U kontekstu OO programiranja:

- Kohezija klase predstavlja stepen međupovezanosti elemenata jedne klase
  - ako je nizak nivo kohezije u klasi, to ukazuje da klasa nije dobro oblikovana
    - ako postoji kohezija između nekih elemenata klase, ali ne svih, ili
    - ako se prepoznaje više različitih skupova elemenata klase unutar kojih postoji snažna kohezija
  - moguće je da postoji problem prepoznavanja odgovornosti klase
  - klasu je možda potrebno podeliti na vise klasa

# 243 Objasniti odnos koncepta klase i pojma spregnutosti u konekstu OO razvoja softvera.

U kontekstu OO programiranja:

- Spregnutost predstavlja stepen međupovezanosti različitih klasa
  - ako je visok nivo spregnutosti različitih klasa, to ukazuje da odgovornosti nisu dobro podeljene među klasama
  - možda je od više klasa potrebno napraviti jednu
  - možda je potrebno drugačije podeliti odgovornosti među klasama

## Spregnutrost i refaktorisanje

Više tehnika refaktorisanja imaju za cilj upravo smanjivanje međusobne spregnutosti komponenti:

- smanjivanje nivoa
- statička u dinamičku
- dvosmerna ili sirkularna u jednosmernu
- ...

## Spregnutrost i arhitektura

Jedan od najvažnijih zadataka pri projektovanju softvera je ostvarivanje potrebne funkcionalnosti uz što višu koheziju i što nižu spregnutost komponenti - svodi se na dobro prepoznavanje komponenti i njihovih odgovornosti

Većina savremenih arhitektura i metodologija razvoja softvera uzima u obzir upravljanje složenostima problema i rešenja.

Primeri:

- višeslojne arhitekture softvera
- arhitekture orijentisane prema uslugama (eng. service oriented architecture)
- softver upravljan događajima

# 244 Šta su arhitekture zasnovane na događajima?

Klasičan pristup razvoju softvera podrazumeva implementiranje algoritama koji obavljaju određeni posao

- primenom metoda od vrha naniže određene celine problema izdvajaju u potprograme
- i dalje aspekti problema ostaju celina

Primer: Program koji u petlji proverava da li je korisnik pritisnuo neki taster aa zatim pristupa u skladu sa akcijom korisnika.

Kod arhitektura zasnovanih na događajima, eiksplicitna komunikacija među objektima se zamenjuje mehanizmom emitovanja događaja i distribuiranja zainteresovanim objektima.

Primer:

- Program obaveđtava OS da je zainteresovan da reaguje na pritisak tastera od strane korisnika
- OS prepoznaje događaj i šalje programu poruku
- Program (odgovarajući objekat) reaguje na poruku

# 245 Objasniti motivaciju za upotrebu arhitektura zasnovanih na događajima.

Arhitekture zasnovane na događajima spuštaju nivo međusobne zavisnosti objekata i tako smanjuju ukupnu složenost sistema

- uobičajen je nivo spregnutosti putem parametara ili čak putem poruka
- uobičajena je dinamička sprega
- konfigurisanje odnosa među objektima pri pravljenju objekata
- smanjuje se ukupan intenzitet spregnutosti

Cena smanjivanja ukupne složenosti primenom arhitektura zasnovanih na događajima može biti uvećavanje lokalne složenosti

- komponente koje sarađuju u sistemu zasnovanom na događajima su jednostavnije iz ugla složenosti koda
- njihove operacije su definisane apstraktnije pa mogu biti teže za razumevanje, ako se posmatraju lokalno, bez razmatranja čitavog sistema

# 246 Objasniti osnovne pojmove i koncepte arhitekture zasnovane na događajima.

**Događaj** je prepoznatljiv uslov koji inicira obaveštenje.

**Obaveštenje** je događajem iniciran signal koji se šalje primaocu.

Posledice:
Ovakva arhitektura omogućava da se za mnoge složene probleme napravi rešenje koje obezbeđuje najniži mogući (poznati?) intenzitet sprege. Pored toga, postoje dodatni kvaliteti:

- ako neki signali prestanu da budu značajni, može da prestane njihovo praćenje, a da komponenta koja ih emituje pri tome ne mora da se menja
- ako je neke singnale potrebno obraditi višestruko, dovoljno je dodati nove slotove koji će ih prihvatiti, a da se komponente koje emituju signale, kao ni druge komponente koji već obrađuju te signale, ne moraju menjati

## 247 Navesti i objasniti slojeve toka događaja kod arhitektura zasnovanih na događajima.

**Generator događaja** - objekat koji prepoznaje da je napustio uslov koji predstavlja neki definisan događaj

**Mašina za obradu događaja** - mesto na kome se prepoznaje nastali događaj i odabire i pokreće odgovarajuća akcija (ili niz akcija)

**Kanal događaja** - mehanizam putem koga se događaj prosleđuje od generatora do mašine za obradu događaja

**Niz događaja upravljanih aktivnosti** - mesto ispoljavanja događaja

## 248 Objasniti osnovne koncepte primene arhitekture zasnovane na događajima u okviru biblioteke QT.

Biblioteku QT koristimo za upravljanje događajima. Pretpostavke:

- klasa koja generiše ili prihvata događaje mora da nasledi klasu QObject
- na samom početku deklaracije klase navodi se makro Q_OBJECT

## 249 Objasniti koncept signala u kontekstu primene arhitekture zasnovane na događajima u okviru biblioteke QT.

Metodi koji proizvode događaje nazivaju se **signali**

- deklarišu se u okviru sekcije *signals*
  - kao metodi tipa *void*
  - ne implementiraju se
- signali se emituju u obliku: *emit signal(...)*
- ključna reč **emit** je makro
  - za standardni C++ ima praznu definiciju - zato se može i izostavljati
  - preporučuje se da se ne navodi radi kompatibilnosti koda, dokumentovanja i lakšeg pronalaženja emitovanja

## 250 Objasniti koncept slotova u kontekstu primene arhitekture zasnovane na događajima u okviru biblioteke QT.

Metodi koji obrađuju događaje nazivaju se **slotovi**

- deklarišu se u okviru sekcije *private slots* ili *public slots*
- moraju biti istog tipa kao signali za čiju su obradu namenjeni

## 251 Objasniti povezivanje signala i slogova u slučaju primene arhitekture zasnovane na događajima u okviru biblioteke QT.

Povezivanje se ostvaruje pomoću statičkog metoda *OObject::connect*

- prvi argument je pokazivač na objekat koji emituje signal

- drugi argument je signal koji se emituje - navodi se pomoću makroa *SIGNAL*
- treći argument je pokazivač na objekat koji hvata signal
- četvrti argument je slot kojim se signal hvata - navodi se pomoću makroa *SLOT*
- slot i odgovarajući signal moraju biti istog tipa - nije važno ime

Zavisno od konfiguracije, nakon emitovanja signala se može sačekatida se završi njegova obrada ili odmah nastaviti sa radom. Podrazumevano ponašanje je da se sačeka sa obradom.

## 252 Objasniti odnos arhitektura zasnovanih na događajima i problema kohezije i spregnutosti.

Sprega preko signala i slotova je dinamička sprega:

- objekti koji se povezuju nisu spregnuti u kodu kojim su definisani
- sprega se uspostavlja u posebnom konfiguracionom delu programskog koda

Ipak, uspostavlja sprege (tj. povezivanje slotova i signala) je implementirano na statički način:

- Veze se uspostavljaju jednokratno
- Veze između signala i slotova se uspostavljaju bezuslovno

Uspostavljanje sprege putem povezivanja signala i slotova može da bude implementirano i na dinamički način:

- Veze se mogu predstavljati u zavisnosti od konteksta
- Tokom izvršavanja programa se veze mogu raskidati i ponovo uspostavljati
- metod *QObject::disconnect*

Naredni korak bi bilo dinamičko programiramsko pravljenje slotova i signala i zatim njihovo povezivanje. Biblioteka Qt nema neposrednu podršku za dinamičko pravljenje slotova i signala, ali postoji više primera dodatnih klasa koje to omogućavaju.

Moguće primene su u implementaciji skript jezika, dinamičkih koruženja za pravljenje ili menjanje korisničkih intefejsa i sl.

## 253 Šta je konkurento izvršavanje?

Dva posla $T1$ i $T2$ se izvršavaju konkurentno ako nije unapred poznata njihova međusobna vremenska lociranost

- posao $T1$ može da počne i završi pre početka $T2$
- posao $T1$ može da počne i završi posle započinjanja posla $T2$
- posao $T1$ može da počne pre započinjanja posla $T2$ i završi posle završavanja posla $T1$
- posao $T1$ može da počne posle započinjanja posla $T2$ i završi pre završavanja posla $T2$

## 254 Objasniti pojam paralelno izvršavanje.

Dva posla se izvršavaju paralelno ako postoji period vremena u kome su (doslovno) istovremeno aktivni

- konkurentni poslovi se mogu, ali ne moraju, izvršavati paralelno
- paralelni poslovi ali ne moraju biti konkurentni
- paralelni poslovi nisu konkurentni ako je unapred poznata njihova međusobna vremenska lociranost

# 255 Objasniti pojam distribuirano izvršavanje.

Dva posla se izvršavaju distribuirano ako rade u različitim adresnim prostorima

- ovo je opštija definicija, koja teži da izjednači (apstahuje) distribuiranost između više računara i distribuiranost između komponenti (procesora) istog računarskog sistema
- neke definicije si strože i podrazumevaju različite računarske sisteme, ali svi aspekti distribuiranja su uglavnom isti

## Motivacija

Paralelizacija, tj. podela posla na procese/niti, se preduzima iz dva osnovna razloga:

- razdvajanje odgovornosti
  - omogućava da potpuno razdvojimo kod koji radi jedan celovit deo posla
  - npt, jedna nit komunicira sa korisnikom i pravi zadatke, druga nit izvršava, a treća asinhrono prikazuje dokle se stiglo sa izvršavanjem poslova
  - često različiti procesi/niti imaju uloge klijenata ili servera u odnosu na druge niti/procese
  - poslovi se mogu dodatno razdvojiti tako da se izvršavaju na različitim računarima (distribuirano izvršavanje)
  - iako može da izgleda kao podizanje složenosti, razdvajanje odgovornosti na procese/niti je često prirodniji način rešavanja problema i zapravi ima nižu složenost nego veštačko spajanje u jedan postupak
- podizanje performansi
  - ako je potrebno da se neki posao uradi nad velikom količinom podataka, a na raspolaganju je više procesora, onda se efikasnost može višestruko uvećati podelom posla na više procesa/niti, od kojih svaki radi nad delom podataka
  - npr, ako je potrebno primeniti neku lokalnu transformaciju slike, onda se slika može podeliti na oblasti i obrada svake od oblasti poveriti drugom procesoru
  - takva podela često nije sasvim prirodna i podiže složenost arhitekture
  - u takvim slučajevima se često ubraja u optimizacije

# 256 Objasniti pojam proces.

Proces je instanca programa koja se izvršava na računarskom sistemu. Obično se razmatra samo u kontekstu računarski sistema koji imaju mogućnost izvršavanja višeprograma (ili instanci programa) u "isto"vreme. Ranije je definicija podrazumevala da se proces izvršava sekvencijalno, ali to više nije tako.

Proces obuhvata

1. kod programa koji se izvršava
2. tekuće stanje procesa

**Stanje procesa** obuhvata

- podatke o izvršavanju
- stanje izvršavanja (npr. spreman, radi, čeka, stoji, ...)
- brojač instrukcija (adresa naredne instrukcije)
- sačuvane vrednosti registara procesora

- informacije o upravljanju resursima
- informacije o memoriji (tablica stranica, podaci o alokaciji memorije, ...)
- deskriptori datoteka
- ostali resursi, poput U/I zahteva i sl.

**Promena konteksta** (eng. context switch) je promena stanja procesora koja je neophodna u slučaju kada procesor sa izvršavanja koda jednog procesa prelazi na izvršavanje koda drugog procesa. Stanje prethodno izvršavanog procesa zapisuje u memoriji. Stanje narednog procesa koji će se izvršavati se čita iz memorije.

Cena procesa:

- procesi su dvojako skupi
- promena konteksta uzima značajno vreme procesora
- promena konteksta se dešava veoma često - od nekoliko puta do nekoliko hiljada puta u sekundi
- veliki broj procesa može da oslabi performanse
- podaci o stanju procesa su obimni, zauzimaju mnogo memorije i pri svakoj promeni konteksta se promšuju u kešu

# 257 Objasniti pojam nit.

Nit izvršavanja je komponenta koja se izvršava sekvencijalno

- jedan proces može imati više niti, ako računarski sistem to podržava
- svaki proces započinje sa jednom glavnom niti

#### Stanje procesa i niti

- Podaci koji se vode na nivou procesa i zajednički su za sve niti tog procesa obuhvataju:
  - kod programa koji se izvršava
  - informacije o upravljanju resursima
  - informacije o memoriji (tablica stranica, podaci o alokaciji memorije, ...)
  - deskriptori datoteka
  - stanje izvršavanje procesa
  - ostali resursi, poput U/I zahteva i sl.
- Podaci o niti obuhvataju
  - stanje izvršavanje niti
  - brojač instrukcija
  - sačuvane vrednosti registara procesora

# 258 Zašto se uvodi koncept niti, ako već postoji koncept procesa?

Koncept niti izvršavanja koda se uvodi kako bi se spustila cena promena konteksta. Jednom procesu može da odgovara više niti izvršavanja. Ako se većina podataka vodi na nivou i deli među nitima jednoh procesa, štedi se na resursima i dobija na performansama.

# 259 Objasniti sličnosti i razlike niti i procesa.

- I procesi i niti se mogu izvršavati konkurentno i/ili paralelno
- unačajan broj problema pri konkurentnom izvršavanju se odnosi na isti način i na procese i na niti
- samo procesi se mogu izračunavati distribuirano
- sve niti jednog procesa rade istom adresnom prostoru
- procesi ne dele među sobom resurse (bar ne neposredno)
- komunikacija između procesa se odvija isključivo kroz posebne mehanizme za međuprocesnu komunikaciju
- nije potrebno eksplicitno staranje o eventualnom sukobljavanju oko resursa
- niti jednog procesa dele resurse tog procesa
- komunikacija između niti se najčešće odvija posredstvom deljenih resursa
- neophodno je eksplicitno staranje o eventualnom sukobljavanjem oko resursa

# 260 Objasniti kako se programiraju niti pomoću biblioteke QT. Klase, metodi, ...

Osnovnu podrške predstavlja klasa QThread. Osnovni metodi:

- bool isFinished()
- bool isRunning()
- bool wait(unsigned long time_msec = ULONG_MAX)

Slotovi (mogu se koristit kao metodi):

- void start()
- void quit()
- void terminate()

Singnali:

- void finished()
- void started()
- void terminated()

Nova klasa niti nasleđuje klasu QThread implementira se zaštićeni metod void run().

```
#include <QThread>
class MyThread : public QThread {
    Q_OBJECT
    public:
    MyThread(...);
    protected:
    void run();
};
```

## 261 Navesti i ukratko objasniti osnovne operacije sa nitima.

**Pravljenje**

- pravljenje niti na nivou OS-a obično podrazumeva i započinjanje njenog izvršavanja
- pravljenje objekta niti QThread ne podrazumeva i pravljenje niti na nivou OS-a
- nit će biti napravljena na nivou OS-a tek pozivanjem metoda start()

**Dovršavanje**

- kada nit svoje izvršavanje, oslobađa se većina resursa vezanih za nit
- ostaje samo konačan status niti (rezultat izvršavanja, slično funkciji main)

**Suspendovanje i nastavljanje**

- neki OS omogućavaju da se nit privremeno suspenduje i da se kasnije njeno izvršavanje nastavi (Windows, Solaris)
- kod OS koji ne podržavaju suspendovanje, odgovarajuće ponašanje se može ostvariti samo složenijim mehanizmima za sinhronizaciju
  - razlozi za višestruki, ali se sumiraju u potencijalno nekontrolisanom držanju zakljulanih resursa od strane suspendovane niti
  - POSIX

**Prekidanje**

- jedna nit može da prekine izvršavanje druge niti
- iako se ostvaruje naizgled jednostavno (pozivom jedne funkcije OS-a) ovo ej veoma osetljiva operacija
- prekidanje niti preti da uukgrozi konzistentnost deljenih podataka

**Čekanje**

- elementaran vid sinhronizovanja niti je kada jedna nit čeka da druga završi sa radom
- ostvarivanje čekanja je relativno jednostavno, ali se preporučuje implementacija složenijih mehanizama
- npr, nit koja završava može da postavlja neki deljeni signal, koji se zatim može lako proveravati od strane drugih niti

## 262 Objasniti detaljno operaciju pravljenja niti. Kako se implementira pomoću biblioteke QT?

Objekat klase QThread predstavlja pomoćno sredstvo za rukovanje nitima OS-a. Konstrukcija objekata ne obuhvata pravljenje niti. Nova nit se zaista pravi (na nivou OD-a) kada se pozove metod *void start()* objekta klase QThread.

## 263 Objasniti detaljno operaciju dovršavanja niti. Kako se implementira pomoću biblioteke QT?

Nit se dovršava kada se završi izvršavanje metoda *void run()* objekta klase QTHread. Dinamički napravljen objekat će biti automatski uklonjen ako mu se pri konstrukciji dodeli roditeljski objekat.

## 264 Objasniti detaljno operaciju suspendovanja i nastavljanja niti. Kako se implementira pomoću biblioteke QT?

Suspenodvanje niti nije u QT moguće.

## 265 Objasniti detaljno operaciju prekidanje niti. Kako se implementira pomoću biblioteke QT?

Nit se može prekinuti pozivanjem metoda *void terminate()*. Nit ne mora biti prekinuta tokom izvršavanja metoda

- kada će stvarno biti prekinuta nit biti prekinuta zavisi od odgovarajuće politike OS-a
- ako je potrebno da se nit pouzdano završi, nakon pozivanja ovog metoda se može sačekati da se niti završi

## 266 Objasniti detaljno operaciju čekanja niti. Kako se implementira pomoću biblioteke QT?

Nit se može čekati pozivanjem metoda *bool wait(unsigned long time = ULONG_MAX)*. Ako se argument ne navede (ili se navede ULONG_MAX), metod čeka dok se nit ne završi. Inače, metod čeka na završavanje niti najviše *time* milisekundi. Vraća *true* ako niti više nije aktivna (ili je završena ili nije ni započela) i *false* ako je nit još aktivna.

## 267 Koji su najvažniji problemi pri pisanju konkurentnih programa?

- deljenje resursa između jediniva izvršavanja
- komunikacija među jedinicama izvršavanja
- sinhronizacija jedinica izvršavanja

## 268 Šta je muteks? Kako se upotrebljava?

Muteksi (eng. mutex) su jedan od osnovnih načina bezbednog deljenja podataka u konkurentim okruženjima. Muteks ima sematiku katanca

- samo jedanput se može zaključati
- ako je muteks već zaključan, pokušaj zaključavanja će čekati na njegovo otključavanja da bi ga mogao zaključati

Implementiraju se (u osnovi) u okviru OS-a tako da operacije sa muteksima budu atomične.

## 269 Objasniti podršku za mutekse u okviru biblioteke QT.

Muteksi su implementirani klasom *QMutex*. Osnovni metodi:

- *void lock()* zaključava muteks
  - ako je već zaključan, čeka da se otključa oa ga zaključava
  - u svakom slučaju, vraća se tek kada je uspešno zaključan
- *void unlock()* otključava muteks
  - ako je zaključan od strane iste niti, otključaga
  - ako je zaključan od strane druge niti, proizvodi grešku
  - ako nije zaključan ponašanje je nedefinisano
- *bool tryLock()* pokušava da zaključa muteks
  - ako je muteks otključan, zaključava ga i vraća *true*

  - ako je već zaključan, ne radi ništa i odmah vraća *false*
  - opcioni argument je maksimalno trajanje čekanja u ms
- konstruktor *QMutex ( RecursionMode mode = NonRecursive )*
  - opcioni argument može da bude Qmutex::Recursive
  - tada jedna nit može da zaključa muteks više puta uzastopno
  - mora da ga otključa onoliko puta koliko ga je puta zaključala

## Upotreba

Uobičajeno e da se jedinm muteksom obezbeđuje jedan podatak ili skup podataka koji u jednom trenutku sme da upotrebljava samo jedna nit. Na početku atomičnog postupka muteks se zaključava, a na kraju atomičnog postupka se otključava. Veoma je važno da se svaki zaključani muteks otključa. Ako jedan isti muteks zaključava različite podatke, onda se potencijalno spušta nivo konkurentnosti. Ako više muteksa zaključava jedan isti podatak, otvara se prostor za ozbiljne previde u upravljanju tim resursom.

# 270 Čemu služi klasa QMutexLocker biblioteke QT? Objasniti detaljno.

Klasa QMutexLocker pojednostavljuje rad sa mutekstima i čini ga ispravnim i u slučaju izuzetaka

- upotrebljava se isključivo za automatske objekte
- objekat se pravi za konkretan muteks
  - konstruktor *QMutexLocker(QMutex *)*
  - činom pravljenja objekta muteks se zaključava
- pri uklanjanju se otključava katanac
  - radi ispravno čak i u kontekstu izuzetaka
  - dobar način da se preduprede grešeke sa propuštanjem otključavanja

# 271 Šta su katanci? Kako se upotrebljavaju?

Muteksi se ponašaju kao specijalan slučaj katanaca - imaju samo ekskluzivan režim pristupa. Apstrakcija katanca omogućava različite pristupe pod različitim uslovima. Uobičajeno

- ako jedna nit čita podatke, tada i druge niti mogu čitati podatke - **deljeni katanac**
- ako jedna nit menja podatke, niko drugi ni na koji način ne sme kotistiti podatke

# 272 Objasniti podršku za katance u okviru biblioteke QT.

Katanci su implementirani klasom *QReadWriteLock*. Osnovni metodi

- *void LockForRead()*
  - postavlja katanac za čitanje
  - ako je zaključan za pisanje, metod čeka da se katanac za pisanje otključa
- *lockForWrite()*
  - postavlja katanac za pisanje
  - ako je zaključan (bilo kako), metod čeka da se svi katanci otključaju
- *void unlock()*

  - otključava katanac
- *bool tryLockForRead(), bool tryLockForWrite()*
  - pokušavaju da zaključaju katanac
  - opcioni argument je maksimalno trajanje čekanja u ms
- kostruktor QReadWriteLock( RecursionMode mode = NonRecursive )
  - opcioni argument može da bude QReadWriteLock::Recursive

## 273 Čemu služi klasa QReadLocker biblioteke QT? Objasniti detaljno.

Slično klasi QMutexLocker pojednostavljuje rad sa katancima. Dobar način da se preduprede greške sa propuštanjem otključavanja.

## 274 Čemu služi klasa QWriterLocker biblioteke QT? Objasniti detaljno.

Slično klasi QMutexLocker pojednostavljuje rad sa katancima. Dobar način da se preduprede greške sa propuštanjem otključavanja.

## 275 Šta je sinhornizacija? Šta se sve može sinhronizovati u konkurentnim programima?

Da bi se jedinice izvršavanja sinhornizovale, one moraju da koriste neke zajedničke resurse za obaveštenje. U slučaju niti, sinhornizacija se može izvoditi putem deljenih resursa. Najvažnije je voditi računa o atomičnosti operacija.

## 276 Šta su semafori?

Jedan od osnovnih mehanizama za sinhronizaciju i deobu resursa predstavljaju semafori. Semafori omogućavaju da se neki brojač kontrolisano i bezbedno povećava, smanjuje i proverava. Imaju semantiku brojača slobodnih resursa.

## 277 Objasniti podršku za semafore u okviru biblioteke QT.

Semafori su implementirani klasom *QSemaphore*. Osnovni metodi:

- *int available()*
  - vraća trenutnu vrednost semafora
- *void acquire( int n = 1)*
  - smanjuje semafor za $n$
  - ako je n>available(), čeka se da postane n<=available()
- *void release( int n = 1)*
  - uvećava semafor za $n$
  - metod se upotrebljava i za "pravljenje novih resursa"
- *bool tryAcquire( int n = 1)*
  - pokušava da smanji semafor za $n$ ako spe vraća *true*, a inače vraća *false*

- *bool tryAcquire( int n, int timeout)*
  - pokušava da smanji semafor za $n$ i čeka najduže *timeout* ms
  - ako uspe vraća *true*, a inače *false*
- konstruktor *QSemaphore(int n=0)*
  - pravi semafor i inicijalizuje brojač datom vrednošću

## 278 Kakvi mogu biti potprogrami u kontekstu konkurentnog programiranja?

Potprogrami mogu imati relativno složen odnos sa konkurentnim okruženjem. Uopšteno mogu biti:

- sa jedinstvenim povezivanjem
- sa ponovljenim povezivanjem (eng. reentrant)
- bezbedne po niti (eng. thread-safe)

## 279 Šta su potprogrami sa jedinstvenim povezivanjem?

Kažemo da je funkcioja sa jedinstvenim povezivanjem ako ne sme biti istovremeno upotrebljivana u različitim nitima.

Mogući uzorci

- u telu funkcije se upotrebljavati globalni podaci ili drugi globalni resursi na način koji nije bezbedan
- ti glavni resursi se upotrebljavaju nezavisno od navedenih argumenata

## 280 Šta su potprogrami sa ponovljenim povezivanjem?

Kažemo da je funkcija sa ponavljanim pozivanjem ako sme biti istovremeno upotrebljivana u različitim nitima samo uz pretpostavku da se ne upotrebljava na istim podacima.

Mogući uzorci:

- upotreba podataka ili resursa na način koji nije bezbedan

## 281 Šta su potprogrami bezbedni po nitima?

Kažemo da je funkcija bezbedna po niti ako sme biti istovremeno upotrebljivana u različitim nitima bez ograničenja. Funkcija je bezbedna po niti ako sve podatke (osim lokalnih automatskih) upotrebljava na način koji pretpostavlja konkurentno okruženje.

## 282 Kakve mogu biti klase u kontekstu konkurentnog programiranja? Objasniti.

Sve što važi za funkcije važi i za klase. Klasa je

- sa jedinstvenim povezivanjem
  - ako je bar jedan metod klase sa jedinstvenim povezivanjem
- sa ponovljenim povezivanjem
  - ako nijedan metod nije sa jedinstvenim povezivanjem

  - bar jedanm metod je sa ponovljenim povezivanjem
  - nijednom podatku se ne može neposredno pristupati
- bezbedna po niti
  - ako su sbi metodi bezbedni po niti
  - nijednom podatku se ne može neposredno pristupati

## 283 Koje su najčešće greške pri pisanju konkurentnih programa? Objasniti.

- Previđanje problema deljenja podataka pri pisanju funkcija i klasa
- Previđanje neprilagođenosti upotrebljivanih klasa (funkcija) konkurentnim okruženjima tj. njihova upotreba kao da su bezbedne po niti, a da one to zapravo nisu

## 284 Navesti neke načine razvoja programa koji omogućavaju bezbedno pisanje konkurentnih programa? Objasniti.

**Pisanje čisto funkcionalnog koda** - potprogrami se pišu tako da

- koriste argumente i lokalne promenljive
- ne menjaju argumente
- potprogrami se pozivaju samo sa sigurnom argumentima koji ne mogu da budu menjani u drugim nitima

**Upotreba lokalnih podataka i podataka koji su lokalni za nit**

- takve podatke nijedna druga nit ne može da koristi
- samim tim, čim su takvi podaci bezbedni, bezbedan je i potprogram koji samo njih upotrebljava

**Obezbeđivanje međusobnog isključivanja**

- muteksi, katanci, ...

**Obezbeđivanje atomičnih operacija**

- kritične sekcije, muteksi, katanci, ...

## 285 Kada dolazi na red staranje o ponašanju koda u konkurentom okruženju?

- Ne odlagati staranje o bezbednosti niti za kasnije
  - od samog početka projektovanja neke klase ili funkcije mora se imati u vidu da li se planira upotreba u okruženju sa više niti
  - naknadno prilagođavanje može da bude veoma skupo, pa i uz skoro potpuno preplavljanje klasa
- Deljenje podataka
  - izbegavati deljenje podataka među nitima
  - posebno izbegavati menjanje istih podataka od strane više niti
  - ako je deljenje neophodno, obavezno zaključavati (podatke ili kod)
  - minimizovati deljenje podataka

- Zaključavanje
  - izbegavati dugotrajna zaključavanja
  - ako je potrebno postaviti veći broj katanaca, uvek ih postavljati istim redom (makar i azbučnim)
- Model konkurentnosti
  - strogo definisati namenu niti
  - ako je nit namenjena tačno jednom poslu ne omogućavati joj pristup podacima i kodu koji se ne odnose na taj posao
  - striktnim vezivanjem niti za što uži prostor koda i podataka smanjuje se prostor za sukobljavanje niti
  - bezbednije je napraviti više specijalizovanih niti nego manje opštih
  - dokumentovati model konkurenosti
- Ne praviti bezbednim po niti kod za koji to nije potrebno
  - to je gubljenje vremena tokom razvoja
  - postavljanje suvišnih katanaca spušta performanse i vodi mrtvim petljama

## 286 Kako se bira gde se i kako postavljaju muteksi i katanci?

- Ne stavljati katance i mutekse odoka
  - katanci, muteksi i slični mehanizmi za izolovanje će kod učiniti bezbednim ako se dobro upotrebljavaju
  - međutim, istovremeno će i spustiti nivo paralelnosti i iskorišćenosti višeprocesorske mašine
  - bolje je stavljati više manjih katanaca nego manje većih
  - sitniji katanci smanjuju verovatnoću čekanja niti
  - povećanje broja katanaca stvara uslove za nastajanje mrtvih petlji
- Ne štiti samo kod ili samo podatke
  - značajna unapređenja kvaliteta programa i nivoa performansi se mogu postići ako se zaključava prava vrsta resursa
  - nekada je bolje štiti podatke, ali ne uvek
    - najčešće je dobro štiti podatke
    - ako je mnogo podataka koji se dele, dolazi do opasnosti od mrtvih petlji
  - nekada je bolje štiti kod, ali ne uvek
    - obično je bolje štiti kod jednim katancem (muteksom, ...) nego podatke sa mnogo katanaca (muteksa, ...)
    - ali ako se podaci koje taj kod čita/menja mogu koristiti i na nekom drugom mestu, onda to nije dovoljno

## 287 Navesti osnovne vidove međuprocesne komunikacije.

Neki od uobičajenih vidova međuprocesne komunikacije (eng. inter-process communication - IPC) su :

- signali
- soketi (sockets)
- slanje poruka
- redovi poruka (message queues)
- cevi (pipes)

- imenovane cevi (named pipes)
- semafori, ...
- deljena memorija
- datoteke preslikane u memoriju (memoty mapped files)
- datoteke

## 288 Objasniti razliku između komunikacije među procesima i komunikacije među nitima.

Komunikacija između procesa se može odvijati usključivo putem komunikacionih kanala obezveđenih na nivou operativnog sistema ili računarske mreže.

Komunikacija između niti se može odvijati i putem resursa koji se dele među nitima, kao što su memorija ili otvorene datoteke. Mora se voditi računa o ispravnom deljenju komunikacionih resursa tj. atomilnosti operacija na deljenim resursima. Često je najbolje i za komunikaciju među nitima koristit mehanizme namenjene za procese.

## 289 Šta je softverska metrika?

Metrika je nauka o merama i merenju.
**Softverska metrika** se bavi merenjem različitih karakteristika siftverskih rešenja. Cilj je opisivanje stanja razvojnog projekta. Numeričkim opisivanjem različitih aspekata razvojnog projekta se stiču uporediva znanja o projektu.

## 290 Navesti najvažnije tipove softverskih metrika.

- Neposredne (apsolutne) mere
  - npr. broj jedinica koda ili broj linija koda
- Indirektne/izvedene mere
  - npr. količnih drugih mera
- Intervalne mere
  - mera iskazana kao opseg vrednosti
- Normalne mere
  - iskazana rečima, npr. "jeste", "nije", "plavo", ...
- Uporedne mere
  - npr. "veći od", "manji od", ...

## 291 Objasniti vrste metrika u razvoju softvera.

**Metrike praćenja razvoja softvera**

- iskazuju u kojoj meri tok projekta prati planove
- odnose se na
  - proces razvoja
  - upotrebljene resurse (vreme, troškovi i sl.)

**Metrike softvera**

- iskazuju neke merljive odlike softvera
- odnose se na softver, kao proizvod razvojnog procesa

## 292 Navesti nekoliko metrika praćenja razvoja softvera. Šta one opisuju?

Zovu se i metrike upravljanja zato što su važne pre svega upravljačkom timu:

**Mera napretka**

- iskazuje uspešnost praćenja planova
- predstavljanje broja planiranih i ostvarenih zadataka u odnosu na vreme
- na kvalitet utiče ujednačenost celičine zadataka

**Mera napora**

- iskazuje planirani i ostvareni broj radnih sati u odnosu na vreme

**Troškovi**

- iskazuje planirani i ostvareni utrošak sredstava u odnosu na vreme

**Problemi**

- uskazuje broj otvorenih i rešenih problema u odnosu na vreme

**Stabilnost zahteva**

- iskazuje broj zahteva tokom vremena

**Stabilnost veličine**

- iskazuje veličinu softvera, obično iskazanu u broju jedinica koda, tokom vremena

## 293 Navesti nekoliko metrika dizajna razvoja softvera. Šta one opisuju?

Iskazuju različite merljive odlike softvera.

Neposredne mere softvera su često lako merljive veličine:

- broj jedinica koda
  - iskazuje broj celina koda

  - može da predstavlja broj
    * programskih datoteka
    * klasa
    * paketa
    * komponenti
    * izvršnih datoteka
- broj linija koda
  - iskazuje količinu napisanog koda
  - obično obuhvata
    * sve neprazne linije koje nisu komentari
    * uključujući izvršni kod i deklaracije, definicije i druge vrste neizvršnog koda
- broj funkcionalnih elemenata koda
  - opisuje broj nekih funkcionalnih elemenata
    * broj klasa u paketu
    * broj metoda u klasi
    * broj metoda u interfejsu paketa
    * ...

Izvedene mere softvera mogu da opisuju veoma složene odlike dizajna softvera:

- kohezija jedinice koda
- spregnutost jedinice koda
- stabilnost jedinice koda
- apstraktnost jedinice koda

## 294 Objansiti metriku stabilnost paketa.

Stabilnost paketa predstavlja njegovu tendenciju da se ne menja tokom vremena.

- $C_a$ = broj klasa u drugim paketima koje zavise od kalsa u posmatranom paketu
  - spregnutost prema paketu (eng. afferent coupling)
  - što ih je više, to će promene paketa biti teže izvodive, a time i ređe
- $C_e$ = broj klasa u posmatranom paketu koje zavise od klasa u drugim paketima
  - spregnutost od paketa (eng. efferent coupling)
  - što ih je više, to će njegove promene češće biti potrebne
- $I$ = nestabilnost, može se izračunati kao:

$$I = \frac{C_e}{C_a + C_e}$$

  ima opseg $[0, 1]$

Nije moguće da svi paketi budu stabilni. Ako bi bili, to bi značilo da je čitav sistem nepromenljiv. Umesto toga želimo da neki paketi budu stabilniji a neki manje stabilni.

**Princip odnosa zavisnosti i stabilnosti:**
Zavisnost paketa bi trebalo da ide od nestabilnih prema stabilnim paketima. Nije dobro ako postoji zavisnosti stabilnog paketa od nekog koji je relativno fleksibilan (nestabilan).

# 295 Objansiti metriku apstraktnost paketa.

Apstraktnost paketa je relativna zastupljenost apstraktnih klasa u paketu:

- $N_a$ = broj apstraktnih klasa u paketu
- $N_c$ = broj klasa u paketu
- $A$ = apstraktnost

$$A = \frac{N_a}{N_c}$$

ima opseg [0,1]

**Princip odnosa stabilnosti i apstraktnosti:**
Praket treba da bude onoliko apstraktan koliko je stabilan. Drugim rečima, manje apstraktni paketi bi trebalo da zavise od apstraktnih, a ne obrnuto.

# 296 Objasniti odnos metrika stabilnosti i apstraktnosti paketa.

Slika 10: Dijagram zavisnosti nestabilnosti i apstrakcije

Poželjno je da odnost apstraktnosti i stabilnosti bude što bliže glavno sekvenci. Udaljenost $D$ se računa kao:

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$

ili normalizovano kao:

$$D = |A + I - 1|$$

# 297 Objasniti metriku funkcionalna kohezija paketa.

Jedan način izražavanja funkcionalne kohezije paketa je u obliku količnika vroja zavisnosti među klasama paketa i broja paketa. Predstavlja intenzitet odnosa paketa sa spostvenim klasama

- $R$ = broj međusobnih zavisnosti među klasama paketa
- $N$ = broj klasa u paketu
- $H$ = relaciona kohezija

$$H = \frac{R + 1}{N}$$

Alternativa je da se svaka zavisnost dveju klasa predstavi brojem metoda koji je ostvaruju.

# 298 Šta su sistemi za kontrolu verzija? Objasniti.

Sistem za kontrolu verzija je softver za upravljanje izmenama u programskom kodu, dokumentima i drugim vrstama fajlova. Obično se razmatra u kontekstu projekata razvoja softvera, ali može imati i druge namene. Postoji veliki broj sistema za kontrolu verzija:

- CVS
- SVN (Subversion)
- Git
- Bazaar
- ...

# 299 Objasniti arhitekturu i navesti osnovne operacije pri radu sa sistemima za kontrolu verzija.

Sistem za kontrolu verzija (SKV) obično ima arhitekturu klijent-server. Server je instanca sistema za kontrolu verzija, a klijent je korisnik SKV
Neki sistemi imaju distribuiranu arhitekturu

- decentralizovani
- omogućavaju podnošenje i bez povezivanja sa serverom
- odlaganje razrešavanja i spajanja

SKV (bilo da je centralizovan ili distribuiran) sadrži spremište. Spremište e mesto na kome se čuvaju različite verzije projekta. Obično se implementira pomoću odgovarajuće baze podataka.

## Koncept upotrebe

Prvi korak je pravljenje *radne kopije*

- radna kopija (eng. working set, working copy) je lokalna kopija verzije na klijentu, koja se menja tokom rada
- osim fajlova projekata sadrži i metapodatke o verzijama fajlova
- postupak pravljenja radne kopije se naziva *preuzimanje* (eng. checkout)
- radna kopija ne mora uvek da se pravi samo na osnovu osnovne linije

Po potrebi se radna kopija može *ažurirati* (eng. update, sync) preuzimanjem aktuelne verzije iz spremišta.

Nakon menjanja fajlova radne kopije, izmenjena verzija se podnosi (eng. commit, checkin, install, record) tj. postavlja se u spremište kao nova aktuelna verzija.

## Uvoz i izvoz

Nakon pravljenja novog projekta u spremištu, obično se projekat inicijalno popunjava uvozom (eng. import) odgovarajuće lokalne kopije.
Po potrebi se verzija može izvoziti (eng. export), pri čemu se pravi nova lokalna kopija, nalik na radnu kopiju, ali koja ne sadrži metapodatke o preuzimanju i spremištu.

## Osnovna linija

Sekvenca verzija koja se uzastopno menjaju naziva se glavna linija. Usled mogućnosti grananja, može postojati više glavnih linija.
Osnovna glavna linija se naziva *osnovna linija* (eng. trunk, main, baseline).
Poslednje podnošenje osnovne linije se naziva *glava* (eng. head).

## Izmene

Svakim podnošenjem se prave nove verzije pojedinačnih menjanih fajlova i nova verzija projekta. Izmene se prave pravljenjem novih grana. Poseban vid izmena je tzv. *promocija* (eng. promote) - kopiranje verzije fajla koja nije kontrolisana u spremište (ili radnu kopiju).

# 300 Šta je spremište? Šta sadrži? Kako je organizovano?

SKV (bilo da je centralizovan ili distribuiran) sadrži spremište. Spremište e mesto na kome se čuvaju različite verzije projekta. Obično se implementira pomoću odgovarajuće baze podataka.

## 301 Šta je radna kopija u kontekstu upotrebe sistema za kontrolu verzija?

- radna kopija (eng. working set, working copy) je lokalna kopija verzije na klijentu, koja se menja tokom rada
- osim fajlova projekata sadrži i metapodatke o verzijama fajlova
- postupak pravljenja radne kopije se naziva *preuzimanje* (eng. checkout)
- radna kopija ne mora uvek da se pravi samo na osnovu osnovne linije

## 302 Objasniti pojam oznake u kontekstu upotrebe sistema za kontrolu verzija.

Često je pojedine verzije potrebno posebno označiti

- na primer, verzija koja se javno publikuje
- ali i verzija koja presstavlja neki interno značajan trenutak

Označena verzija (ili kraće smao oznaka) (eng. tag, label) je verzija koja je posebno označena radi kasnijeg referisanja

- verzija se označava davanjem odgovarajućeg imena
- u zavisnosti od sistema postoji određena sloboda u određivanju naziva oznake

## 303 Objasniti grananje u kontekstu upotrebe sistema za kontrolu verzija.

Često je potrebno istovremeno razvijavati više različitih verzija koda

- na primer, i nakon objavljivanja nove verzije potrebno je razvijati popravke za staru
- razvijanje novih funkcija za koje se ne zna kada će biti uvrštene u osnovnu liniju

Verzija koja se razvija nezavisno od glavne linije razvoja naziva se *grana* (eng. branch). Svaka grana ima svoju glavnu liniju, kao što čitav projekat ima svoju osnovnu liniju. Fajlovi i direktorijumi se mogu deliti između grana - izmene deljenih fajlova se takođe dele u svim granama. Grane se u nekim slučajevima spajaju nazad u stablu. To se naziva *povratno integrisanje* (eng. reverse integration). Npr. razvijanje komponente za koju nije unapred poznato kada će biti uključena u osnovnu liniju.

## 304 Šta su konflikti i kako se rešavaju u kontekstu upotrebe sistema za kontrolu verzija?

Često se dešava da se neki fajl menja od strane vie klijenata "istovremeno"

- ko prvi podnese izmenjen fajl, imaće jednostavniji posao
- onaj ko pokuša podnošenje kao drugi, biće obavešten da je fajl u međuvremenu menjan i da je potrebno da preduzme spajanje verzija

Konflikti se, pored opisanog slučaja, dešavaju i kada se

- neka grana spaja sa stablom ili sa drugom granom
- u jednoj grani ispravi neki problem koji je postojao i pre grananja, pa je potrbno ispravke preneti i u ostale grane

# 305 Šta je spajanje verzija u kontekstu upotrebe sistema za kontrolu verzija?

*Spajanje verzija* (eng. merge, integration) je operacija spajanja dva skupa izmena fajla ili skupa fajlova. Primenjuje se kada je potrebno rešiti konflikte.

Sastoji se od skupa pojedinačnih razrešavanja (eng. resolve) problema. Razrešavanje je manuelna intervencija radi rešavanja sukobljenih izmena na istom dokumentu.

# 306 Objasniti primer strategije označavanja verzija.

Određivanje strategije označavanja verzija je sastavni deo upravljanja razvojnim projektom. Uobičajeno četvorodelno označavanje verzije
<glavna verzija>.<podverzije>[.<popravka>[.<revizija>]]

**Glavna verzija**

- menja se kada su uvedene značajne izmene u odnosu na prethodnu verziju softvera
- žnačajne izmene"može da ima različita značenja
- dodate su suštinski nove funkcije
- kod je pisan u potpunosti iznova
- softver se prodaje kao poseban proizvod

**Podverzija**

- menja se kada su uvedene izmene koje nisu samo popravke uočenih nedostataka, ali ne zavređuju novu glavnu verziju
- dodate su neke nove funkcije
- menjane su neke postojeće funkcije

**Popravka**

- podrazumeva da nisu uvedene izmene u funkcionalnosti softvera već samo ispravke uočenih nedostataka
- nekad su kao sastavni deo ispravki obezbeđene i manje izmene ili novine

**Revizija**

- broj koji označava reviziju popravke
- obično predstavlja smao internu oznaku za izmene koda ili dokumentacije koje imaju sasvim ograničen značaj

Često se razdvajaju javni i interni brojevi verzija. Javne verzije se obično sastoje od dva dela, a popravke se ili dodaju kao treći broj ili se označavaju opisno:

- DB2 v9.1 – fixpack 3
- Firefox 3.6.3

Brojevi popravke i revizija su često internog karaktera i koriste se samo u okviru razvojnog tima.

## Dodatne oznake

Da bi bio jasan kontekst neke verzije, često se broju dodaju opisni elementi, na primer:

- 3.1a - alfa verzija, interna pregledna verzija

- 3.1b - beta verzija, interna verzija koja može koristiti radi upoznavanja softvera ali nije još dobršena
- 3.1rc1 - kandidat za objavljivanje, predstoje samo popravke poznatih i kasnije uočenih nedostataka i eventualno dodavanje nekih manjih funkcionalnosti za koje se naknadno ustanovi da su neophodne
- 3.1rc2 - kadnidat za objavljivanje, predstoje samo popravke poznatih i kasnije uočenih nedostataka

## Postupak numerisanja

Obično se pre prvog publikovanja softver vodi sa glavnim brojem verzije "0- sve verzije koje prethodne verziji "1šu "nezvanične". Ne postoji koncenzus oko toga kako se interno numerišu interne razvojne verzije koje prethode novoj glavnoj verziji.

**Numerisanje razvojnih verzija**

**v1** :

- nova glavna verzija ima sve ostale brojeve verzije (osim glavnog) jednake "0"
- 2.0.0
- tada je problem numerisanja internih verzija
- obično razvoj verzija 2 u takvim uslovima počinje od npt. 1.9.0.0

**v2**

- prva interna verzija koda nove glavne verzije odmah dobija svoj glavni broj
- tada se publikovanje odvija sa nekim drugim brojem, na primer 2.0.7

**v3** : broju popravke se daju posebna značenja, na primer:

- 2.0.0.x – alfa verzija
- 2.0.1.x – beta verzija
- 2.0.2.x – kandidat za objavljivanje
- 2.0.3.x – kandidat za objavljivanje 2
- 2.0.4.x – objavljena verzija

**v4**

- potpuno se razdvajaju interni brojevi verzija od javnih
- interna verzija 2.0.12.5 se ptoglašava za javnu verziju 2.0.0

**Numerisanje razvojnih verzija**

Svaki od brojeva počinje od 0 i povećava se

- verzija 3.10.1 je posle verzije 3.9.2

Za sve brojeve, osim poslednjeg, uobičajeno je da se vraćaju na 0 kada se prethodni broj promeni

- posle 3.1.6 ide 3.2.0
- tako se omogućava da se i posle objavljivanja nove verzija (podverzije) i dalje obezbeđuju popravke prethodnih verzija (posle 3.2.0 može da se objavi 3.1.7)

Poslednji broj može da ima različito značenje

- revizija u okviru popravke - vraća se na nulu nakon promene broja popravke
- redni broj punog građenja sistema

  - nikada se ne vraća na 0
  - tada je samo ovaj broj dovoljan za interno označavanje verzije
  - obično se taj broj koristi samo za interne svrhe
  - samo prva 2 ili 3 se koriste za javno označavanje
- npr:
  - 2.0.0.782
  - 2.0.0.791.a
  - 2.0.0.842.b
  - 2.0.0.864.rc1
  - 2.0.0.872.rc2
  - 2.0.0.875.public – javno objavljena verzija 2.0

## 307 Šta su sistemi za praćenje zadataka i bagova? Objasniti namenu i osnovne elemente.

*Sistemi za praćenje bagova* predstavljaju alate za upravljanje evidencijom i komunikacijom u vezi sa uočenim neispravnostima. Predstavljaju poseban slučaj opštijih sistema za praćenje zadatka. Nazivaju se i sistemi za praćenje poslova. Eng. termini: bug tracking system, issue tracking system, ticket system, ...

## 308 Navesti i ukratko objasniti osnovne koncepte sistema Redmine.

*Savremeni sistemi za praaćenje poslova* su fleksibilni

- podržavaju više vrsta kartica (npr. bagovi, zadaci, nagradnje, diskusije i sl.)
- za svaku vrstu se mogu definisati različita stanja i načini prolazaka kroz stanja
- različite grupe korisnika imaju različita prava, u zavisnosti od stanja kartice
- automatsko slanje obaveštenja elektronskom poštom
- dokumentacija
- programski kod
- ...

## 309 Objasniti ulogu stanja kartica i način njihovog menjanja (na primeru sistema Redmine).

Kartica (ili stavka, eng issue, ticket) je jedan evidentiran bag / problem / posao / zadatak

- kartica predstavlja centralni objekat svakog sistema za praćenje zadatka
- kartica može imati različita stanja koja opisuju koje se aktivnosti očekuju u odnosu na zadatak
- obično joj se može dodeljivati kategorija, prioritet, detaljan opis

# 310 Šta čini dokumentaciju softvera?

Dokumentacija softvera je tekst i/ili ilustracija koja prati računarski softver. Ona objašnjava kako se koristi softver ili kako on radi. Može da ima različito značenje za različite tipove korisnika.

Obično sadrži:

- Opis atributa, mogućnosti ili karakteristike softvera.
- Opis namene - šta softver radi (ili bi trebalo da radi).
- Arhitekturu/dizajn softvera - opis samog softvera. Uključuje odnose sa softverskih komponenti sa okruženjem.
- Tehničku dokumentacija - sastoji se od koda, algoritama, interfejsa, i aplikacionog programskog interfejsa.
- Korisničku dokumentaciju - uputstva za krajne korisnike, administratore sistema i stručno osoblje.
- Marketing - kako da se reklamira i prodaje proizvod i analizu zahteva tržišta.

# 311 Kome je i zašto potrebna dokumentacija?

Svima, ali svakome na drugi način i u drugačijem obliku

**Naručiocu posla**

- da zna šta je naručio
- da bude siguran da izvođači znaju šta je naručio
- da misli da je dobio šta je tražio

**Rukovodiocima projekta**

- da vide šta su zahtevi
- da vide šta je urađeno

**Projektantima**

- da znaju šta je potrebno da naprave
- da zapišu implementatorima kako zamišljaju da to naprave
- da misle da je implementirano ono što su isprojektovali i kako su isprojektovali

**Implementatorima**

- da znaju šta je potrebno da naprave
- da znaju kako je potrebno da to naprave
- da izveste rukovodioce o tome kako veruju da su to napravili
- da objasne budućim korisnicima programskog koda šta je šta i čemu služi
- da objasne korisnicima softvera šta softver radi i kako se koristi

**Korisnicima**

- da vide čemu softver služi kako se upotrebljava

# 312 Navesti i ukratko objasniti osnovne opravdane i neopravdane motive za pravljenje dokumentacije.

**Zato što to traži klijent**

- ✓U pitanju je poslovna odluka
  - ⊗ Klijent obično ne ume da proceni potreban obim, kao ni pravi trenutak za pisanje dokumentacije
- ✓iz ugla razvojnog tima je to legitiman razlog, ali je istovremeno i frustrirajuće
- ⊗ Često proizvodi nepotrebne troškove, bez značajnih pozitivnih posledica ako nije praktično neophodna u trenutku kada se traži

**✓Da bi se formalizovale specifikacije**

formalizovanje interfejsa za upotrebu komponenti

važno za sve koji će razvijati i koristiti komponentu

precizno objavljivala ponašanja komponenti

važno za sve koji će razvijati i koristiti komponentu

konceptuialan i implementacioni model komponente

važno svim učesnicima u njenom razvoju

**✓Da bi se podržala komunikacija sa udaljenim timovima**

nisu uvek svi članovi tima na istoj lokaciji

dokumentacija je kao vid statičke komunikacije pogodna za informisanje dislociranih članova tima

posebno važno u slučaju otvorenih projekata, *autosorsinga* i sl.

**Da bi se definisali ciljevi rada za neku drugu grupu**

- **✓u nekim slučajevima može da bude dobro**
  - ako je u pitanju specifikacija onoga što se pravi u tekućem razvojnom ciklusu
  - ako samo ukratko i konceptualno ukazuje na dalje planove
- **⊗ ali često nije dobro**
  - ako je dokumentacija nepotrebno detaljna i obimna
  - ako opisuje zadatke i ponašanje koji nisu deo tekućeg razvojnog ciklusa
- usmena komunikacija je obično efikasnija

**✓Da bi se podstaklo širenje i memorisanje informacija u timu**

informacije o urađenom i planiranom su važne za razvoj, održavanje i upotrebu

**Radi praćenja razvoja**

- ✓dokumentacija omogućava uvid u dostignuća razvoja (za to nije potrebna obimna i precizna dokumentacija)
- ⊗ klijenti često imaju stav da je dokumentacija pokazatelj uspešnosti razvoja
  - potpuno pogrešno
  - dokumentacija se može napisati i o nečemu što nije čak ni dovoljno detaljno zamišljeno, a kamoli napravljeno

**✓Da bi se nešto temeljno razmotrilo**

pisanje dokumentacije je dobar vid proveravanja pretpostavki

pri pisanju dokumentacije se podstiče kritičko razmatranje

ovaj motiv često podstiče specifične vidove dokumentacije, čija važnost rapidno opada sa protokom vremena

⊗ **Po inerciji**

posledica navike klijenta

klijent ne zna za drugačiju praksu

ali to ne znači ni da je ona dobra

ni da je loša

ali navika nije opravdanje

zato što razvojni proces tako propisuje

nijedan proces nije dovoljno dobar i pouzdan da se slepo sledi

⊗ **Kao vid obezbeđenja klijenta**

✓ako razvijalac nije dovoljno dobar, onda klijent može predati projekat i dokumentaiciju drugom izvođaču, da bi nastavio posao

ali ako razvijalac nije dovoljno dobar, onda nije dobra ni dokumentacija

informacije razmenjene u ovom obliku često su dezinformacije o pogrešnim planovima

čak i kada je dokumentacija dobra, ona opisuje kako je nešto isplanirao jedan tim, a drugi može da ima drugačije iskustvo, pristupe problemu i praksu

## 313 Objasniti ulogu dokumentacije kao vida specifikacije zahteva projekta.

Pogledati pitanje 312

## 314 Objasniti ulogu dokumentacije kao sredstva za komunikaciju.

Pogledati pitanje 312

## 315 Objasniti ulogu dokumentacije u razmatranju nedoumica u projektu.

Pogledati pitanje 312

## 316 Objasniti podelu dokumentacije po nameni.

Prema nameni se razlikuju:

- Korisnička dokumentacija
  - Namenjena je krajnjem korisniku softvera
  - Objašnjava sve aspekte primene softvera
- Tehnička dokumentacija
  - Namenjena je svim (sadašnjim i budućim) učesnicima u razvoju ili održavanju softvera
  - namenjena je i tehničkim licima koja moraju da pružaju podršku korisnicima

Dokumentacija se odnosi na ceo softverski projekat

- ne samo na gotov program
- ne samo na postavljen zadatak
- već na sve segmente posla, od prve zamisli o softveru do gotovog softvera

## 317 Šta obuhvata korisnička dokumentacija softvera?

Korisnička dokumentacija obuhvata

**Opis čitavog sistema** - uopšteni opis funkcija koje sistem pruža

**Uputstvo za instalaciju** - objašnjava kako se sistem priprema za rad, kako se prilagođava specifičnom okruženju, potrebama korisnika ili konkretnom računarskom sistemu

**Vodič za početnike** - pruža pojednostavljena objašnjenja kako započeti upotrebu sistema, obično u obliku tutorijala

**Referentni priručnik** - detaljan opis svih karakteristika i mogućnosti sistema i načina njihove upotrebe

**Prilog o dopunama / Opis izdanja** pregledni izveštaj o svim izmenama i dopunama novog izanja softvera

**Skraćeni referentni pregled** - pomoćni priručnik za brzo podsećanje

**Upuststvo za administraciju** - tehnički priručnik sa detaljnim objašnjenjima o mogućim naprednim prilagođavanjima specifičnim okolnostima i sa detaljnim opisom problema

## 318 Šta obuhvata tehnička (sistemska) dokumentacija softvera?

Sistemska dokumentacija obuhvata

**Vizija** - objašnjama ciljeve sistema

**Specifikacija i analiza zahteva** - pruža detaljan opis zahteva ugovorenih između zainteresovanih strana (naručioci, klijenti, korisnici, projektanti, izvođači . . . )

**Specifikacija ili projekat** - detaljni opisi svih aspekata implementacije:

- kako se sistem deli na celine
- kako se implementiraju pojedinačni zahtevi ili celine
- koju ulogu ima koja komponenta sistema

**Opis implementacije** - detaljni opisi:

- kako se pojedini elementi sistema modeliraju na konkretnim programskim jezicima
- opisi značajnijih algoritama
- specifikacije komunikacije

**Plan testiranja softvera** - opis protokola testiranja softvera (testovi jedinica koda i sve ostale vrste razvojnih testova)

**Rečnik podataka** - sadrži rečnik termina i posebno opis osnovnih vrsta podataka koji se koriste u sistemu

## 319 Kakav je odnos agilnog razvoja softvera prema pisanju dokumentacije? Koji vidovi dokumentacije se podstiču a koji ne?

Agilne metodologije podstiču specifičan pristup pravljenju i održavanju dokumentacije.

Motivacija:
Razvojna dokumentacija je često neažurna. Velika količina dokumentacije nije korisna, ili bude izmenjena pre upotrebe, ili postane neažurna.
Dokumentacija nije cilj nego sredstvo za ostvarenje komunikacije.
Cena održavanja dokumentacije je visoka.

Specifikacije: u agilnom razvoju u obliku korisničkih celina (sasvim površne). Korisničke celine - samo okvirni opis, bez pojedinosti, ukazuje se na sadržaj i obim.

Modeli: početna faza implementacije korisničkih celina je pravljenje modela. Modeli su deo razvojne dokumentacije. Na osnovu njih može da se pravi i formalna dokumentacija, ali najčešće prestaje da bude ažurna već tokom razvoja.
Modeli se prave u različitim fazama implementacije. Predstavljaju sredstvo za razmatranje i razmenu informacija o konceptualnom rešenju. Neophodan su deo razvojne dokumentacije ali najčešće postaje neažuran već tokom razvoja, ili kasnije, tokom održavanja. Neažurna dokumentacija je *dezinformacija*. Mogu da prerastu u trajnu dokumentaciju ako je model stabilan, ako je značajan za buduće faze razvoja, ako klijent želi da investira.

Dokumenti: ozvaničeni modeli, opisi ponašanja ili programskog koda. Prave se relativno reko, zato što ih je skupo održavati, osim ako su potrebni za dalji rad (uključujući održavanje). Važno je da imaju oznake ažurnosti.

Programski kod: dobar programski kod je najbolji vid dokumentacije. Komentari omogućuju automatsko pravljenje ažurne dokumentacije.
Programski kod mora da se pravi, štaviše, to je cilj. Uvek ažurno predstavlja aktuelno stanje razvoja. Važno je da ima dobru strukturu, radi lakšeg razumevanja i radi lakšeg održavanja (refaktorisanje).
Testovi jedinica koda u agilnom razvoju su u obliku korisničkih celina (sasvim površne).

## 320 Šta su alati za unutrašnje dokumentovanje programskog koda? Zašto su potrebni i po čemu se suštinski razlikuju od održavanja spoljašnje dokumentacije?

## 321 Šta je Doxygen? Šta omogućava? Navesti primere anotacije koda.

Veoma široko usvojen alat za dokumentovanje programskog koda.
Praktično standard za programske jezike: $C++$, $C$, $Objective-C$, $C\#$, $PHP$, $Java$, $Python$, $IDL$,...

Pravi dokumentaciju u HTML-u za onlajn pregledanje.
Pravi dokumentaciju u LATEX, RTF, PostScript, PDF ili Unix man formatu.
Dokumentacija se pravi automatski na osnovu anotiranih izvornih fajlova. Može da se konfiguriše da napravi dokumentaciju na osnovu strukture koda čak i iz izvornih fajlova koji nisu posebno anotirani.
Automatsko ilustrovanje dokumentacije grafovima zavisnosti, dijagramima nasleđivanja i dijagramima saradnje.
Može da se koristi i za pravljenje obične dokumentacije.
Anotacija programskog koda se vrši navođenjem specifičnih oblika komentara, koje Doxygen izdvaja i od njih pravi dokumentaciju.

## 322 Šta je optimizacija softvera?

Optimizacija softvera je proces menjanja strukture i implementacije programa u cilju postizanja manjeg zauzeća resursa:

- procesorskog vremena
- radne memorije
- prostora u trajnom skladištu
- i drugo

Veoma često ušteda na jednom resursu podiže opterećenje drugog.
Optimizacija softvera je raspoređivanje opterećenja po resursima u skladu sa potrebama.

## 323 Koje su informacije neophodne za uspešnu optimizaciju?

Razumevanje problema i rešenja

- zadatak
- algoritmi
- implementacija

Razumevanje ograničenja

- poslovni zahtevi
- arhitektura računara
- arhitektura procesora

Poznavanje alata

- programski jezik
- asembler i mašinski jezik
- alati za merenje performansi

## 324 Objasniti "optimizaciju unapred". Dobre i loše strane?

Kontinualno staranje o performansama. Ako u toku razvoja znamo koje delove je potrebno optimizovati. Neophodno je pažljivo struktuiranje koda sa definisanjem ciljnih performansi svake od komponenti.

Potencijalni problemi:

- Da li smo sigurni da znamo koji su to delovi?
- Da li smo sigurni koje su kritične granice performansi?
- Da li smo sigurni da taj deo koda neće biti menjan ili izbačen?

## 325 Objasniti "optimizaciju unazad". Dobre i loše strane?

Nakon izvršenog razvoja mere se performanse, sagledavaju se problemi i planiraju optimizacije.

Potencijalni problemi:

- Da li upotrebljen algoritam uopšte može da se optimizuje?
- Da li implementirana arhitektura softvera može da se optimizuje?

## 326 Kakva je suštinska razlika između optimizacija unapred i unazad? Kada je bolje primeniti koju od njih?

Najčešće je mnogo bolje da se optimizuje unazad.
Optimizacija unapred značajno otežava održavanje softvera.
Dobar dizajn softvera omogućava relativno lako održavanje, pa čak i zamenjivanje algoritama ili nekih delova arhitekture. Na taj način se omogućava zadržavanje obe verzije koda (optimizovane i neoptimizovane), čime se omogućava lakša lokalizacija eventualnih bagova.

## 327 Koji osnovni problem proizvodi primena optimizacije u agilnom razvoju softvera? Kako se prevazilazi?

Prevremena optimizacija - optimizacija preduzeta pre nego što znamo šta je i koliko potrebno da se optimizuje.
Sukobljava se sa principom agilnog razvoja "*neće biti potrebno*".
"*Optimizuj kasnije*". je primena principa "*neće biti potrebno*" na problem optimizacije.
"*Neće biti potrebno*": "Implementiraj tek onda kada je potrebno, a ne kada predviđaš da će biti potrebno!"

## 328 Kako se dele tehnike optimizacije? Navesti nekoliko primera.

Dele se na opšte tehnike: tehnike koje mogu da se primene na sve ili bar veći broj različitih programskih jezika ili problema i na specifične tehnike: tehnike koje se mogu primeniti na manji broj programskih jezika.

## 329 Navesti bar 7 opštih tehnika optimizacije koda.

- Odbacivanje nepotrebne preciznosti
- Upotreba umetnutih funkcija i metoda
- Integracija petlji
- Izmeštanje invarijanti van petlje
- Razmotavanje petlji
- Tablice unapred izračunatih vrednosti
- Eliminacija grananja i petlji
- Zamenjivanje dinamičkog uslova statičkim
- Snižavanje složenosti operacije
- Snižavanje složenosti algoritma
- Pisanje zatvorenih funkcija
- Smanjiti broj argumenata funkcija
- Izbegavati globalne promenljive
- Redosled proveravanja uslova
- Izbor rešenja prema najčešćem slučaju
- Konkurentnost i distribuiranost

## 330 Objasniti tehnike optimizacije "odbacivanje nepotrebne preciznosti" i "tablice unapred izračunatih vrednosti".

Smanjivanje preciznosti u pokretnom zarezu i smanjivanje opsega celih brojeva može da smanji vreme izvršavanja i više od 50%, posebno ako je dužina podatka inicijalno veća od dužine procesorske reči ili širine magistrale podataka.

Ako se neke funkcije izračunavaju za ograničen broj različitih argumenata, umesto izračunavanja se mogu konsultovati tablice. U slučaju složenijih izračunavanja može doći i do dramatičnog ubrzavanja, po nekoliko puta.

## 331 Objasniti tehnike optimizacije "integracija petlji", "izmeštanje inavarijanti izvan petlje" i "razmotavanje petlji".

Smanjivanje broja petlji (integracija), umesto dve uzastopne petlje pravljenje jedne sa složenijim korakom (u slučaju jednostavnih koraka) može da doprinese i do 35%.

Sve ono što može, izračunava se van petlje

```
for( int i=0; i<limit(a,b,c); i++ )...
```

Smanjivanje broja ponavljanja uz povećanje složenosti koraka petlje može da doprinese i do 50%, ali može i da uspori izvršavanje u nekim slučajevima.

## 332 Objasniti tehnike optimizacije "smanjiti broj argumenata funkcije" i "izbegavati globalne promenljive".

Ako funkcija ima manji broj argumenata, veća je verovatnoća da će prevodilac za prenošenje argumenata upotrebiti registre umesto steka. Posebno je korisno ako se funkcija poziva često ili iz petlji. Vid ove optimizacije je i prenošenje podataka u ovkiru strukture koja se prenosi po adresi.

Lokalne primenljive mogu da se optimizuju zapisivanjem u registrima, globalne ne smeju, jer se nikada ne zna da li ih još neko koristi (potprogram, druge niti i sl.).

## 333 Objasniti tehnike optimizacije "upotreba umetnutih funkcija" i "eliminacija grananja i petlji".

Uotreba umetnutih funkcija u slučaju jednostavnih funkcija (npr. swap) može da smanji vreme izvršavanja i do 50%.

U nekim slučajevima se grananja ili petlje mogu zameniti nešto složenijim izračunavanjem bez grananja i petlji. Može da se dobije i po nekoliko puta efikacniji kod.
Primer: brojanje bitova u 32-bitnom broju:

```
const short tablica[] = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
short brojBitova(int x)
{
return tablica[(x) & 0xF] +
   tablica[(x >> 4) & 0xF] +
   tablica[(x >> 8) & 0xF] +
   tablica[(x >> 12) & 0xF] +
   tablica[(x >> 16) & 0xF] +
   tablica[(x >> 20) & 0xF] +
   tablica[(x >> 24) & 0xF] +
   tablica[(x >> 28)];
}
```

Skraćivanje vremena izvršavanje je oko 90% u odnosu na pojedinačno brojanje.

## 334 Objasniti tehnike optimizacije "zamenjivanje dinamičkog uslova statičkim" i "snižavanje složenosti operacije".

Zamenjivanje dinamičkog uslova statičkim - ako opseg broja ponavljanja nije fiksan, nekada može biti efikasnije uraditi posao za pun obim nego proveravati granice. Kao naredni korak može da se primeni i eliminacija petlje.

Neke operacije mogu da se zamene efikasnijim operacijama na primer umesto $a * 8$ možemo da napišemo $a << 3$

## 335 Objasniti tehnike optimizacije "redosled proveravanja uslova" i "izbor rešenja prema najčešćem slučaju".

Ako se proverava veći broj uslova i postupa u skladi sa više invarijanti, redosled proveravanja uslova se mora odrediti tako da prosečan broj provera bude što manji.

Neki algoritmi su efikasniji za neke slučajeve a manje efikasni za druge. Izbor algoritma mora da se obavlja u skladu sa očekivanim slučajevima upotrebe.
Primer: za kratke nizove primitivan algoritam sortiranja *bubblesort* može da bude efikasniji od algoritma *quick-sort*

## 336 Koje su najčešće greške pri optimizaciji? Objasniti.

Ništa ne pretpostavljati. Često se pogrešno pretpostavlja da je "A efikasnije nego B". Svaka pretpostavka mora da se proveri.
Primer besmislene optimizacije:
Umesto da napišemo $a = b * 40$;
Možemo da napišemo optimizovano $a = (b << 5) + (b << 3)$;
Motiv: pomeranje je brže nego množenje. Međutim, većina prevodilaca to može da uradi umesto nas. Štaviše, to neće uraditi ako je procesor takav da to nije brže.

Smanjivanje koda nije uvek i njegovo optimizovanje. Dobar primer je upravo eliminacija petlji.

Optimizovanje tokom inicijalnog kodiranja je najčešći problem. Prvo treba napisati ispravan kod (sa testovima, naravno) pa tek onda optimizovati.

Posvećivanje više pažnje performansama nego korektnosti. Nekada performanse jesu primaran cilj ali najčešće su korektnost i preciznost mnogo važniji.

## 337 Navesti i ukratko objasniti tri tehnike optimizacije specifične za programski jezik C++.

Koristiti standardnu biblioteku. Teško je nešto uraditi efikasnije nego u standardnoj biblioteci. čak i kada u tome uspemo, to će verovatno već sledećom verzijom biblioteke biti prevaziđeno.

Upotrebljavati reference umesto pokazivača - jednako efikasno, a mnogo čistije.

Odložena inicijalizacija objekata. Za neke operacije nam možda i nisu neophodni kompletno inicijalizovani objekti. Umesto da se u konstrukciji izvede puna inicijalizacija, možda je dovoljno da se izvede samo priprema za inicijalizaciju, a da se ostalo uradi pri prvoj upotrebi nekog od složenijih metoda.

Iz delova koda koji se optimizuju izbaciti rukovanje izuzecima - obrada izuzetaka je relativno neefikasna.

## 338 Šta su "optimizacije u hodu"? Navesti primere.

Neke optimizacije mogu da se prave u hodu. Ali veoma oprezno.

- Prenošenje objekata po referenci (osim, eventualno, u slučaju sasvim malih objekata)
- Deklarisanje privremenih promenljivih što dublje u kodi (da se ne prave ako nije potrebno)
- Koristiti konstrukciju objekata (inicijalizaciju) pre nego dodeljivanje
- Koristiti liste incijilazicija članova i baznih objekata
- Implementirati sopstvene operatore alikacije i dealokacije (new i delete)
- Upotreba šablona funkcija i metaprogramiranja

## 339 Šta su profajleri? Čemu služe? Šta pružaju programerima?

Profajleri su alati za podršku optimizovanju. Za svaki potprogram (ili čak blok ili naredbu programa) računaju ukupno vreme izvršavanja, broj izvršavanja, vreme provedeno u pozivima i vremeno provedeno u samom kodu.
Mogu da mere i zauzeće memorije, upotrebu keš memorije, razne druge stvari.

GNU gprof
Valgrind