[RS] Skripta: Pitanja i odgovori

Momir Adžemović

1. april 2020.

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 tj. p2 je običan ceo broj.

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

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

$sizeof(s1) == 4$
$sizeof(s2) == 3$

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, šabloni.

5 Koji je redosled uništavanja objekata u primeru:

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

Napomena! Tip *int* je primitivni tip, ne objekat. Ako bi promenljiva a predstavljala objekat, onda bi redosled uništavanja bio sledeći: Prvo će 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];
```

$short * *p = newshort[900]$; će da vrati grešku pri kompiliranju, jer se ne poklapaju tipovi, p je $short**$ i pokušavamo da mu dodelimo nešto što je $short*$. **"error: cannot convert 'short int*' to 'short int**' in initialization"**

7 Koje operacije klase Lista se izvršavaju u sledećoj naredbi:

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

1. alocira se memorija za objekat Lista;
2. poziva se eksplicitan konstruktor za objekat Lista;

- Zašto se ne poziva operator dodele?
- Zašto se ne poziva konstruktor kopije? [1] [2]
- **the rule of three** [3]

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;
}
```

Tri puta. Pri deklaracija promenljivih a, b i p. Promenljive c i q su reference na objekte.

9 Šta su šabloni funkcija?

Šabloni funkcija predstavljaju specijalne funkcije koje rade sa generičkim tipovima. Ukoliko nam je potrebno više verzija iste funkcije samo sa različitim tipom podataka ili klasa, onda šablon funkcija može da zameni tu grupu funkcija. Takođe konstante mogu da budu generički parametri. U suštini šabloni nam omogućavaju generičko programiranje. Šabloni predstavljaju tip statičkog polimorfizma tj. polimorfizam koji se obrađuje tokom kompilacije.[4]

10 Kako se pišu i koriste šabloni funkcija?

Primer:

```
template<typename T>
T sum(T a, T b)
{
  return a + b;
}
```

Napomena: Ako napišemo šablon f-je i ne koristimo u programu nijednom konkretnu f-ju onda će nam kompajler dati isti izvršni kod kao da nismo pisali šablon. Kompajler prevodi sve konkretizovane funkcije nekog šablona. Isto važi i za klase.
Napomena: Ako definišemo konkretnu funkciju (npr. int sum(int a, int b)), onda on ima prioritet u odnosu na šablon i ona će biti pozvana.

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?

Primer:

```
template<typename A, typename B>
struct Pair{
  A first;
  B second;

  Pair(A f, B s)
    : first(f), second(s)
    {}
}
```

Isto kao i šabloni funkcija, samo se odnosi na generičke klase. Primeri generičkih klasu su mape, vektori, skupovi, ..., koji su već deo C++-a.

Primer: 'vector<int>', možemo posmatrati kao funkciju koja prima kao argument celobrojni tip i vraća vektor celih brojeva (konkretan vektor).

13 Kako se prevode šabloni klasa?

Svi generički tipovi se zamene sa konkretnim tipovima. Primer: Za prethodno pitanje 'vector<int>' dobijamo tako što zamenimo sva pojavljivanja generičkog tipa 'T' sa konkretnim tipom 'int'. Dobijeni kod kompilator može prevesti na klasičan način. Bitno je naglasiti da je ovaj proces statičan.

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

Napomena: Mora da važi svaki neophodan uslov i mora da važi barem jedan dovoljan uslov!
Neophodan uslov: Metod mora biti virtuelan. Metod je virtuelan ako ima ključnu reč 'virtual'.
Dovoljan uslov: Objektu pristupimo preko reference ili preko pokazivača. Primer:

```
object o1 = Object();
o1.method(); // nije dinamicki

object *o2 = new Object();
o2->method() // jeste dinamicki, ako je virtualan metod

object &o3 = o1;
o3.mehtod() // jeste dinamicki ako je virtualan metod
```

15 Šta je virtualna funkcija?

Virtuelnce funkcije (virtuelne metode) su metode neke klase koje uz sebe imaju ključnu reč „virtual“. U C++-u podrazumevano povezivanje je statičko. Ako želimo da koristimo dinamički polimorfizam, potrebno je naglasiti to ključnom reči „virtual“ i pozivati taj metod preko pokazivača ili reference na objekat.

U Java-i su sve nestatičke metode implicitno virtuelne što donosi neke svoje olakšice. Mana toga je efikasnost.

16 Šta je čisto virtuelna funkcija?

U C++-u čisto virtuelna funkcija i apstraktna funkcija predstavljaju sinonime. Čisto virtuelna funkcija je funkcija koja nema svoju implementaciju u okviru klase. Klasa koja sadrži čisto virtuelnu funkciju je apstrakna klasa. Čisto virtuelnu metodu označavamo sa '= 0' kao sufiks. Primer:

```
class VirtualObject{
  public:
  VirtualObject() {}
  virtual void doSomething() = 0;
}
```

17 Šta je apstraktna klasa?

Apstraktna klasa je klasa koja ima barem jednu čisto virtuelnu metodu. Apstraktna klasa ne može da ima svoju instancu, ali možemo imati pokazivač ili referencu na objekat apstrakne klase što nam omogućava da koristimo dinamički polimorfizam. Apstraktna klasa i dalje može da ima svoj konstruktor koji se poziva samo u okviru konstruktora klasa koje je nasleđuju.

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

Umetnute „inline“ funkcije su funkcije koje imaju prefiks „inline“. Predstavlja sugestiju kompajleru da 'proširi' funkciju na mestu gde je pozvana. Ideja je da se zaobiđe postavljanje funkcije na 'stackframe' i da se povećaju performanse.

Ovo predstavlja samo sugestiju. Kompajler odlučuje da će li funkcija zapravo biti proširena. Funkcije koje nemaju „inline“ takođe mogu biti proširene.

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

Isto kao „inline“ funkcije.

20 Koja su osnovna merila neuspeha pri razvoju softvera? Objasniti svaku ukratko.

Osnovno merilo neuspeha je izgubljena materijalna vrednost tj. plaćena cena neuspeha. Ovo se može reći i za projekte koji nisu nužno vezani za softver.
Merila neuspeha:

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

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

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

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

Sistem je uspešno implementiran po zadatim zahtevima, ali nije upotrebljiv ili se ne isplati koristiti ga.
Aspekti neupotrebljivosti:

- **Neupotrebljiv korisnički interfejs:**
  - Loše rešeni ergonomski uslovi (odnosi se na komfort i efikasnost);
  - 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 sistem ne dopušta da se cena manuelno postavi na 0, takode neće biti moguće.
  - Pretraživanje knjiga (ili slicnog kataloga) koje podrazumeva unosenje tacnog naslova.
  - Ako IS omogućava upravi da tacno vidi i kritikuje magacionare zbog neracionalnog zauzeca prostora, a ne omogućava automatsku podrsku u vidu saveta, magacioneri pocinju da trose nesrazmerno mnogo vremena za planiranje prostora. Iako rezultat jeste mnogo bolja iskoriscenost prostora, gubi se na vecem utrošku radnog vremena.

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

- Nerealni ili nejasni ciljevi projekta.
- Neprecizna procena potrebnih resursa.
- Loše definisani zahtevi: Komumanikacija sa klijentom u prvoj fazi je veoma bitna gde se razjašnjavaju svi detalji i postavljaju se sva neophodna pitanja.
- Slaba komunikacija između klijenta, razvijaoca 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 razvijaoca;
- Višestrani problemi.

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

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

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

- 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 cvrstih principa i pravila.

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

- 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 planiranje i preterano planiranje.

Nedovoljno planiranje se obično ogleda kroz nedovoljnu analizu problema i/ili loše definisane zahteve i/ili neprecizno definisane zahteva. Najčešće posledice nedovoljnog planiranja su nesrazmerno veliki broj naknadnih korekcija zahteva, slaba upotrebljivost rešenja ili probijanje rokova.

Preterano planiranje se obično ogleda kroz preširoko i nekoncentrisano ulaženje u projekat (suviše duboka i obimna analiza sa ranim detaljima) i/ili preširoko ulaženje u implementaciju i/ili preambicioznost koja je nesrazmerna realnim mogućnostima. Najčešće posledice su kasno uočavanje propusta, otežana tranzicija i 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

Ovo važi i za razvoj projekta koji nije nužno vezan za softver. Najveći značaj ima dobro upravljenje rizicima u početnim fazama. Ovaj proces je otežan u slučaju dugačkih razvojnih ciklusa.

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

1. manjak osoblja
2. nerealni rokovi i budzet
3. razvoj pogrešnih funkcija
4. razvoj pogrešnog interfejsa
5. preterivanje („pozlaćivanje“)
6. neprekidni niz izmena u zahtevima
7. slabosti eksterno realizovanih poslova
8. slabosti u eksterno nabavljenim komponentama
9. slabe performanse u realnom radu
10. 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 žizi su objekti a ne procesi
- Pravljenje prototipova

32 Objasniti princip inkrementalnog razvoja.

Umesto da se razvoj posmatra kao jedna velika celina, sa složenim projektom i implementacijom, ta celina se deli u niz inkrementalnih faza. Te faze se nazivaju **inkrementi ili ciklusi**. Svaki inkrement ima sve osnovne podfaze, kao što su: analiza zahteva, dizajn, implementacija i testiranje. [5]
Dobit:

- Svaka faza je manja i jednostavnija, čime je pojednostavljeno planiranje i implementacija.
- Veća tačnost planiranja troškova i rokova i tačnije izveštavanje o toku razvoja.
- Klijent može da odgovori na svaki napredak.

Rizik:

- Ako u prvim koracima zanemarimo predstojeće korake, onda postoji rizik da će u narednim koracima biti potrebne veće izmene.
- Ako se u prvim koracima uzmu u obzir svi prestojeći, onda postoji rizik da se njihova složenost približi složenosti čitavog sitema („naduvavanje koraka“), čime ovakav pristup gubi smisao.
- Često nije moguće sagledati unapred broj, cenu i ukupno trajanje svih koraka.

33 Objasniti princip određivanja koraka prema rokovima.

Za svaki inkrementalan korak se prvo odrede budžet i rokovi, a tek posle toga poslovi koji će tim korakom biti obuhvaćeni.
Dobit:

- Drastično smanjivanje rizika od prekoračenja budžeta ili rokova;
- Uspostavljanje ritma redovnog isporučivanja novih verzija sistema;

- Redovnija kontrola kvaliteta i viši stepen poverenja između subjekta;
- Postepeno prilagođavanje korisnika novim elemetnima sistema.

Mane:

- 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 velika stavka u slučaju velikog broja malih koraka;
- U prvim koracima se obično biraju poslovi koji donose vecu dobit;
- Pri kraju razvoja postoji rizik da se ne implementiraju poslovi „čija je cena veca od dobiti“ iako su značajni za sistem kao celinu.

34 Objasniti princip pojačane komunikacije među subjektima.

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

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

Mane:

- 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 dode do „konačnog“ rešenja.

35 Objasniti princip davanja prednosti objektima u odnosu na procese.

U središte pažnje se stavljaju objekti pre procesa.
Dobit:

- Objekti su obično stabilniji od procesa (manje se menjaju i manje su šanse da se skroz obrišu);
- Prilagođeniju su inkrementalnom pristupu;

- Smanjuje se rizik od pogrešnih odluka u ranijim fazama razvoja (u ranijim fazama je više akcenat na objektima, a na kasnijim na procesima).

Rizik:

- Potpuno zanemarivanje procesa u ranim fazama može voditi ka pogrešnoj arhitekturi sistema što se kasnije teško menja;
- Sasvim detaljno razmatranje procesa u ranim koracima može da dovede do „naduvavanja“ koraka.

36 Objasniti princip pravljenja prototipova

U okviru analize problema se pravi prototip koji održava način funkcionisanja i upotrebe softvera.
Dobit:

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

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 napravljen 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?

1. Postojanje metodologija koje strukturirano pristupaju analizi i opisivanju procesa;
2. Potreba za strukturiranim opisivanjem podataka;
3. Potreba za opisivanjem entiteta koji menjaju stanja;
4. Podizanje nivoa apstraktnosti posmatranja elemenata sistema;
5. Programiranje upravljano događajima;
6. Vizuelni korisnički interfejsi;
7. Povećana modularnost softvera;
8. Skraćivanje razvojnog ciklusa;

9. Tranzicija modela;
10. Višestruka upotrebljivost softvera;
11. i drugo ...

Za razliku od strukturnih, koje su u središte pažnje stavljale uređivanje procesa i algoritama, objektno orijentisane metodologije u središte pažnje stavljaju uređivanje objekata kojima se opisuje sistem. Razvoj objektno orijentisanih 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 se stavljaju objekti, a ne procesi;
- Sve objektno orijentisane metodologije se odlikuju skraćivanjem trajanja razvojnog ciklusa:
  - RUP (i druge) propisuju inkrementalni razvoj,
  - Agilne metodologije propisuju određivanje koraka prema rokovima i troškovima;
- Pojačana komunikacija među subjektima u svim fazama.

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

„Objekti imaju stanje, ponašanje i identitet“ (Booch, 1994)

Primer:

```
class Point {
private:
  int x, int y;
public:
  Point(int _x, int _y)
    : x(_x), y(_y)
    {}

  void move(int dx, int dy)
  {
    x += dx;
    y += dy;
  }
};

int main()
{
  Point point(2, 3);
  point.move(1, 4);
  return 0;
}
```

Promenljiva „point“ predstavlja objekat klase „Point“. Njeno stanje je određeno atributima (poljima) „x“ i „y“. U ovom slučaju je stanje (x=2, y=3). Njeno ponašanje je određeno metodom „move()“ tj. tačka može da se pomera. Njen identitet je „Point“.

40 Šta je klasa? Atribut? Metod?

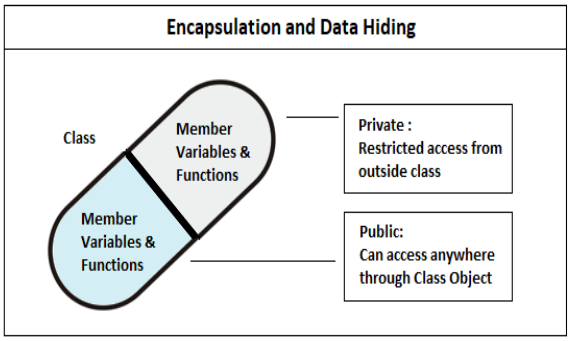
- **Klasa** predstavlja skup objekata koji imaju isto ponašanje (isti skup metoda) i isti skup atributa.
- **Objekat** predstavlja konkretan primerak klase. Skup metoda opisuje ponašanje objekta.
- **Metodi** su obično funkcije definisane u okviru neke klase. Atributi su polja klase i preko njih tj. njihovih vrednosti se opisuje trenutno stanje objekta.

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

- enkapsulacija
- interfejs
- polimorfizam
- nasleđivanje: specijalizacija i generalizacija u hijearhiji klase. Smanjuje dupliran kod i koristi se za dinamički polimorfizam.

42 Objasniti koncept enkapsulacije.

Ideja je da se podaci i funkcije koje rade sa tim podacima spoje u jednu klasu. Klasa se deli na javni i privatni deo. Podaci se uglavnom nalaze u privatnom delu, sa izuzecima (javne statičke konstante su relativno prihvatljive). Javne funkcije opisuju klasu i njenu svrhu u okviru modula u kojem se koristi.



Svrha je apstrahovanje strukture metodima i viši nivo međusobne nezavisnosti klasa od modula u kojem se upotrebljava.

43 Objasniti koncept interfejsa.

Klasa je opisana korisniku preko javnih metodima klase (plavi, javni deo kapsule). Korisnik interfejsa ne mora da zna implementaciju klase da bi je koristio.
Svrha je sakrivanje složenosti implementacije. Suština objekta je u njegovom ponašanju.
Napomena: Ako objekat ima više interfejsa, znači da ima više funkcija, pa je potrebno razmotriti njegovo razlaganje na više objekata.

44 Objasniti koncept polimorfizma.

Poenta polimorfizma je da se jedan kod što više puta iskoristi.
Vrste:

- hijearhijski (dinamički): dobijen kombinacijom dinamičkog povezivanja i nasleđivanja;
- parametarski (statički): šabloni;
- implicitni: retki jezici ovo podržavaju.

Pišemo apstrakniji kod koji ima veću upotrebljivost. [6]

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

Predstavlja osnovu za građenje hijerarhija klasa.
Svrha:

- Nasleđivanje se koristi za eksplicitno označavanje sličnosti među klasama (objektima).
- Predstavlja osnovu za hijerarhijski polimor zam: 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 opstiji (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?

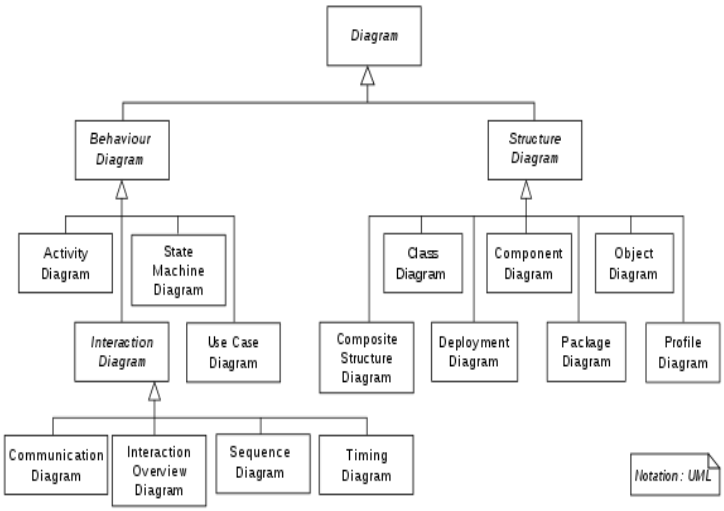
- Nekoliko dubljih metodologija koncetrisanih na različite faze razvoja
- Pokušaji ujednačavanja notacije
- Akcenat na notaciji

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?

UML je **Objedinjeni jezik za modeliranje (Unified Modeling Language)**. Oblikovan je paralelno uz metodologiju „Unified Approach“ (Objedinjeni pristup). Pretežno grafički jezik (minimalno se koristi tekst van grafičkog prikaza). Olakšava vizuelizaciju kompleksnog sistema i predstavlja tip dokumentacije. [7]



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

Dijagrami se dele u tri grupe:

- dijagrami ponašanja
- dijagrami interakcije (može se gledati i kao podgrupa dijagram ponašanja)
- strukturni dijagrami

52 Navesti strukturne dijagrame UML-a.

1. Dijagram klasa (class diagram)
2. Dijagram komponenti (component diagram)
3. Dijagram objekta (object diagram)
4. Dijagram profila (profile diagram)
5. Dijagram složene strukture (composite structure diagram)
6. Dijagram isporučivanja (deployment diagram)
7. Dijagram paketa (package diagram)

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

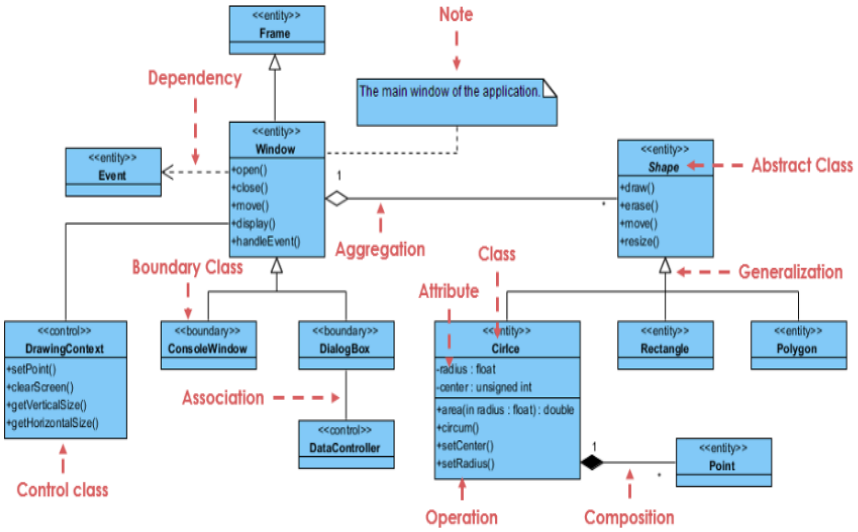
Ilustruje elemente statičkog modela tj. klase, njihov sadržaj i međusobne odnose. Klasa je predstavljena sa:

- nazivom
- atributima (ime, tip, vidljivost)
- metodama (ime, parametri, povratna vrednot, vidljivost)

Odnosi mogu biti:

- nasleđivanje (specijalizacija ili generalizacija)
- asocijacija:
  - agregacija
  - kompozicija
- zavisnost (Neka je A zavisna od B. Ako se klasa B menja, onda klasa A može zahtevati promene, ali ne važi obrnuto)

Kako razlikovati asocijaciju, agregaciju i kompoziciju?
Agregacija i kompozicija su specijalni slučajevi asocijacije. Agregacija predstavlja odnos gde dete može da postoji bez roditelja (klase su čvorovi), a kompozicija predstavlja odnos gde dete ne može da postoji bez roditelja.

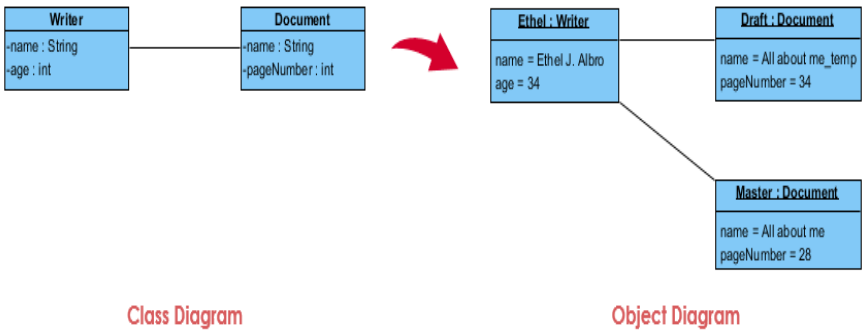


54 Koja je uloga i šta su osnovni elementi dijagrama komponenti?

Dijagram komponenti razbija sistem koji se razvija na neke celine po njihovim funkcionalnostima. Ima dosta sličnosti kao dijagram klase po odnosima elemenata. Ovaj dijagram možemo posmatrati kao apstraktniji nivo dijagrama klase. Svaki element ima naziv i interfejs.

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

Ilustruje elemente dinamičkog modela tj. predstavlja objekte u jednom trenutku (konkretna situacija). Koristi se kao dopuna dijagrama klase i komunikacije. Sastoji se od objekata (naziv klase i opis stanja) i njihovih odnosa.



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

Predstavlja elemente fizičke arhitekture sistema. Ovo je nešto što bismo koristili ako imamo distribuirani sistem gde imamo više servera. Koriste se za planiranje arhitekture sistema.
Sadrži:

- čvorove (primer: server)
- softverske ili hardverske podsisteme
- međusobne veze tih podsistema
- može da ilustruje zasupljenost 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. Paket je često sinonim za prostor imena. Okuplja elemente koji su semantički povezani i očekuje se da se zajedno menjaju.
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.

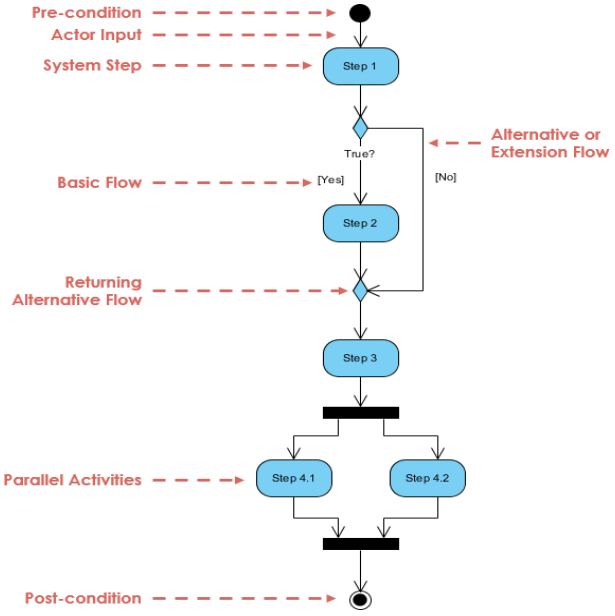
1. Dijagram aktivnosti (activity diagram)
2. Dijagram stanja (state machine diagram)
3. Dijagram slučajeva upotrebe (use case diagram) - u ove dijagrame se mogu ubrojati i dijagrami interakcije

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

Predstavlja poslovne procese višeg nivoa, tokove podataka i eventualno složene logičke elemente sistema. Veoma sličan dijagramu za opisivanje algoritama.
Sadrži:

- procese

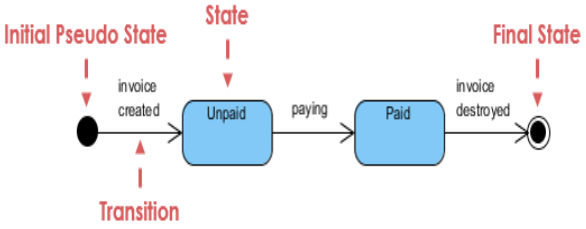
- tokove podataka
- čvorove i grananja
- uslovne tačke
- početne i završne tacke
- može da sadrži i „linije autora“



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

Ovaj dijagram se koristi da se opišu ponašanja koja zavise od trenutnog stanja. Ovde smatramo da objekat reaguje drugačije na istu situaciju u zavisnosti od trenutnog stanja.
Sadrži:

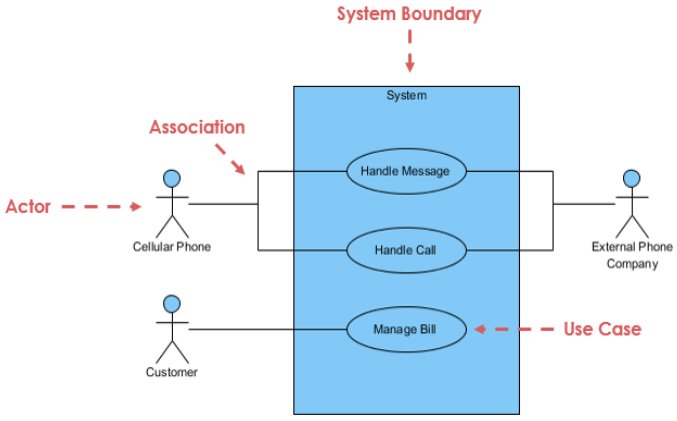
- sva moguća stanja objekta
- posebno označeno početno i završno stanje
- prelaske između stanja (događaji)
- strelica od prethodnog prema narednom stanju
- nazivi događaja koji menjaju stanje objekata
- odgovarajuća objašnjenja



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

Predstavlja slučajeve upotrebe, aktere i njihove međusobne odnose. Ne prikazuje redosled po kojem se koraci izvršavaju.
Sadrži:

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



62 Navesti dijagrame interakcija.

1. Dijagram komunikacije (communication diagram) - raniji naziv, Dijagram saradnje (collaboration diagram)

2. Dijagram interakcija (interaction diagram ili interaction overview diagram)
3. Dijagram sekvence (sequence diagram)
4. Dijagram vremena (timing diagram)

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

Predstavlja produženje dijagrama objekata gde su predstavljene interakcije između objekata. Dijagram prikazuje objekte uz poruke koje oni šalju. Sadrži objekte, poruke i evetualno komentare.

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

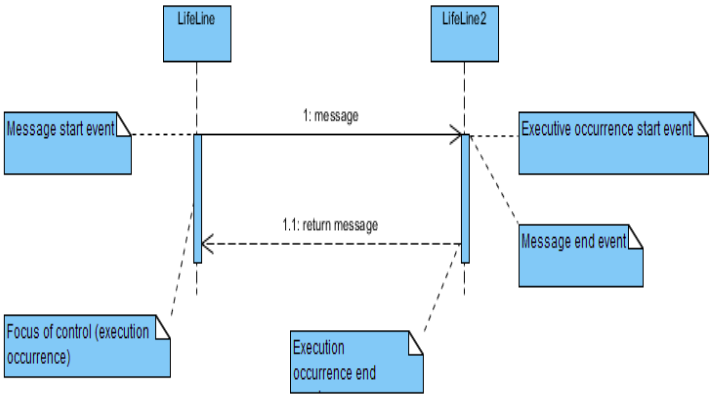
Predstavlja varijantu dijagrama aktivnosti gde su čvorovi interakcije ili aktivnost.
Sadrži:

- 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?

Prikazuju interakcije objekata u kontekstu saradnje. Ovi dijagrami imaju akcenat na vremenu i prikazuju redosled interakcija.
Sadrži:

- aktere
- linije života (vreme)
- poruke poziva (započinu drugu liniju života)
- povratna poruka
- ostali tipovi poruka



66 Šta je uzorak za projektovanje? Čemu služi?

„Svaki uzorak za projektovanje opisuje problem koji se stalno ponavlja u 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.“
— Cristopher Alexander

Kristofer je ovde govorio za uzorke za zidanje zgrada, ali to što je rekao isto važi i za uzorke objektno orijentisanog projektovanja. Softverska rešenja su izražena pomoću objeta i interfejsa umesto preko zidova i vrata, ali suština obe vrste uzoraka je rešavanje problema u svom kontektsu.

Uzorci za projektovanje su rešenja za opšte probleme sa kojim se razvijaoci često susreću. Ova rešenja su sakupljenja preko iskustava dobijenim na prethodnim lošim rešenjima koja su se javljala proteklih godina.

Gang of four: Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides su objavili knjigu „Design Patterns: Elements of Reusable Object-Oriented Software“, što je započelo koncepte o uzorcima projektovanja.

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 problem, 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 sledeča 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.
- **Primenljivost:**
  - Na koje situacije se uzorak može primeniti?
  - 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 korisćenja 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 (primer. Fabrika)
- Strukturni (primer. Dekorator)
- Uzorci ponašanja (primer. Posetilac)

74 Objasniti namenu gradivnih uzoraka za projektovanje.

Apstrahuju proces pravljenja objekata. Pomoću njih sistem postaje nezavisan od načina pravljenja objekata. Postoje dva domena: **Uzorak za pravljenje sa domenom klase**, koji koristi nasleđivanje za menjanje klase koja se instancira, i **Uzorak za pravljenje sa domenom objekta**, koji delegira pravljenje nekom drugom objektu. U ovim uzorcima postoje dve teme koje se stalno ponavljaju:

1. Svi uzorci enkapsuliraju znanje o tome koje konkretne klase sistem koristi;
2. Kriju kako se prave primerci ove klase i kako se sastavljaju.

U celom sistemu o objektima se samo zna interfejs, na osnovu toga kako su definisani u apstraktnim klasama. Omogućavaju veliku fleksibilnost u smislu:

- Šta se pravi?
- Ko se pravi?
- Kako se pravi?
- Kada se pravi?

Konstrukcija može biti statična ili dinamična.

Suština: Daju nam mogućnost da kreiramo objekte, a da u isto vreme sakrijemo logiku, što omogućava veću fleksibilnost prilikom određivanja koji objekat treba da se kreira (program odlučuje o tome).

75 Navesti bar četiri gradivna uzorka za projektovanje.

1. Apstraktna fabrika (Abstract Factory)
2. Graditelj(Builder)

3. Proizvodni metod (Factory Method)
4. Prototip (Prototype)
5. Unikat (Singleton)

Uzorci za projektovanje koji nisu objašnjeni u okviru narednih pitanja, a opet mogu potencijalno da dođu na ispit:

- **Builder:** Ukoliko je potrebno da se pravi više kompleksnih objekata sa sličnom strukturom, onda je **Graditelj** dobro rešenje. Primer: Konstrukcija kuća koje mogu, a ne moraju da imaju neke elemente kao što su garaža, bazen, kamin, ... Jedno rešenje je da se koristi komplikovani konstruktor sa parametrima koji govore koje elemente treba kuća da sadrži. Alternativa je da se konstrukcija apstrahuje u zasebnu klasu koja ima metode tipa *buildWalls(), buildDoors(), buildWindows, buildPool(), ...*
- **Prototype:** Ovaj uzorak za projektovanje odgovara interfejsu **Cloneable** u Java-i. Svaka klasa može da nasledi klasu (C++) ili implementira interfejs (Java) sa jednostavnom metodom *clone()* koja kopira objekat. Motivacija: Samo sam objekat zna kojeg je tipa (primer: polimofrizam) i vrednosti svojih privatnih polja.

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 sastavljenje interfejsa ili implementacija. Strukturni uzorci sa domenom objekata opisuje načine za postizanje nove funkcionalnosti kombinovanjem objekata. Fleksibilnost sastavljenje 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.

1. Adapter (Adapter)
2. Most (Bridge)
3. Sastav (Composite)
4. Dekorater (Decorator)
5. Fasada (Facade)
6. Muva (Flyweight)
7. Proksi (Proxy)

Uzorci za projektovanje koji nisu objašnjeni u okviru narednih pitanja, a opet mogu potencijalno da dođu na ispit:

- **Adapter:** Predstavlja specijalan objekat koji se koristi za komunikaciju dve klase (biblioteke) koje imaju nekompatiblne interfejse, gde je njegov zadatak da omogući njihovu komunikaciju. Primer: Napravljena je biblioteka koja predstavlja Scraper sa određene vrste *XML* fajlova. Pored te biblioteke postoji odlična biblioteka za vizuelizaciju statistika nad podacima, ali problem je što ona radi sa *JSON* podacima. Rešenje je da se koristi Adapter objekat koji prevodi *XML* u *JSON* podatke.

- **Bridge:** Predstavlja uzorak za projektovanje koji nam omogućava da razdvojimo veliku klasu ili grupu bliskih klasa na dve odvojene hijearhije koje mogu odvojeno da se razvijaju. Primer: Apstraktnu klasu Predmet nasleđuju klase CrvenaLopta, PlavaLopta, CrvenaKutija, PlavaKutija. Ovde ima smisla napraviti zasebnu klasu Boja koju nasleđuju klase Plava i Crvena, a da u hijearhiji klase Predmet ostanu samo klase Lopta i Kutija. Klasa Predmet sada ima posebno polje tipa Boja. Sada hijearhije Predmet i Boja mogu nezavisno da se razvijaju.

- **Facade:** Predstavlja uzorak za projektovanje koji olakšava održavanje interfejsa kompleksnog sistema.

- **Flyweight:** Predstavlja uzorak za projektovanje koji omogućava da se više objekata smesti u radnu memoriju tako što deli zajedničke delove objekata. Primer: Ukoliko se prezentuje više identičnih slika na različitim pozicijama, dovoljno je da se pamti slika samo na jednom mestu.

- **Proxy:** Ovaj uzorak za projektovanje se koristi kada se radi sa skupom objekata koji su skupi, a potencijalno se koriste. To znači da ima smisla da se implementira lenja inicijalizacija tj. da se objekat zapravo kreira tek kad se on zahteva. Ovo može da izazove dupliranje koda. Rešenje je koristiti **Proksi** uzorak za projektovanje koji nudi „prazan“ objekat sa istim interfejsom. Kada je potrebno da se zapravo koristi pravi projekat, proksi kreira taj objekat.

78 Objasniti namenu uzoraka ponašanja.

Bave se algoritmima i raspodelama odgovornosti. Ne opisuju samo uzorke objekata ili klasa već i uzorke njihove međusobne komunikacije (observer). 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**, koji koriste nasleđivanje za distribuiranje ponašanja (šablonski metod, interpretator) i **objektne uzorke ponašanja**, koji koriste sastavljanje objekata (umesto nasleđivanja).

79 Navesti bar sedam uzoraka ponašanja.

1. Lanac odgovornosti (Chain of Responsibility)
2. Komanda (Command)

3. Interpretator (Interpreter)
4. Iterator (Iterator)
5. Posrednik (Mediator)
6. Podsetnik (Memento)
7. Posmatrač (Observer)
8. Stanje (State)
9. Strategija (Strategy)
10. Šablonski metod (Template Method)
11. Posetilac (Visitor)

Uzorci za projektovanje koji nisu objašnjeni u okviru narednih pitanja, a opet mogu potencijalno da dođu na ispit:

- **Chain of Responsibility:** Koristi niz *Handler*-a, gde svaki od njih ima opciju da obradi zahtev koji stiže od klijenta ili prosledi zahtev sledećem u nizu.
- **Command:** Predstavlja uzorak za projektovanje koji preslikava zahtev u zaseban objekat. Ovo nam omogućava parametrizaciju metoda za različite zahteve i postavljanje zahteva na čekanje. Primer: Imamo aplikaciju nalik na *Notepad* i sa različim načinima za čuvanje teksta na *clipboard*: *shortcut*, padajući meni ili dugme. Kopiranje koda je loša praksa. Bolja ideja je da svaka opcija kreira *Command* objekat sa svojim parametrima i da objekat izvrši svoju funkciju, što je u ovom slučaju kopiranje.
- **Iterator:** Iterator nam omogućava da prolazimo kroz kolekcije podataka (primer: lista, niz, binarno drvo, ...), a da se pritom ne vodi računa o internoj implementaciji te kolekcije. Ovo nam omogućava da jednostavno promenimo klasu kolekcije, a da ne menjamo dalje kod (primer: lista u niz). Iterator se definiše kao interfejs. U slučaju Java-e, klasa mora da implementira *iterator* interfejs.
- **Mediator:** Posrednik se koristi da se reše odgovarajući problemi koji su nastali kao posledica velike spregnutostni kompenenti. Primenom ovog uzorka dobijamo ograničenu komunikaciju između objekata tj. komunikacija se vrši preko posrednika.
- **Memento:** Predstavlja uzorak za projektovanje koji omogućava da se objekat vraća u prethodno stanje bez otkrivanja njegove implementacije.
- **State:** Predstavlja uzorak projektovanja koji koristi slične koncepte kao konačni automat. Klasa izvršava operacije i zavisnosti od stanja (kao da se sama klasa menja).
- **Template Method:** Predstavlja uzorak za projektovanje gde se u baznoj klasi definiše skelet za algoritam, a u klasama koje nasleđuju baznu klasu se redefinišu odgovarajući koraci bez menjanja strukture algoritma. Primer: klasa *Reader* u Java-i.

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

Kreiranje objekta se vrši bez otkrivanja implementacije korisniku. Za korisnika je dovoljno da zna interfejs za kreiranje, a klasa prepušta instanciranje potklasi.

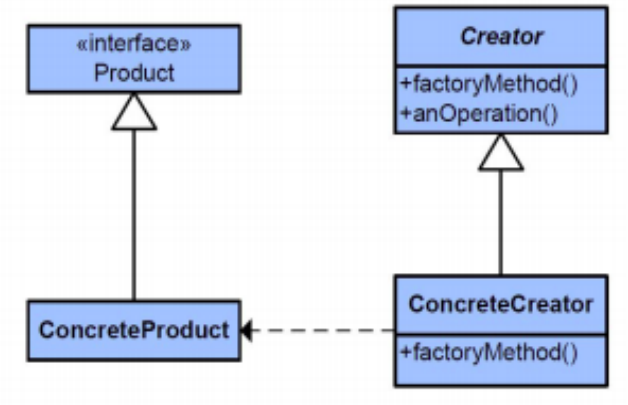
Koristi se kada:

- Klasa ne može da predvidi klasu objekta koji mora da stvori.
- Klasa želi da njena potklasa specizira objekte koje stvara.
- Klasa prenosi odgovornost na jednu od nekoliko pomoćnih potklasa.

U Javi se ovaj uzorak za projektovanje dosta koristi. Primer: *InetAddress* je klasa koja reprezentuje IP protokol. Postoje dve verzije IP adresa: *ipv*4 i *ipv*6. Zbog toga postoje dve potklase klase *InetAddress*: *InetAddress*4 (za *ipv*4) i *InetAddress*6 (za *ipv*6). Umesto da korisnik za ove klase piše svoju funkciju za dedukciju tipa adrese, koristi se statički metodi klase InetAddress kao što je, na primer, $getByName()$ koji vraća *InetAddress*4 ili *InetAddress*6 objekat:

```
InetAddress address4 = InetAddress.getByName(,,www.google.rs‘‘);
InetAddress address6 = InetAddress.getByName(,,ipv6.google.com‘‘);
```

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



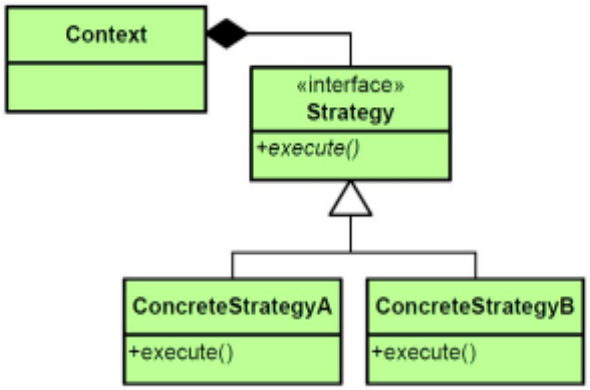
82 Objasniti kada se i kako primenjuje uzorak Strategija.

Definisati familiju algoritama, enkapsuliraj svaki. Ovaj obrazac reprezentuje skup različitih strategija, gde se primenjuje jedna, odgovarajuća strategija u

zavisnosti od konteksta.
Koristi se kada:

- se mnoge srodne klase razlikuju samo u njihovom ponašanju. Strategija obezbeđuje način da podesite klasu sa jednim od mnogih ponašanja.
- potrebne su 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. Obrazac Strategija se ovde upotrebljuje da bi se izbeglo izlaganje složenih struktura podataka specičnih 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.



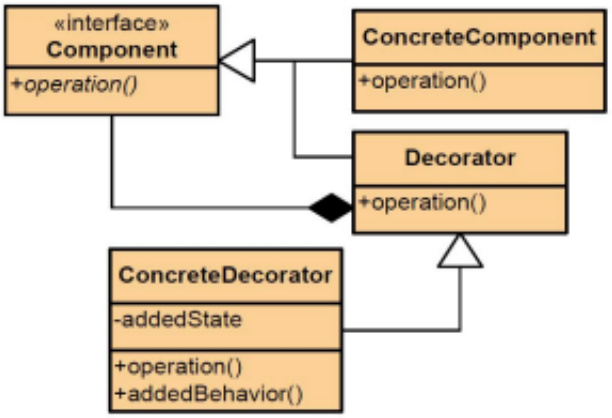
84 Objasniti kada se i kako primenjuje uzorak Dekorater.

Dekorater omogućava da se funkcionalnosti objekta povećaju bez menjanja njene strukture. Pravi se dekorater klase koji predstavlja okvir za originalnu klasu, koji proširuje njenu funkcionalnost, ne menjajući njenu suštinu.
Koristi se kada:

- je potrebno da se dodaju odgovornosti pojedinačnim objektima, a da to ne utiče na druge objekta.
- je potrebno dodati neke obaveze koje se mogu povući.

- Kada je proširivanje potklase nepraktično. Ponekad je moguć veliki broj nezavisnih ekstenzija, što bi proizvelo eksploziju uz svaku kombinaciju.

85 Skicirati dijagram klasa uzoraka za projektovanje Dekorater.



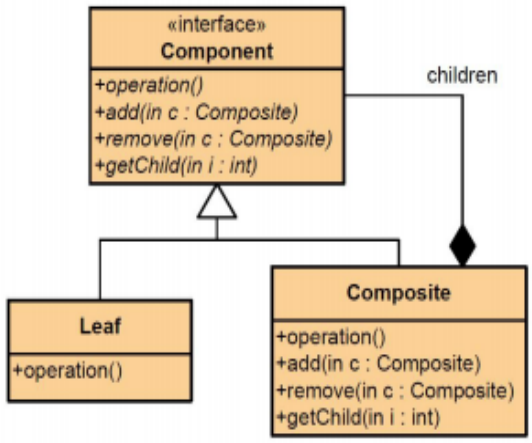
86 Objasniti kada se i kako primenjuje uzorak Složeni objekat (Sastav, Composite).

Ponekad je potrebno da grupu objekata tretiramo isto kao i jedan objekat. Tada koristimo sastav kao uzorak za projektovanje.
Koristi se kada:

- je potrebno predstaviti celu hijearhiju objekata;
- je potrebno ignorisati razliku između kompozije objekata i jednog objekta.

Primer: Na kursu KK (Konstrukcija Kompilatora) se implementira kompilator za deo jezika *Paskal* koji generiše apstraktno sintaksno stablo na osnovu koda. Tu naredba(čvor) može da bude tipa naredba ispiši ili naredba dodele, a može da bude blok koji se sastoji od niza naredbi. U ovom slučaju je poželjno koristiti sastav.

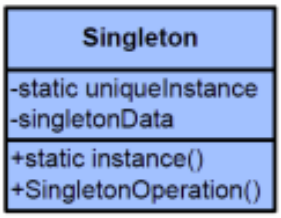
87 Skicirati dijagram klasa uzoraka za projektovanje Slozeni objekat (Sastav, Composite).



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

Koristi se kad je potrebno osigurati da klasa ima samo jednu instancu i obezbediti globalnu tačku pristupa za nju.

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



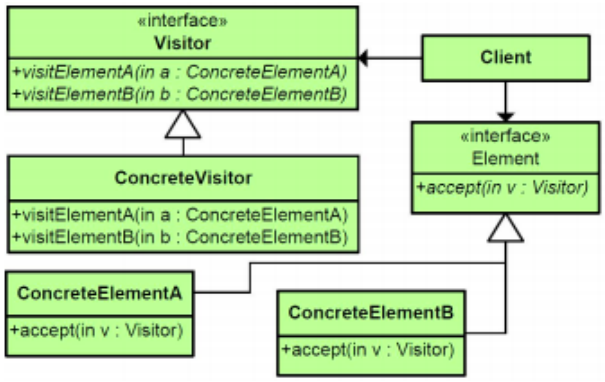
90 Objasniti kada se i kako primenjuje uzorak Posetilac (Visitor).

Poenta uzorka posetilac je mogućnost da se definiše nova operacije bez vršenja promena na već postojećoj strukturi objekta. Posetilac ne zna nužno kojeg je tipa objekat (ako se pristupa objektu preko pokazivača ili reference), ali objekat sigurno zna kojeg je on tipa. Zbog toga je potrebno da objekat prihvati posetu

od posetioca kako bi posetilac mogao da izvrši odgovarajuću operaciju.
Koristi se kada:

- struktura objekta sadrži mnoge klase objekata sa različitim interfejsima i potrebno je izvršiti operacije nad ovim objektima koji zavise od njihovih konkretnih klasa;
- je potrebno izvršiti mnoge razne nepovezane operacije zajedno;
- kada je struktura objekata deljena od strane dosta aplikacija, u posetilac se mogu staviti operacije samo onih aplikacija kojima je to potrebno;
- kada se klase koje određuju strukturu objekata retko menjaju, ali je često potrebno definisati novu operaciju nad tom strukturom.

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

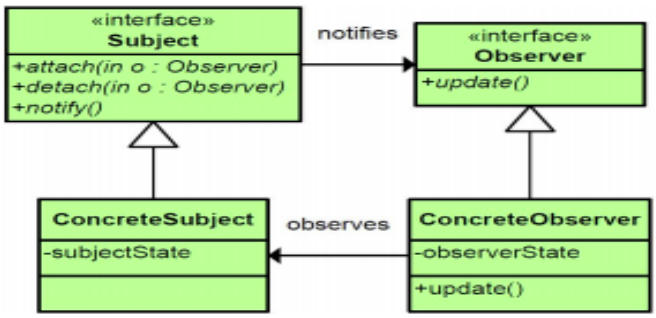


92 Objasniti kada se i kako primenjuje uzorak Posmatrač (Observer).

Posmatrač se koristi kada postoji jedan-ka-više veza između objekata tako da ako se desi promena na određenom objektu potrebno je automatski obavestiti sve druge objekte koje zavise od njega.
Koristi se kada:

- apstrakcija ima dva aspekta, gde je jedan zavisan od drugog. Enkapsulacija ovih aspekata u odvojenim objektima omogućava da ih menjamo i koristimo samostalno;
- kada promena u jednom objektu zahteva promenu u drugim objektima, gde je broj objekata koji treba da se promene nepoznat;
- kada objekat treba da obavesti druge objekte bez pretpostavke o tome kojeg su tipa ti objekti.

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



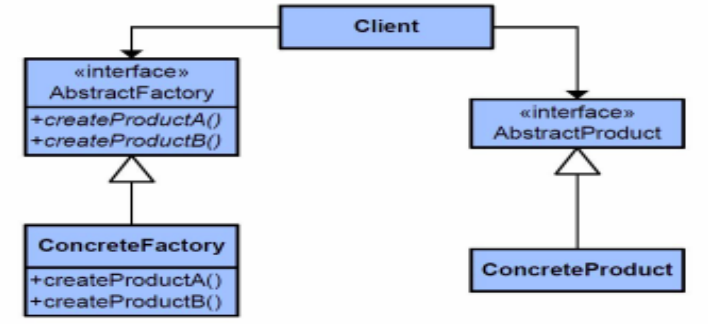
94 Objasniti kada se i kako primenjuje uzorak Apstraktna fabrika (Abstract Factory).

Apstraktna fabrika se koristi kada je potrebno kreiranje porodice srodnih ili zavisnih objekata bez navođenja njihove konkretne klase.
Koristi se kada:

- sistem treba da bude nezavisan od toga kako su njegovi proizvodi kreirani, od sastava, i načina predstavljanja.
- sistem treba da bude konstruisan sa više od jedne porodica klasa.
- familija srodnih klasa je dizajnirana da se koriste zajedno, i treba da sprovede ovo ograničenje.
- je potrebno da se obezbedi biblioteka klase proizvoda, a takođe je potrebno da se otkrije samo njihov interfejs, a ne i njihova implementacija.

Primer: Implementacija više tema (*light/dark*) za interfejs. [8]

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



96 Šta je Agilni razvoj softvera?

Agilni razvoj softvera (ARS) je familija metodologija nastala krajem 80-tih, početkom 90-tih. Naziv je oblikovan 2001. godine kada je formulisan **Manifest agilnog razvoja softvera**. Agilni razvoj softvera ima dosta sličnosti sa objektno orijentisanim metodologijama, ali sa drugačijem pristupu planiranja. Uglavnom se pretpostavlja upotreba tehnika OO projektovanja i programiranja. Propisuje skup principa tehnika za njihovo ostvarivanje. Promoviše dinamičan i disciplinovan timski rad. Brzo se reaguje na svaku promenu u okruženju. [9][10]
Agilni razvoj **NIJE**:

- Odsustvo sistematičnosti i strukturnog prisustva;
- 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 pretpostavke na kojima počiva agilni razvoj i određuje 12 osnovnih principa agilnih metodologija. Konkretne agilne metodologije mogu imati dodatne pretpostavke, principe i da određuju metode i tehnike kojima se principi ostvaruju.
Pretpostavke:

- Pojedinci i saradnja ispred procesa i alata;
- Funkcionalan softver ispred iscrpne dokumentacije;
- Saradnja sa klijentom ispred pregovaranja;
- Reagovanje na promene ispred praćenja plana.

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

Ima smisla staviti pojedince i saradnju ispred procesa i alata, jer su ljudi ti koji odgovaraju na zahteve posla i razvijaju procese. Ako procesi ili alati vode razvoj, onda su manje šanse da tim uspešno odgovori na promene ili ispoštuje zahteve klijenta. Komunikacija je primer gde može da se predstavi razlika između vrednovanja pojedinaca naspram vrednovanja procesa. U slučaju vrednovanja pojedinaca, komunikacija dolazi prirodno kada je potrebna, a u slučaju vrednovanja procesa je komunikacija zakazana i zahteva konkretnu temu.

Ljudi predstavljaju najvažniji deo uspešnog razvoja. Dobar proces ne može uspeti sa lošim ljudima, dok loš proces i najbolje ljude čini neproduktivnim. Veoma je bitno da se izgradi dobar tim u kojem će biti dobra saradnja. Dobri alati su dobrodošli, ali previše pažnje posvećeno alatima je jednako loše kao potpuno odsustvo alata.

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

Agilne metodologije ne eliminišu dokumentaciju, jer je softver bez dokumentacije beskoristan. Međutim, agilne metodologije više vrednuju funkcionalan softver nego dokumentaciju. Taj stav ima smisla, jer je osnovni cilj softver, a ne dokumentacija. Korisno je pisati popratnu dokumentaciju, ali bez preteranog iscrpljivanja.

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

Softver ne može da se naručuje kao nameštaj, jer u svakom malo složenijem slučaju je praktično nemoguće da se napiše opis zadatka i prepusti nekome da se napravi softver. Bitna je aktivna komunikacija sa klijentom i tokom razvoja. Klijent nije tehničko lice i ne može da zna precizno šta želi, a ne može ni da zna šta razvojni tim može da uradi za njega. Zbog toga uspeh projekta zavisi od redovne komunikacije sa klijentom.

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

Tradicionalno je promena smatrana kao dodatan trošak, zbog čega je izbegavana. Promene su neizbežne i sigurno nastupaju, a pitanje je samo kada će nastupiti i koje će promene nastupiti. Zbog toga je bitno da razvojni tim bude sposoban da reaguje na promene. Takođe je potrebno da plan bude fleksibilan kako bi se lako prilagodio tim promenama. Planiranje treba da se vrši na kraći period, jer se detaljni planovi, iako su koristni, teško održavaju na dužem periodu. Naravno, vizija i glavni ciljevi su neophodni i bitno je da se zna šta se pravi (ipak, ponekad se i oni menjaju).

102 Navesti bar 8 principa agilnog razvoja softvera.

1. Najviši prioritet je zadovoljiti klijenta kroz rano i neprekidno isporučivanje vrednog softvera.
2. Treba uvek otvoreno prihvatani promene, čak i u kasnim fazama razvoja. Promene se uvažavaju kao sredstvo postizanja kvaliteta za klijenta.
3. Softver treba isporučivati često, od nekoliko nedelja do nekoliko meseca sa preferencom na što kraćem periodu.
4. Poslovni ljudi i razvijaoci moraju sarađivati svakodnevno na projektu.
5. Zasnivati projekat na motivisanim pojedincima. Pružati im okruženje i potrebnu podršku i imati poverenja da će obaviti posao.
6. Najefikasniji način za razmenu informacija u okviru razvojnog tima je razgovor lice u lice. Timovi bi trebalo da rade u istoj zgradi.
7. Funkcionalan softver je osnovno merilo napretka.
8. Agilni procesi promovišu uzdržani razvoj. Sponzori, razvijaoci i korisnici bi trebalo da održavaju ujednačeni ritam.
9. Neprekidno posvećivanje pažnje tehničkoj doteranosti i dobrom dizajnu podiže agilnost.
10. Jednostavnost, umetnost da se maksimizuje količina posla koji se ne obavlja, je od suštinskog značaja.
11. Najbolje arhitekture, zahtevi i projekti potiču iz samoorganizovanih timova.
12. U redovnim intervalima tim mora da sagleda svoj rad i mogućnosti unapređivanja svoj ponašanja i efikasnosti. [10]

103 Navesti bar 3 metodologije agilnog razvoja softvera.

1. Agilno modeliranje
2. Agilan objedinjen proces (AUP)
3. Ekstremno programiranje (XP)
4. Otvoren objedinjen proces (OpenUP)
5. Scrum

104 Šta je ekstremno programiranje?

Jedna od najpoznatijih agilnih metodologija. Čini da jednostavne, međusobno zavisne metode u skladu sa pretpostavkama i principima ARS i preciznije opisuje ostvarivanja principa ARS. Trenutno nije među najzastupljenijim metodologijama, ali se principi ove metode koriste.

Zašto se zove **„Extreme Programming“**?
- Zato što tako zvuči „kul“.
- Ovaj naziv su odredili menadžeri, ne programeri.
- Prikladnija alternativa: **„BusinessValueOrientedProgramming“**. [11]

105 Navesti bar 8 metoda ekstremnog programiranja.

1. Klijent je član tima (The Customer)
2. Korisničke celine (Stories)
3. Kratki ciklusi (Lifecycle, Weekly Cycle, Quarterly Cycle)
4. Praćenje toka razvoja
5. Testovi prihvatljivosti (Acceptance Tests)
6. Programiranje u paru (Pair Programming)
7. Razvoj vođen testovima (Test-Driven Development)
8. Kolektivno vlasništvo (Collective Code Ownership)
9. Neprekidna integracija (Continuous Integration)
10. Uzdržan ritam (Sustainable pace)
11. Otvoren radni prostor (Open Work Space)
12. Igra planiranja (Planning Game)
13. Jednostavan dizajn (Simple Design)
14. Refaktorisanje (Refactoring)
15. Metafora (Metaphore)[12][13]

106 Objasniti metod ekstremnog programiranja „Klijent je član tima“.

Metoda „Klijent je član tima“ (eng. „The customer“) podrazumeva da u timu, pored programera, mora da postoji i barem jedan predstavnik klijenata. Idealno je da je predstavnik stalno tu i sarađuje sa razvijaocima. U ekstremnom programiranju klijent ima uloga člana tima.

107 Objasniti metod ekstremnog programiranja „Korisničke celine“.

Metoda „Korisničke celine“ (eng. „Stories“) podrazumeva da postoje kratka objašnjenja šta klijent želi u okviru neke celine koju razvojni tim treba da implementira. Radi okvirnog planiranja potrebno je sagledati 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.

Korisničke celine služe kao referenca na nešto što će se detaljnije razmatrati kada dođe red na implementaciju.

108 Objasniti metod ekstremnog programiranja „Kratki ciklusi“.

Metoda „Kratki ciklusi“ (eng. „Lifecycle“) podrazumevaju redovne iteracije (eng. Weekly Cycle) i izdanja (eng. Quarterly Cycle, release).

Iteracija: Sinonim nedeljni ciklus već dosta govori o samom pojmu iteracija. Iteracije su manji ciklusi koji traju relativno kratko, jednu-dve nedelje. Iteracija počinje sa određivanjem budžeta i dužine trajanja na osnovu odabranih korisničkih celina (bira ih klijent uz konsultaciju sa razvojnim timom) koji bi trebalo da budu urađeni u okviru te iteracije. Zadatak je da razvojni tim na kraju iteracije da klijentu deo softvera (rađen po prethodnom dogovoru) koji klijent može da testira.

Izdanje: Takođe sinonim, tromesečni ciklus dosta govori i samom pojmu izdanja. Izdanje obuhvata nekoliko iteracija (primer: 6). Ovo se odnosi na neku značajniju celinu u odnosu na delove koji se obrađuju u okviru iteracija. Bitno je da klijent ostavi generalni plan koji predstavlja „Pogled na šumu“ programerima dok su oni na „drveću“. Sastoji se od korisničkih celina sa procenama i prioritetima. Uokvirno se definišu sadržaji pojedinačnih iteracija. Iteracija mogu da se menjaju, a čak može i da se promeni broj iteracija u okviru izdanja (nisu poželjne veće promene).

109 Objasniti metod ekstremnog programiranja „Testovi prihvatljivosti“.

Metoda „Testovi prihvatljivosti“ (eng. „Acceptance Tests“) podrazumeva da postoje testovi u okviru korisničkih celina. Ove testove određuje klijent i pišu se neposredno ili čak paralelno sa implementacijom iste celine. Testovi se po mogućnosti automatizuju. Ovi testovi se ponavljaju svaki put po izgradnji sistem (nekoliko puta dnevno). [14]

110 Objasniti metod ekstremnog programiranja „Programiranje u paru“.

Metoda „Programiranje u paru“ (eng. „Pair programming“) podrazumeva da se delovi softvera realizuju u paru (programera), gde dvoje ljudi sede jedan pored drugog za istom mašinom („Dve glave su pametnije od jedne“). Jedan član tima piše kod, a drugi komentariše i ispravlja kolegu ako je potrebno.

Uloge se menjaju više puta. Istraživanja pokazuju da se ovako značajno smanjuje broj grešaka, a greška koštaju, pa samim tim se smanjuje cena projekta. Ovo podrazumeva da pojedinci mogu da sarađuju. Bitno je naglasiti da specijalnost i dalje ostaje na pojedincima, ali su drugi ipak upoznati sa rezultatima.

U okviru iteracije svaki član tima mora da radi sa svim članovima tima i da radi skoro na svim delovima iteracije. Ovim se informacije veoma efikasno razmenjuju i svaki član tima je upoznat sa svakim delom projekta.

111 Objasniti metod ekstremnog programiranja „Razvoj vođen testovima“.

Metoda „Razvoj vođen testovima“ (eng. „Test-Driven Development) podrazumeva da razvijanje počinje pisanjem testova. Naravno, ti testovi na početku ne mogu da prođu, jer funkcionalnosti nisu implementirane. Nakon napisanih testova se piše kod koji prolazi date testove. Programer naizmenično piše testove i kod koji prolazi te testove.

Prednosti koje ova metoda donosi su:

- Uz kod se dobija kompletna kolekcija testova;
- Testovi mogu da se koriste da se proveri da li jedinica koda radi ispravno ili ne;
- Sprečava nastajanje grešaka prilikom naknadnih izmena;
- Pomaže prilikom refaktorisanja;
- Metod vrši pritisak da se razdvajaju jedinice koda, čime se dobija kvalitetniji dizajn.

112 Objasniti metod ekstremnog programiranja „Kolektivno vlašnistvo“.

Metoda „Kolektivno vlašnistvo“ (eng. „Collective code ownership“) podrazumeva da svako može da proveri bilo koji deo modula i da ga uporedi. Nijedan programer nije pojedinačno odgovoran za bilo koji konkretan modul ili tehnologiju. Svako može da radi na bilo kojoj oblasti. To ne znači da se specijalnosti i dalje ne poštuju. Svako će najviše da radi u svojoj struci, ali može da se priključi i u druge celine i da uči od drugih specijalista.

113 Objasniti metod ekstremnog programiranja „Neprekidna integracija“.

Metoda „Neprekidna integracija“ (end. „Continuous process“) podrazumeva da se intenzivno koristi sistem za upravljenje verzijama, gde se na dnevnom nivou više puta postavlja nova verzija koda. Najveća prednost ove metode je to što se problemi prilikom integracije vrlo brzo prepoznaju i rešavaju.

Obično se prave testovi i produkcioni kod koji se kasnije, u nekom pogodnom trenutku (relativno često), integrišu. Pre svake integracije se proverava da li svi testovi prolaze.

Prednosti:

- Postupak debagovanje je značajno lakši;
- Testovi ukazuju da postoji problem (ukazuju na njegovu prostornu i vremensku lokaciju);
- Istorija verzija omogućava sužavanje prostorne i vremenske lokacije nastajanja problema.

114 Objasniti metod ekstremnog programiranja „Uzdržan ritam“.

Metoda „Uzdržan ritam“ (eng. „Sustainable pace“) podrazumeva zabranu prekovremenog rada. Jedini izuzetak za prekovremeni rad je poslednja nedelja izdanja: „Ako je tim dovoljno blizu cilja, *sprint* je dopušten.

„Razvoj softvera nije *sprint*, nego maraton.“ (R. Martin)

Prednosti: Bolji odnosu u timu, manja rasterećenost članova tima, manji broj grešaka i efikasnija iskorisćenost radnog vremena.

115 Objasniti metod ekstremnog programiranja „Otvoren radni prostor“.

Metoda „Otvoreni radni prosto“ (eng. Open work space) podrazumeva da tim radi u jednoj velikoj otvorenoj prostoriji, gde se na svakom stalo nalaze dve

ili tri radne stanice, par stolica, a na zidovima su table i panoi. Uobičajni ton je tiha komunikacija i svi mogu međusobno da komuniciraju. Iako u teoriji ova metoda zvuči loše, u praksi se pokazuje da znatno poboljšava efikasnost.

116 Objasniti metod ekstremnog programiranja „Igra planiranja“.

Metoda „Igra planiranja“ (eng. Planning Game) podrazumeva podelu odgovornosti između klijenta i razvojnog tima, gde klijent odlučuje šta je značajna karakteristika softvera, a razvijaoci odlučuju koliko bi ta karakteristika koštala.
U okviru planiranja izdanja postoje tri faze:

- Faza istraživanja: Klijenti definišu zahteve preko korisničkih priča;
- Faza ivršavanja: Planiranje i implementacija klijentskih zahteva;
- Faza upravljanja: Prethodno definisani planovi se usaglašavaju sa zahtevima projekta.

U okviru planiranja iteracije postoje tri faze:

- Faza istraživanja: Beleže se zahtevi projekta;
- Faza ivršavanja: Podela poslova po razvijaocima i njihova izrada;
- Faza upravljanja: dobijeni rezultati se detaljnije upoređuju sa početnim zahtevima.

117 Objasniti metod ekstremnog programiranja „Jednostavan dizajn“.

Metoda „Jednostavan dizajn“ (eng. Simple design) podrazumeva da se pravi ono što je u tom trenutnu potrebno, bez komplikovanja unapred. Cilj je da dizajn bude što jednostavniji i što izražajniji. Pažnja se posvećuje samo onome što je planirano za tekuću iteraciju i ne razmatra se ono što će možda doći kasnije. Razvoj obično ne potiče od infrastrukture.
Tri osnovna pravila:

- **Razmotriti prvo najjednostavnije rešenje koje bi moglo da uradi posao:** Cilj je da se sa što manje rada i u što kraćem vremenu doći do cilja, a dizajn se može unapređivati kasnije. Primeri:
  - Ako nešto može da se reši preko datoteka, nema potrebe da se odmah upliću baze podataka ili objekti srednjeg sloja.
  - Ako nešto može bez paralelnog izračunavanja, valja tako i rešiti.
- **„Neće biti potrebno! “:** Kad god se postavlja pitanje da li je nešto potrebno ili ne onda je odgovor: „Neće biti potrebno! “. Izbegava se dodavanje nečega što će možda biti potrebno. Cilj je izbegavati komplikovan dizajn radi nečega što neće biti potrebno.

- **„Jedanput i samo jedanput“:** Ekstremno programiranje ne dopušta ponavljanje u kodu. Kada god se naiđe na ponavljanje u kodu, ono se mora ukloniti. Čak i kad su delovi koda veoma slični, oni se moraju apstrahovati. Cilj je minimizovati redundantnost koda apstrahovanjem.

Napomena:

- Razmatranje najjednostavnijeg rešenja nikako ne znači da kod sme da bude loše dizajniran.
- „Jednostavno“ ne znači „na brzinu“ ili „nepažljivo“.

118 Objasniti metod ekstremnog programiranja „Refaktorisanje“.

Metoda „Refaktorisanje“ (eng. „Refactoring“) podrazumeva učestalo popravljanje i održavanje kvaliteta koda refaktorisanjem.

- **Šta je refaktorisanje?** Refaktorisanje je proces koji se sastoji iz niza manjih transformacija nad kodom bez menjanja ponašanja koda. Ove transformacije služe samo za poboljšanje kvaliteta koda. Refaktorisanjem se ne dodaje nova funkcionalnost.
- **Zašto refaktorisati?** „Kod je kvarljiv“ R. Martin
  Jednom napisan kod nije gotov, jer će u jednom trenutku biti potrebno da se izvrši neka promena. Postepenim menjanjem dizajna koda se kod kvari. Ako dodajemo neku novu funkcionalnost na kod, njegov kvalitet može da opadne. Ekstremno programiranje se suprostavlja lošem kvalitetu koda čestim refaktorisanjem. Testovi su jako korisni prilikom refaktorisanja, jer oni proveravaju da li se ponašanje koda promenilo.
- **Kada refaktorisati?** Refaktorisanje treba stalno da se primenjuje u hodu, kada se primeti da neka izmena narušava postojeći dizajn. Refaktorisanje se vrši paralelno sa razvojem testova i produkcionog koda. Dobra je praksa da se kod refaktoriše svaki put pre nego što se doda nova funkcionalnost.

119 Objasniti metod ekstremnog programiranja „Metafora“.

Metoda „Metafora“ (eng. „Metaphore“) podrazumeva da postoji velika slika čitavog sistema koja predstavlja viziju sistema kao celine. Iz metafore (ne)posredno potiču svi konkretni moduli i zahtevi.

- Ako ne znamo šta želimo sa projektom, onda ne treba ni da ga razvijamo.
- Često se realizuje u vidu rečnika pojmova koji identifikuju najvažnije koncepte sistema i problema koji bi on trebalo da reši.

120 Šta je „Razvoj vođen testovima“?

Razvoj vođen testovima je stil vođenja razvoja koji kaže sledeću stvar: Ako treba nešto da se isprogramira, pre nego što se isprogramira, prvo se piše deo koda koji proverava da li taj kod radi kako treba. Prvo se pišu testovi, potom se piše kostur datog koda koji u početku ne radi (ne prolazi testove), ali omogućava da se kod prevede. Nakon toga se piše kod.

Kada se priča o razvoju vođen testovima, onda se prvenstveno priča o testovima jedinica (eng. unit test).

Napomena: Pisanje testova nije dokaz korektnosti koda!

121 Navesti i objasniti vrste testova softvera.

- **Testovi jedinica (Unit tests):** Testovi jedinica koda. Jedinica koda može biti funkcija, metod, klasa, komponenta, ...
- **Integracioni testovi (Integration tests):** Testovi kojim se proverava da li moduli funkcionišu zajedno.
- **Sistemski testovi (System Tests):** Testiranje sistema kao celine.
- **Testovi prihvatljivosti (Acceptance Tests):** Testiranje iz ugla korisnika. Često se izvode preko skriptova koji simuliraju ili emuliraju korisnički interfejs.

122 Navesti i objasniti ukratko osnovne principe razvoja vođenog testovima.

Razvoj vođen testovima je princip koji kombinuje princip „Testovi prethode kodu“ i sistematičnost (ne sme da postoji nijedan deo koda koji nije pokriven prilikom pravljenja testova). Ovaj princip takođe podrazumeva refaktorisanje koda nakon što prođu svi testovi.

123 Objasniti princip razvoja vođenog testovima „Testovi prethode kodu“ i način njegove primene.

Osnovna pravila:

- Pre pisanja koda se prave odgovarajući testovi.
- Nijedna funkcija programa se ne razvija sve dok ne postoji test koji ne uspeva zbog njenog odsustva.
- Nijedna linija koda ne sme da se dodaje dok postoji test koji zbog nje ne uspeva.

- Tek nakon definisanih testova mogu da se implementiraju funkcionalnosti koje zadovoljavaju uslove tih testova.

Svaka iteracija počinje sa pisanjem testova. Ti testovi se često najpre ne prevode zbog nedostataka odgovarajućih metoda ili klasa. Zatim se piše kostur koji omogućava da se kod prevede (obično se vraćaju netačni rezultati, npr. funkcija vraća nulu). Nakon toga se piše kod koji zadovoljava testove.

Princip „Mali Koraci“:

- Treba testirati što manje celine;
- Ako prolaze testovi, a postoji bag, onda testovi nisu dovoljno dobri;
- Iteracije pisanja testova i pisanje koda se brzo smenjuju;
- Poželjno je da testovi i kod evoluiraju zajedno gde testovi malo prethode kodu;
- Nije dobro napisati veliki broj testove bez odgovarajućeg koda;

124 Objasniti princip razvoja vođenog testovima „Sistematičnost“.

Princip sistematičnosti podrazumeva da svaka karakteristika softvera mora da bude pokrivena testovima. Bitno je da svi opšti slučajevi budu pokriveni, a i neki specijalni slučajevi. Svi granični slučajevi moraju da budu obuhvaćeni testovima. Svaka linija koda mora biti testirana.

Tehnike:

- **Softver se razvija od vrha prema dnu:** Potrebno je imati „stub“ (privremeni deo koda) u kodu koji implementiraju interfejs i slične metode tako da ne rade ništa. Njihova svrha je samo da kod može da se prevede.
- **Softver se razvija od dna prema vrhu:** Testira se pomoću izvođača (drajvera) koji predstavljaju privremeni deo koda koji koriste manje funkcionalne celine u odsustvu velikih funkcionalnih celina koje tek treba da se implementiraju.

125 Navesti osnovne uloge testova.

1. Predstavljaju vid verifikacije;
2. Pomažu pri refaktorisanju;
3. Predstavljaju vid dokumentacije;
4. Omogućavaju više uglova posmatranja koda;
5. Pomažu pri debagovanju (lakše pronalaženje i trajne provere);

126 Objasniti ulogu testova kao vida verifikacije softvera.

Kolekcija testova omogućava progameru da proverava da li jedinica koda radi ispravno ili ne. Za svaku funkciju programa postoje odgovarajući testovi koji proveravaju njenu ispravnost. Testovi omogućavaju da se proveri da li je nešto možda loše izmenjeno. Takođe, pomažu da se rano prepoznaju greške na nivou koda ili na konceptualnom nivou. Testovi vrše dodatni pritisak na razdvajanje jedinica koda, što dovodi do boljeg dizajna.

127 Objasniti ulogu testova u okviru refaktorisanja.

Refaktorisanje podrazumeva izmene na kodu u cilju poboljšanja kvaliteta koda, ali prilikom refaktorisanja ne sme doći do promene ponašanja koda. Testovi mogu da provere da li je došlo do promene ponašanja koda prilikom refaktorisanja. Ako izmene narušavaju pravila, onda ne bi trebalo da prođu testove.

128 Objasniti ulogu testova u kontekstu ugla posmatranja koda.

Pravljenje testova postavlja programera u poziciju korisnika. U prvom planu su interfejs jedinice koda i apstrakcija njenog ponašanja. Testovima se omogućava lako upotrebljiv kod. Takođe se dobije lako proverljiv kod, što se kasnije može teško postići ako sistem nije dobro oblikovan.

129 Objasniti ulogu testova kao vida dokumentacije.

Testovi predstavljaju oblik specifikacije zahteva (prikazuju jednostavnije zahteve koje program u radu treba da ispuni). Opisuju uslove funkcionisanja koda i način njegove upotrebe (primer upotrebe interfejsa). Testovi predstavljaju karakteristične primere (granični slučajevi). Prednost testova kao dokumentacije je to što je takav tip dokumentacije konstantno ažuran.

130 Šta može biti jedinica koda koja se testira?

1. operaciju, funkciju, metod;
2. strukturu, klasu;
3. više integrisanih jedinica koda;
4. softverski paket;
5. podsistem, spoljašnji podsistem;
6. interfejs.

131 Šta može biti predmet testiranja jedinice koda?

- **Zavisnost postuslova od preduslova:** Provera se da li funkcija vraća očekivane rezultate za dati ulaz tj. da li funkcija izračunava tačno izlaz i da li funkcija na ispravan način menja stanje programa.
- **Robusnosti:** Proverava ispravnost ponašanja u slučaju neispravnih ulaznih podataka.
- **Integracija:** Rezultat integracije manjih jedinica koda je veća jedinica koda koja može biti predmet testiranja.
- **Interfejs špoljasnjeg podsistema:** Provera se da li interfejs radi po specifikaciji. Testiraju se delovi softvera koji se koristi pri izradi softvera.

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

1. CppUnit
2. CppUnitLite
3. Boost.Test
4. NanoCppUnit
5. Unit++
6. CxxTest
7. Google Test
8. CATCH
9. CPUnit

133 Opisati ukratko osnovne mogućnosti biblioteke CppUnit.

Biblioteka *CppUnit* ima bogat skup makroa za proveru tvrdnji (assert) koji su laki za korišćenje. U okviru ove biblioteke se mogu grupisati testovi po nekom pravilu, praviti posebne klase za testiranje koda koje mogu da imaju neograničen broj grupa testova. U okviru tih klasa možemo definisati podrazumevano ponašanje pre i posle testa (initialization/deinitialization). Takođe postoji mogućnost korisničkih definisanih poruka u slučaju pada testa. Za svaki tip tvrdnji postoji ista verzija koja ima sufiks „-_ *MESSAGE*“ i zahteva dodatni prvi argument za poruku.

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

U okviru projekta se pravi novi cilj (program za ivršavanje testova). Može po jedan program za svaki produkcioni cilj, ali može i jedan univerzalan. U programu se dodaje glavna funkcija koja pokreće testove. Za svaku jedinicu koja se piše po jedna ili više test jedinica koda. Svaki put kada se prevodi ciljni produkcioni kod, prevodi se kod za izvršavanje testova.

Za definisanje tvrdnji u okviru testova se koriste makroi. Makroi u slučaju greške izbacuju izuzetke se podrazumevanom ili korisnički definisanom porukom.

Koristi se „Test Slučaj“ (CppUnit::TestCase) klasa za implementiranje klasa za testiranje koda. Klase za testiranje koda naslađuju TestCase klasu. Glavni metod ove klase je „runTest()“ kojom se pokreće dati test.

Biblioteka obuhvata alate za izvšavanje testova. Novi testovi mogu se relativno jednostavno dodavati u skup već postojećih. Osnovni izvršavač testova je „Pokretač testova“ (CppUnit::TextTestRunner) koji se prvo inicijalizuje, a onda se dodaju test okviri (suite) koji on treba da pokrene. Metodom „run()“ se pokreću svi test okviri zakačeni za pokretač testova.

135 Šta je Test suit?

Test okviri („Test suit“) su grupisani testovi koji testiraju rad neke manje celine. Definišu se preko makroa u okviru CppUnit biblioteke:

```
CPPUNIT_TEST_SUITE( VektorTest );       // pocetak
  CPPUNIT_TEST( constructorTest );      // test jedinica
  CPPUNIT_TEST( operatorEqualityTest ); // test jedinica
CPPUNIT_TEST_SUITE_END();               // kraj
```

Makroi za prijavljivanje okvira implementiraju statički metod za pravljenje objekata kompleta testova (suite). Izvršavaču se ne dodaje objekat, već statički napravljen komplet: „runner.addTest(VektorTest::suite());“.

136 Navesti osnovne vrste pretpostavki koje podržava biblioteka CppUnit.

- **CPPUNIT_ASSERT:** Tvrdi se da je uslov ispunjen.
- **CPPUNIT_ASSERT:** Tvrdi se da je uslov ispunjen uz korisnički definisanu poruku.
- **CPPUNIT_FAIL:** Test pada uz definisanu poruku.
- **CPPUNIT_ASSERT_EQUAL:** Tvrdi da su dve vrednosti jednake.
- **CPPUNIT_ASSERT_EQUAL_MESSAGE:** Tvrdi da su dve vrednosti jednake (sa porukom).

- **CPPUNIT_ASSERT_DOUBLES_EQUAL:** Tvrdi se da su dve vrednosti jednake uz definisanju tačnost.
- **CPPUNIT_ASSERT_DOUBLES_EQUAL_MESSAGE:** Tvrdi se da su dve vrednosti jednake uz definisanju tačnost (sa porukom).
- **CPPUNIT_ASSERT_THROW:** Tvrdi se da je dati izraz izbacio (throw) grešku datog tipa.
- **CPPUNIT_ASSERT_THROW_MESSAGE:** Tvrdi se da je dati izraz izbacio (throw) grešku datog tipa (sa porukom).
- **CPPUNIT_ASSERT_NO_THROW:** Tvrdi se da dati izraz nije izbacio (throw) grešku datog tipa.
- **CPPUNIT_ASSERT_NO_THROW_MESSAGE:** Tvrdi se da dati izraz nije izbacio (throw) grešku datog tipa (sa porukom). [15]

137 Opisati ukratko osnovne mogućnosti biblioteke Catch. Napisati primer testa.

- Biblioteka *Catch* ima svoju implementaciju *main* funkcije koju je opcionalno koristiti. U zavisnosti od toga da li želimo da koristimo naš ili *Catch*-ov *main*, definišemo odgovarajući makro. Ako želimo *Catch*-ov main onda definišemo makro **CATCH_CONFIG_MAIN** i u nastavku koda je dovoljno samo pisati test slučajeve.
- Postoje dva tipa makroa koji proveravaju da li je uslov tačan. Razlika je u reakciji u slučaju da je uslov netačan:
  - **CHECK:** Ne prekida testiranje ostalih testova u jednom testu;
  - **REQUIRE:** Prekida testiranje ostalih testova u jednom testu.
- Pisanje test slučajeva preko makroa **TEST_CASE** sa odgovarajućim nazivom i pisanje sekcije u okviru test slučajeva preko makroa **SECTION**.

138 Koji su osnovni elementi koje programer pravi pri pravljenju testova uz primenu biblioteke Catch? Kako? Napisati primer testa.

U slučaju da koristimo predefinisan *main* u okviru *Catch* biblioteke, potrebno je implementirati sledeće elemente (podrazumeva se da je kod ili deo koda koji se testira već implementiran):

1. **TEST_CASE:** Test slučaj koji testirana neku funkcionalnost koda (primer: klase). Bitno je da test slučaj ima smisleno ime.
   - Dobar primer: „Two objects created using two parameters should have same state“;
   - Loš primer: „test3“.

2. **SECTION:** Sekcija u okviru test slučaja koja pokriva neki deo test slučaja koji čini celinu. Primer: Testiranje specijalnih slučajeva.

3. **CHECK/REQUIRE:** U okviru test slučajeva (sekcija) se pišu tvrdnje koje proveravaju da li je uslov tačan. Biblioteka *Catch* ima bogat skup različitih tvrdnji.

AAA Pattern: Poželjno je da programer piše svaku sekciju koristeći AAA šablon tj. Arrange-Act-Assert:[16]

- *Arrange*: Definisanje ulaznih parametara i preduslova;
- *Act*: Delanje na objekat ili metodu (poziv metode, konstruktora, ...);
- *Assert*: Provera uslova.

Primer: Zadatak sa vežbi koji proverava rad konstruktora i operatora jednakosti.

```
TEST_CASE("When two integers are created using the same values Then they
   need to be equal", "[equality]")
{
    // Testove u jednom skupu mozemo podeliti u sekcije
    SECTION("Base-10 tests")
    {
        // Arrange
        const auto value = 1000;
        const auto base = 10;

        // Act
        ceo_broj num1(value, base);
        ceo_broj num2(value, base);

        // Assert
        REQUIRE(num1 == num2);
    }
}
```

139 Šta je Test case? Šta je Test case section? (Catch)

Pogledati odgovor na prethodno pitanje.

140 Navesti osnovne vrste pretpostavki koje podržava biblioteka Catch?

- **CHECK/REQUIRE:** Proverava da li je uslov tačan.
- **CHECK_FALSE/REQUIRE_FALSE:** Proverava da li je uslov netačan.
- **CHECK_NOTHROW/REQUIRE_NOTHROW:** Proverava da li izuzetak nije izbačen.
- **CHECK_THROWS/REQUIRE_THROWS:** Proverava da li je izuzetak izbačen.

- **CHECK_THROWS_AS/REQUIRE_THROWS_AS:** Proverava da li je specifičan izuzetak izbačen.
- Ostale, naprednije tvrdnje ... [17]

141 Šta su testovi prihvatljivosti?

Testovi prihvatljivosti se odnose na testiranje sistema kao celine. Obično se definišu nekim specifičnim skript jezikom i relativno često se koristi XML. Obuhvataju sve aspekte upotrebe sistema kao što su: definisanje podataka, izvšavanje operacija, proveravanje rezultata.

142 Po čemu se testovi prihvatljivosti razlikuju od testova jedinica koda?

Testovi jedinica koda, baš kao po imenu, testiraju da li odgovarajuća jedinica koda radi kako treba, a testovi prihvatljivosti testiraju da li program u celini radi ono što treba.

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

Testove jedinica koda pišu programeri, a testove prihvatljivosti članovi razvojnog tima, ali ne nužno programeri (obično ljudi koji nisu direktno uključeni u samo pisanje programa).

144 Kakav je odnos refaktorisanja i pisanja programskog koda?

Refaktorisanje je tehnika kojom se kod popravlja u smislu dizajna kako bi kod bio razumljiviji, lakši za održavanje i kako bi se povećala efikasnost pisanja koda. Kada je kod dobro struktuiran onda je lako prepoznati šta treba izmeniti i kako uneti tu izmenu.

Imamo delove koda koji su nekako povezani, a refaktorisanjem menjamo način kako su povezani, a da se pritom ponašanje programa ne menja (menja se struktura). Refaktorisanje ne menja ponašanje programa, već samo strukturu.

Pisanje novog koda ne treba da menja postojeći kod već samo da dodaje novo ponašanje. Programiranje se sastoji iz naizmeničnog refaktorisanja i pisanja novog koda.

145 Šta su osnovni motivi za refaktorisanje koda?

Osnovni motiv za refaktorisanje je podizanje kvaliteta dizajna softvera:

- Softver postaje razumljiviji;

- Refaktorisanje pomaže u traženju grešaka;
- Refaktorisanje poboljšava brzinu pisanja novog koda, pa samim tim i ukupno vreme pisanja

146 Kada se pristupa refaktorisanju koda?

Refaktorisanje treba da bude redovno. Dobra praksa je da se kod refaktoriše svaki put pre dodavanja novog koda.
Slučajevi:

- „Kada se nešto ponovi po treći put“;
- Kada se dodaju nove funkcije;
- Kada se proverava ispravnost i kvalitet koda;
- Kada je potrebno da se pronađe bag.

Napomena: Kada se refaktoriše da bi se pronašao bag treba refaktorisati samo delove koda za koje se zna ponašanje. Refaktorisanje podrazumeva da se ponašanje programa ne menja. Ako postoji bag u programu, onda to znači da ponašanje koda nije skroz poznato, što znači da u tom trenutku „refaktorisanje“ može vrlo lako da promeni ponašanje koda.

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

Veoma je nezahvalno definisati uslove po kojima se određuje da li bi neki kod trebalo refaktorisati. Umesto toga se izdvajaju kandidati u kojima je potrebno razmotriti da li se refaktorisanjem može dobiti kvalitetniji kod.

„Ako zaudara, potrebno ga je izmeniti“ („If it stinks, change it! “) - nečija baba

Ne može se skroz objektivno odrediti da li kod treba refaktorisati ili ne. Osetljivost programera na određene probleme nije ista. Programer sa većim iskustvom će bolje umeti da prepozna moguća unapređenja. Programer može neke transformacije koda smatrati trivijalne, zbog čega će se odgovarajuća transformacija 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 da se ta tehnika ne primeni.

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

1. Ponavljanje koda (Duplicate code)
2. Dugački metodi (Long Methods)
3. Velika klasa (Large Class)
4. Dugačka lista argumenata
5. Divergentne promene (Divergent Change)
6. Distribuirana apstrakcija (Shotgun Surgery)
7. Velika zavisnost od drugih klasa
8. Grupisanje podataka
9. Poplava primitivnih podataka
10. Naredba switch
11. Paralelne hijerarhije nasleđivanja
12. Lenje klase
13. Spekulativno uopštavanje (Speculative Generality)
14. Privremeni podaci (Temporary Field)
15. Lanci poruka (Message chains)
16. Posrednik (Middle Man)
17. Nepoželjna bliskost (Inappropriate Intimacy)
18. Alternativne klase sa različitim interfejsima
19. Nepotpuna biblioteka
20. Klasa podatak
21. Nepoželjno nasleđivanje
22. Komentari (Comments) [18]

149 Zašto je dobro eliminisati ponavljanja iz koda? (refaktorisanje)

(eng. Duplicate Code)

Simptomi:

- Dva dela koda su identična.

Uzrok:

- Različiti programeri su napisali isti kod ne znajući da je neki njihov kolega već to uradio.
- Ponekad je dosta suptilno da dva koda imaju isto ponašanje.
- Brzanje ili lenjost (sprint).

Rešenje:

- Izvlačenje metoda (deo koda se izvlači u zasebnu funkciju/metod).
- Povlačenje metoda kroz hijearhiju (dve klase imaju iste atribute, ti atributi mogu biti izdvojeni u natklasu).
- Pravljenje šablonskih metoda (izdvajaju se delovi nekog algoritma u natklasu).
- Zamena algoritma.
- Izdvajanje klase (deo posla se izdvoji u posebnu klasu).

Isplativost:

- Kraći kod i lakše održavanje.

150 Zašto dugački metodi mogu predstavljati problem? (refaktorisanje)

(eng. Long Method)

Simptomi:

- Metod je predugačak. Za svaki metod koji je duži od deset linija potrebno je razmisliti od refaktorisanju.

Uzrok:

- Konstantno dodavanje koda (dve po dve linije).

Rešenje:

- Izdvajanje metoda.
- Zamena privremenih vrednosti upitima (lokalna promenljiva koja se koristi u raznim metodama se izdvaja kao metoda).

- Uvođenje parametarskog objekta (ako se stalno u metodima prosleđuju neki argumenti zajedno, onda se ti parametri izdvajaju u jedan objekat).
- Zamena metoda objektom (od velikog metoda se pravi zasebna klasa).
- Dekompozicija uslova (blokovi u okviru grananja se izdvajaju u posebne metode).

Isplativost:

- Lakše održavanje, razumevanje i debagovanje.
- Dugački metodi nude savršeno sklonište za duplikate koda.

151 Zašto velika klasa može da predstavlja problem? (refaktorisanje)

(Large Class)

Simptomi:

- Klasa je predugačka (broj linija koda).
- Klasa ima previše metoda;
- Klasa ima previše atributa;

Uzrok:

- Uglavnom je lakše dodati novu funkcionalnost klasi nego praviti novu klasu.

Rešenje:

- Izvlačenje klase.
- Izvlačenje potklase.
- Izvlačenje interfejsa (različiti tipovi korisnika koriste istu klasu-interfejs).
- Izdvajanje praćenih podataka u klasu (određeni podaci se pojavljuju stalno zajedno).

Isplativost:

- Ne mora da se pamti ogroman broj atributa te klase.
- Smanjuje se dupliran kod.

152 Šta su divergentne promene? Zašto su problematične? (refaktorisanje)

(end. Divergent Change)

Simptomi:

- Potrebne su promene na nekoliko metoda kada se izvrši neka izmena u okviru klase.
- Primer: Kada se dodaje novi tip artikla, potrebno je izmeniti metode za traženje, prikazivanje i naručivanje artikla.

Uzrok:

- Loša struktura programa ili „copy-paste programming“.

Rešenje:

- Izdvajanje klase.
- Izdvajanje natklase.
- Izdvajanja potklase.

Isplativost:

- Jednostavniji kod.
- Smanjuje se dupliran kod.
- Bolja organizacija koda.

153 Šta je distribuirana apstrakcija? Zašto je problematična? (refaktorisanje)

(eng. Shotgun surgery)

Simptomi:

- Potrebno je da se izvrši promena na nekoliko klasa kao posledica izmena na nekoj klasi.

Uzrok:

- Jedan posao je podeljen na nekoliko klasa.

Rešenje:

- Izdvajanje metoda (ako se metod više koristi u drugoj klasi).
- Izdvajanje podataka (ako se neki podaci više koriste u drugoj klasi).
- Umetanje klase (funkcionalnosti klase se ubace u drugu klasu).

Isplativost:

- Bolja organizacija.
- Smanjuje se dupliran kod.
- Lakše održavanje.

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

Simptomi:

- Klasa ili metod koristi veliki broj metoda druge klase za čitanje vrednosti.

Uzrok:

- Odgovornost između klasa nije pravilno podeljena.

Rešenje:

- Premeštanje metoda.
- Izdvajanje metoda.

Isplativost:

- Bolja struktura softvera.

155 Zašto naredba switch može da predstavlja problem? (refaktorisanje)

(eng. Switch Statements)

Simptomi:

- Kompleksan operator *switch* ili niz naredbi if-ova.

Uzrok:

- Kod koji je implementiran sa *switch*-om često može da se podeli.
- Kada se primeti komplikovana *switch* naredba, često je polimorfizam bolje rešenje.

Rešenje:

- Izdvajanje metoda.
- Premeštanje metoda.
- Zamena kodiranog podatka potklasom (kada ima dosta operacija nad nekim kodiranim tipom , često je dobro rešanje da se taj kodirani podatak izdvoji u klasu).
- Zamena uslova polimorfizmom.
- Zamena parametara eksplicitnim metodom.
- Uvođenje praznih objekata (kada se koristi specijalna „null“ vrednost, umesto nje je dobra praksa da se uvede specijalan prazan objekat).

Isplativost:

- Bolja organizacija koda.

156 Šta je spekulativno uopštavanje? Zašto može da predstavlja problem?(refaktorisanje)

(eng. Speculative generality)

Simptomi:

- Postoji neiskorišćena klasa, metod ili atribut.

Uzrok:

- Pisanje „Just in Case“ koda. Te funkcionalnosti se nekada uopšte ne implementiraju.

Rešenje:

- Sažimanje hijearhije (Ako postoji apstraktna klasa koja se ne koristi, onda se izbacuje).
- Umetanje klase (za neiskorišćenu klasu).
- Umetanje metoda (za neiskorišćenu metodu).
- Brisanje neiskorišćenih parametara u okviru metoda.
- Brisanje neiskorišćenih atributa.

Isplativost:

- Kraći kod.
- Lakše održavanje.

157 Zašto privremene promenljive mogu da predstavljaju problem? (refaktorisanje)

(eng. Temporary field)

Simptomi:

- Postoje privremene promenljive koje se koriste samo u nekim slučajevima.

Uzrok:

- Uglavnom nastaju prilikom implementacije nekog algoritma koji zahteva dosta ulaza.

Rešenje:

- Izvlačenje klase.
- Zamena metode sa objektom.
- Ubacivanje prazne klase.

Isplativost:

- Bolja jasnoća i organizacija koda.

158 Zašto lanci poruka mogu da predstavljaju problem? (refaktorisanje)

(eng. Message chains)

Simptomi:

- U kodu se nalaze linije oblika: *a->b()->c()->d()*

Uzrok:

- Korisnik zatraži objekat koji onda zatraži neki drugi objekat.
- Loša struktura koda.

Rešenje:

- Sakrivanje delegata (Ako klasa A komunicira sa klasom B, a ponekad komunikacija klase A sa klasom B zahteva da B komunicira sa klasom C, onda može da se napravi metod kojim se komunicira samo sa B od strane A, bez znanja o C).
- Izvlačenje metoda.
- Premeštanje metoda.

Isplativost:

- Smanjivanje zavisnosti između klasa.
- Čitljiviji kod.

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

(eng. Middle Man)

Simptomi:

- Klasa ima samo jednu funkcionalnost, a to je da bude posrednik između klasa.

Uzrok:

- Premeštanje funkcionalnosti jedne klase u drugu klasu.
- Preterivanje u rešavanju problema lanca poruka.

Rešenje:

- Uklanjanje posrednika (posrednik se briše i komunikacija postaje direktna)

Isplativost:

- Kod zauzima manje mesta.

Napomena: Neki uzorci za projektovanje prave posrednike namerno (Primer: Dekorator).

160 Šta je nepoželjna bliskost? Zašto je problematična? (refaktorisanje)

(eng. Inappropriate intimacy)
Simptomi:

- Neka klasa suviše često pristupa privatnim delovima neke klase.

Uzrok:

- Klase koje često komuniciraju dovode do ovog problema.

Rešenje:

- Premeštanje metoda.
- Premeštanje atributa.
- Izdvajanje klase.
- Sakrivanje posrednika.
- Transformacija bidirekcione asocijacije u jednodirekcionu asocijaciju (jedna klasa koristi funkcionalnosti druge klase umesto unakrsno).
- Zamena delegacije nasleđivanjem.

Isplativost:

- Bolja organizacija.
- Jednostavniji kod.

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

(eng. Comments)
„Komentari imaju dobar miris, ali se često koriste kao dezodorans za kod koji zaudara“
Simptomi:

- Metod ima previše komentara.

Uzrok:

- Komentari se dodaju kada autor shvati da kod nije baš razumljiv.

Rešenje:

- „Najbolji komentar je dobro ime za metodu ili klasu“ (izdvojiti deo algoritma u zasebnu metodu).
- Izdvajanje promenljivih. (delovi koda koji su teški za razumevanje se izdvajaju u zasebnu promenljivu čiji sam naziv objašnjava šta se računa).

- Izvlačenje metode.
- Preimenovanje metode.
- Dodavanje tvrdnji (assert).

Isplativost:

- Kod postaje intuitivniji.

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

Sistematizacija refaktorisanja podrazumeva ujednačeno opisivanje tehnika.
Svaka tehnika mora da ima:

- Ime, veoma bitan izbor (kao kod uzoraka).
- Sažet opis slučaja u kome se primenjuje.
- Motivacija, šta se može očekivati nakon primenjene tehnike.
- Opis izvođenja tehnike.
- Primer (sa evetualnom vizualizacijom).

163 Navesti bar 5 grupa tehnika refaktorisanja.

Sve grupe refaktorisanja:

- **Metode komponovanja koda:** Koriste se za izbacivanje duplikata u kodu i povećavanje fleksibilnosti koda kada dolazi do dodavanja novih funkcionalnosti. Primeri: izdvajanje metoda, umetanje metoda, zamena metoda klasom, ...
- **Premeštanje koda između objekata:** Koriste se za premeštanje poslova između objekata ili skrivanje funkcionalnosti. Primeri: premeštanje metoda, premeštanje atributa, izdvajanje klasa, izbacivanje posrednika, ...
- **Organizovanje podataka:** Koriste se u okviru klasa da povećaju njenu primenljivost u drugim situacijama. Primeri: transformacija bidirekcione asocijacije u direkcionu asocijaciju, enkapsulacija, ...
- **Pojednostavljivanje uslovnih izraza:** Koriste se da bi se logika u okviru nekog grananja uslova pojednostavila. Primeri: dekompozicija uslova, ubacivanje prazne klase, dodavanje tvrdnji, ...
- **Pojednostavljivanje pozivanja metoda:** Pojednostavljuje korišćenje metoda, a samim tim povećava razumljivost. Primeri: preimenovanje metoda, preimenovanje atributa, dodavanje parametarskog objekta, ...
- **Razrešavanje uopštavanja:** Koriste se uglavnom za pomeranje poslova klasa u okviru hijearhije nasleđivanja. Primeri: prebacivanje atributa u natklasu, prebacivanje metode u natklasu, ...
- **Velika refaktorisanja.**

164 Navesti bar 5 tehnika refaktorisanja.

1. Izdvajanje metoda
2. Umetanje metoda
3. Zamena metoda klasom
4. Premeštanje atributa
5. Premeštanje metoda
6. Dodavanje tvrdnji
7. Ubacivanje prazne klase
8. Dodavanje parametarskog objekta
9. Dekompozicija uslova,
10. Transformacija bidirekcione asocijacije u direkcionu asocijaciju

165 U kojim slučajevima refaktorisanje može biti značajno otežano?

- Ako kod ima neku grešku (bag), onda je refaktorisanje otežano;
- Refaktorisanje velikog, nefaktorisanog koda može da bude veoma teško, a ponekad se više isplati krenuti od nule nego refaktorisati kod;
- Refaktorisanje može da bude otežano i u:
  - prisustvu baze podataka;
  - prisustvu spoljnog interfejsa.

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

Menjanje struktura podataka ima široki uticaj. Migracija podataka je veoma neugodna. Treba menjati i bazu i kod istovremeno. Jedan od principa Agilnog razvoja je ne praviti bazu podataka pre nego što je to potrebno, već prvo pokušati sa datotekama.

„Ne praviti bazu podataka pre vremena“

167 Zašto može biti otežano refaktorisanje spoljnog interfejsa neke klase?

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

„Ne objavljivati javni interfejs pre vremena“

168 U kojim slučajevima refaktorisanje može da ne predstavlja dobro rešenje?

- Ako su problemi višestruki i zavisni.
- Ako se ne mogu primenjivati mali koraci bez remećenja postojećeg ponašanja.
- Ako postojeći kod ne radi.
- Ako je blizu kraj roka.

169 Kakav je odnos refaktorisanja i performansi softvera?

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

- Stepen negativnog uticaja na performanse je obično daleko niži od ostvarenog pozitivnog uticaja na dizajn.
- Pisanje struktuiranog koda je daleko efikasnije nego pisanje optimizovanog koda.
- Troškovi razvoja su često važniji od troškova eksploatacije.

Metode:

- **Dobro struktuiranje sa definisanjem ciljnih performansi za svaku komponentu:** Nakon dostizanja funkcionalnosti pristupa se optimizaciji u meri u kojoj je potrebno za dostizanje ciljnih uslova.
- **Kontinualno staranje o performansama:** Tokom čitavog razvoja se svi 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.

170 Šta su bagovi

Propusti u razvoju softvera se često nazivaju greškama ili bagovima. Generalno greška ima malo šire značenje tj. greška obuhvata sve propuste koji mogu da se dese u bilo kojoj fazi razvoja softvera. Bag je pojam koji se odnosi na grešku u okviru softvera, u smislu da stvara probleme u funkcionisanju softvera kao završnog proizvoda.

171 Navesti jednu klasifikaciju bagova i objasniti je.

- Nekonzistentnost u korisničkom interfejsu;
- Neispunjena očekivanja;
- Slabe performanse;
- Padovi sistema (programa) ili oštećenja podataka.

Padovi softvera i ostećenja podataka: Predstavlja 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.

172 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 je i u planiranju.
Primeri:

- Svi programi za *MS Windows* koriste prečicu *Ctrl + F* za traženje. Program Outlook koristi tu prečicu za prosledivanje poruke (forward).
- *MS Word* za OS/2. (Ako hocemo da izađemo iz programa a nismo sačuvali izmene, OS/2 kaže neću da sačuvam po podrazumevanim podešavanjima, a Microsoft word kaže hoću).

Najčešći uzroci:

- Propusti pri projektovanju korisničkog interfejsa ili čak ustanovljavanja osnovnih koncepata softvera.
- Propusti pri planiranju ili sprovođenju testova.

173 Šta su neispunjena očekivanja i kakve uzroke i posledice imaju?

Dobijanje neočekivanog (pogrešnog) rezultata je jedan od najneugodnijih bagova.
Tipovi:

1. Neispravan pozitivan rezultat;
2. Neispravan negativan rezultat;
3. Neispravan rezultat izračunavanja.

Najčešći uzroci:

- Greške u komunikaciji, kada razvojni tim ne razume ispravno potrebe klijenta.
- Greške u projektovanju, kada projektanti naprave propuste.
- Greške u kodiranju, kada programeri naprave greške u kodu.

174 Objasniti problem slabih performansi i moguće uzroke.

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

- Greške u proceni opterećenja;
- Greške u proceni raspoloživih resursa;
- Greške u projektovanju rešenja;
- Greške u kodiranju rešenje.

175 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;
- ...

176 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;
- Upravljanje rizicima.

177 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.

178 Navesti bar 6 osnovnih pravila za debagovanje.

Otklanjanje grešaka se uobičajno naziva **debagovanje**.
Pravila:

1. Razumeti sistem;
2. Navesti sistem na grešku;
3. Najpre posmatrati, pa tek 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.

179 Objasniti pravilo debagovanja „Razumeti sistem“.

„Da bi se razumeo sistem, potrebno je dobro razumeti sistem“

Razumevanje sistema nije isto što i razumevanje problema, već je potrebno razumevanje prostora u kome postoji problem. Primer: programski modul, aplikacija, ceo sistem.

Ovo pravilo predstavlja preduslov za razumevanje problema (baga). Da bi se u nekom sistemu pronašla greška i uzroci te greške, potrebno je da se taj sistem poznaje.
Osnovni aspekti razumevanja sistema:

- Čitanje upustava za sistem i komponente koje ga čine (napomena: upustva mogu biti neispravna).
- Detaljno čitanje upustava (nije dovoljno samo razumeti koncepte).
- Razumeti šta je očekivano normalno ponašanje.
- Poznavati alate.
- Obratiti pažnju na detalje.

180 Objasniti pravilo debagovanja „Navesti sistem na grešku“.

Razlozi za ponavljanje greške:

- Da bi mogla da se posmatra greška.
- Da bi mogla da se posveti pažnja uzrocima.
- Da bi moglo da se proveri da li je greška otklonjena.

Načini navođenja softvera na grešku:

- **Ponoviti postupak:** Pouzdan način da se greška ponovi je da se postupak ponovi pred publikom. Dokumentovati korake koji vode ka 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, onda se isplati automatizovati ponavljanje.
- **Ne simulirati grešku, nego uslove u kojima se ispoljava:** Simuliranje mehanizma suviše bliskih uzorku može da zaobiđe neispravan deo koda.
- **Ne odbacivati problem kao „nemoguć“.**
- **Čuvati napravljene alata (možda opet zatrebaju).**

Šta ako se problem ispoljava samo povremeno (primer: svaki 5, 10 ili 100 put)? To znači da nisu dovoljno dobro poznate okolnosti ispoljavanja. Potrebno je razmotriti razne nekontrolisane uslove:

- neinicijalizovane podatke;
- slučajno generisane podatke;
- ulazne podatke;
- vezanost za trenutak ili trajanje izvršavanja;
- sinhronizaciju niti;
- spoljašnje uređenje.

Kontrolisanje uslova često vodi do nemogućnosti ponavljanja greške. To znači da jedan od kontrolisanih uslova u nekom specifičnom stanju (različitom od kontrolisanog) proizvodi problem.

Nekada takve uslove nije moguće kontrolisati, ali je moguće učiniti proizvoljnijim. To može da pomogne u razumevanju uslova nestajanja problema, ali može i da proizvede ove greške.

Šta ako smo sve pokušali, ali problem se i dalje ispoljava samo povremeno? Problem nije „svojeglav“ i ima svoj precizan uzrok. Potrebno je pokušati ispunjavanje ciljeva ponavljanja bez samog ponavljanja (kako?).

Ako je teško ponoviti problem, onda je potrebno pažljivije posmatrati slučajeve kada se ponavljanje ostvari. Potrebno je prikupljati i analizirati informacije uočavanjem uslova koji su UVEK povezani ili NIKADA nisu povezani sa ponavljanjem problema. Da bi se (relativno) pouzdano ustanovilo da je problem rešen, potrebno je sekvencu ponoviti statistički značajan broj puta.

181 Objasniti pravilo debagovanja „Najpre posmatrati pa tek onda razmišljati“.

Razmišljanje bez dovoljno informacija je suštinski promašaj u pristupu i može da vodi ka iskrivljenom tumačenju podataka. **Kako?**

- **Dok se greška ne vidi, ne donositi zaključke.**
- **Posmatrati detalje:** Svako posmatranje donosi saznanja o uslovima ili mehanizmu nastajanja problema.
- **Praviti alate koji ističu problematične uslove ili problematičan deo koda:**
  - Neki vidovi takvih alata mogu biti posmatrani još u fazi projektovanja. Primer: trajno zapisivanje informacija o problemima, mogućnost automatskog razlikovanja poruka radi lakšeg pretraživanja i analiziranja.
  - Dodavati 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 deo sistema“. Testiranje i menjanje koda može da ima neugodne posledice, od promene ponašanja pa do prikrivanja grešaka. Sve izmene i testove je potrebno praviti u sasvim mali koracima, kako bi se smanjila mogućnost prikrivanja problema.
- **Praviti pretpostavke samo radi boljeg fokusiranja pri traženju:** Ako se uvede pretpostavka o uzroku greške, ne koristiti je za „otklanjanje greške“ već samo za fokusiranje traženja greške na neki aspekt problema. Izuzetak: Neke greške su 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.

182 Objasniti pravilo debagovanja „Podeli pa vladaj“.

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

- **Šužavati oblast traženja uzastopnih aproksimacija:** Postavljati hipoteze o mestu ili uslovima pojavljivanja problema, a zatim testovima ustanovljavati hipoteze. Birati hipoteze tako da što uspešnije dele mogući prostor problema.
- **Pronaći prave okvire prostora za traženje problema:** Pre sužavanja problema je potrebno postaviti dovoljno široke početne okvire. Neophodno je pouzdano proveriti da li su početni okviri dovoljno široki. Ovo je posebno problematično ako je okvir potencijalno širi od računarskog sistema.
- **Prepoznati na kojoj strani je greška:** Da bi hipoteza bila korisna, potrebno ju je postaviti tako da deli prostor na dva dela, gde se jasno vidi u kojem od ta dva dela se nalazi problem.
- **Koristiti test uzorke koji se lako uočavaju:** Često nije lako uočiti greška na živim podacima, pa je potrebno napraviti veštačke uzorke za koje je poznato kako treba da izgleda rezultat. Poželjno je da uzorci budu što manji i da mehanizam provere ispravnosti ponašanja bude što jednostaniji.
- **Početi od greške, pa ići prema uzorcima:** Obično je bolje traženje započeti od mesta pojavljivanja greške i odatle se kretati ka uzrocima. Mogućih uzroka obično ima previše da bi se proveravali jedan po jedan.
- **Popraviti prepoznate greške i nastaviti dalje:** Bagovi često idu zajedno. Ako je jedan bag pronađen, to ne znači da je i jedini.
- **Otkloniti šum:** Ukoliko je sistem dinamičan, korisno je privremeno „smiriti“ delove sistema koji proizvode visok nivo dinamičnosti da ne bi odvlačili pažnju ili vreme. Bagovi mogu da se dešavaju lančano tj. jedan bag može da bude uzrok drugom bagu itd. Privremenim isključenjem delova koda se mogu prikriti neki bagovi koji su posledica inicijalnog, izvornog baga.

183 Objasniti pravilo debagovanja „Praviti samo jednu po jednu izmenu“.

Ako se načine dve izmene odjednom, onda je teško ispravno oceniti njhov efekat:

- Možda je jedna izmena rešila problem, a druga ne.
- Možda je samo jedna izmena dovoljna.
- Možda su obe izmene loše (a ovo je teže zaključiti u tom slučaju).
- Možda bi one mogle da reše problem, ali nisu ispravne.

Saveti?

- **Izolovati ključni faktor:** „Koristiti pušku, a ne sačmaru“. Izmene koje se prave treba da budu izolovane i lokalizovane. Više izmena koje nisu lokalizovane značajno otežavaju uočavanje posledica. Ako je zaista uočen problem, onda je dovoljna jedna izmena, a ako nije, onda je potrebno dalje posmatrati.

- **Biti uzdržan:** Kad se uoči da se nešto dešava, potrebno je zastati i dobro sačekati sa reagovanjem dok se svi simptomi prikažu, kako se ne bi doneo dvosmislen zaključak. U suprotnom je moguće da se radi o nagađanju i ragovanju na osnovu nagađanja.
- **Menjati i isprobavati jednu stvar:** Ako pokušaj popravke ili testiranja nije pomogao, prvo je potrebno da se programski kod vrati u prvobitno stanje, pa tek onda da se prave nove izmene.
- **Porediti sa ispravnim slučajem:** Poređenje uslova pri kojima nastaje problem i uslova pri kojima nema problema je jedan od najkorisnijih postupaka pri razrešavanja bagova. Porediti: kod, tragove izvršavanja, dnevnike grešaka, itd.
- **Ustanoviti šta je izmenjeno od poslednje poznate situacije u kojoj je ponašanje bilo ispravno:** Nekada neizgled nebitna izmena može da ima značajan uticaj na sistem. Ako je sistem nekada ispravno radio, onda je potrebno lokalizovati interval u kojem je on ispravno radio i onda lokalizovati izmene koje su napravljene na kraju tog intervala.

184 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.
Kako?

- **Zapisivati šta se radi, kojim redom i koje su uočene posledice:** Pisani tragovi o aktivnostima imaju veliki značaj za kasnije lokalizovanje u vremenu.
- **Važno je razumeti da svaki od detalja može biti značajan:** Pisani trag nikada nije suviše detaljan.
- **Povezivati događaje:** Međusobnih povezivanjem simptoma i ishoda (uspešnih ili neuspešnih) olakšava uočavanje uzroka problema.
- **Dnevnici sa tragovima izvršavanja su važni za testiranje:** Na primer, Ostavljanje tragova pri aktiviranju delova koda ili obavljanju određenih poslova kao što je otvaranje datoteka.
- **Potrebno je zapisivati, koliko god to delovalo teško:** „Najkraća olovka je duža od najdužeg pamćenja“

185 Objasniti pravilo debagovanja „Proveravati i naizgled trivijalne stvari“.

Ponekad su rešenja problema veoma jednostavna, ali je to teško zaključiti.
Saveti?

- **Dovoditi u pitanje prethodno uvedene pretpostavke:** Ako smo pretpostavili da je neka komponenta ispravna, to ne znači da ona to i jeste. Nikada ne treba biti ubeđen da su pretpostavke ispravne, posebno ako su u središtu nekog neobjašnjenog problema.
- **Početi od početka:** Da li su uslovi za obavljanje posla ispunjeni? Primer:
  - 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 ispravno raditi.
- **Proveriti alate:** Ako smatramo da alat nešto radi na neki način, to ne znači da nije potrebno i proveriti to, posebno ako postoji problem koji ne razumemo. Ako alat pokazuje da je sve u redu, a mi vidimo da nije, onda je alat možda neispravan ili ne umemo da ga koristimo.

186 Objasniti pravilo debagovanja „Zatražiti tuđe mišljenje“.

Ne ustručavati se da tražite tuđu pomoć! Saveti?

- **Uključiti sveže snage:** Mnogo vremena provedeno uz neki sistem stvara implicitne podsvesne pretpostavke koje ometaju objektivno rasuđivanje. Neko sa strane ko nije opterećen „nebitnim“ detaljima će često lakše uočiti uzrok problema.
- **Za skoro svaku oblast postoje eksperti:** Umesto da gubimo sate (dane, nedelje) mnogo je lakše i jeftinije zatražiti pomoć eksperta. Eksperti znaju gde je potrebno „udariti čekićem“.
- **Pomoć je dostupna na raznim stranama:** Sasvim je moguće da je neko već imao taj problem i vrlo je moguće da je takav problem već dokumentovan. Potražiti „vodiče kroz probleme“ (troubleshooting guides). Potražiti diskusione grupe na internetu.
- **Ne valja biti suviše ponosan:** Greške se dešavaju. Ponos treba da pomogne da se istraje u rešavanju problema, ali ponos ne treba da sprečava da se potraži pomoć. Ipak, nekada ni drugi nisu u pravu (računati i sa tim).
- **Izveštavati o simptomima, a ne o pretpostavkama:** Ako uključimo nekoga u rešavanje problema, potrebno je snabdeti samo pouzdanim informacijma, u suprotnom se prave preduslovi da i pomagač usvoji naše previde.

187 Objasniti pravilo debagovanja „Ako nismo popravili bag, onda on nije popravljen“.

Ako simptomi problema nestanu sami od sebe, to ne znači da je problem rešen. Uobičajno je da se pojavi ponovo kada nam to najmanje bude odgovaralo.
Saveti?

- **Pogledati da li je zaista popravljen:** Ako smo pratili ranija pravila, onda znamo kako da ponovimo problem.
- **Kada pomislimo da smo rešili problem, neophodno je da proverimo da li smo ga rešili načinjenim popravkama ili je nešto drugo u pitanju:** Pokušajmo da ponovimo problem bez ispravke: Ako se tada pojavljuje, a sa ispravkom ne, onda smo rešili problem. U suprotnom, su u pitanju izmenjene okolnosti i sasvim je verovatno da je naše „rešenje“ beznačajno.
- **Problemi nikada ne nastaju sami od sebe:** Ako je problem nestao „sam od sebe“ to samo znači da nismo dovoljno upoznali okolnosti pod kojima se pojavljuje, ili su se te nepoznate okolnosti promenile i sasvim je verovatno da će se okolnosti promeniti u nekom trenutku i da će se problem ponovo pojaviti.
- **Popraviti uzroke problema:** Ako pregori osigurač, zamenom osigurača se samo privremeno rešava problem. Potrebno je pronaći uzroke pregorevanja.
- **Popraviti razvojni proces:** Ako je problem nastao, znači da smo napravili grešku u procesu. Greška u procesu zahteva popravku procesa.

188 Navesti najvaznije 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 (i testiranju).

- Pravljenje pretpostavki (assert);
- Ostavljanje tragova pri izvršavanju (trace);
- Interni opisi stanja;
- Komentarisanje značajnih odluka i mesta u kodu;
- Testiranje jedinica koda.

Spoljašnje tehnike i alati: Različiti alati koji se koriste tokom razvijanja i debagovanja programa.

- Debager;
- Alati za praćenje verzije koda;
- Alati za podršku i praćenje komunikacije;

- Alati za automatizovanje prevljenja dokumentacije;
- ...

189 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 obično važe. Pretpostavke obično prekidaju ili privremeno obustavljaju izvršavanje koda sa odgovarajućim obaveštenjem.

Većina biblioteka ima različite oblike makroa *assert* koji proverava da li je dati uslov zadovoljen. Ako uslov nije zadovoljen, onda se program prekida. Ako nije verzija za debagovanje, onda se provera uopšte ne uključuje u kod. Primer:

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

Kako?

- **Navoditi sistem na sasvim jednostavne uslove:** Ako nije zadovoljen neki složen uslov, onda kako znati koji njegov deo nije zadovoljen? Složene uslove je bolje podeliti na više jednostavnijih.
- **Proveravati da li argumenti potprograma zadovoljavaju neophodne uslove:** Ako ne zadovoljavaju, onda to znači da su neispravne vrednosti.
- **Proveravati da li rezultat funkcije zadovoljava neophodne uslove u odnosu na argumente:** Ako ne zadovoljava, to znači da postoji greška u implementaciji funkcije.
- **U složenim potprogramima proveravati međurezultate.**

U okviru C++-a i QT bibliote se mogu naći različite pretpostavke.

Kada koristiti pretpostavke? Pretpostavke mogu znatno da pogoršaju performanse sistema zbog čega se u verziji za isporučivanje izbacuju. Ponekad prilikom izvršavanja, kada program izbaci (throw) izuzetak (exception) kao posledica greške, nije uvek moguće uhvatiti (catch) tj. obraditi taj izuzetak. Tada jednostavno ne sme doći to takve greške. U tim slučajevima ima smisla koristiti pretpostavke.

Izuzeci su za korisnike, a pretpostavke su za programere. Pretpostavke se koriste kada program ne sme da dozvoli da se nešto desi što uopšte nema smisla. Primer: $\sqrt{4} = 3$.

190 Objasniti tehniku ostavljanja tragova pri izvršavanju kao prevenciju nastajanja bagova.

U zavisnosti od načina implementacije dnevnika, pravi se makro ili šablon (ili više makroa ili šablona) koji ispisuju tekuće stanje.

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

Log vs Trace:

Event logging	Software tracing
Consumed primarily by system administrators	Consumed primarily by developers
Logs "high level" information (e.g. failed installation of a program)	Logs "low level" information (e.g. a thrown exception)
Must not be too "noisy" (containing many duplicate events or information is not helpful to its intended audience)	Can be noisy
A standards-based output format is often desirable, sometimes even required	Few limitations on output format
Event log messages are often localized	Localization is rarely a concern
Addition of new types of events, as well as new event messages, need not be agile	Addition of new tracing messages *must* be agile

191 Objasniti komentarisanje koda kao tehniku za prevenciju nastanaka bagova.

Komentarisanje je vid dokumentovanja koraka koji su doveli do toga da program izgleda kako izgleda.
Komentari 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.

192 Objasniti testiranje jedinica koda kao tehniku za prevenciju nastanaka bagova.

Pogledati pitanja:

- Šta može biti jedinica koda koja se testira? 130;
- Šta može biti predmet testiranja jedinice koda? 131.

193 Navesti osnovne tehnike upotrebe debagera.

- Izvršavanje korak po korak;
- Postavljanje tačaka prekida (breakpoint) u kodu ili podacima;
- Praćenje vrednosti promenljivih;
- Praćenje lokalnih promenljivih;
- Praćenje stanja steka;
- Praćenje na nivou instrukcija i stanja procesora.

194 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 tačku prekida.

195 Objasniti tehniku upotrebe debagera Postavljanje tačaka prekida.

Tačka prekida (eng. breakpoint) 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 (primer: provera vrednosti promenljivih u toj tački).

196 Objasniti tehniku upotrebe debagera Praćenje vrednosti promenljivih.

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

197 Objasniti tehniku upotrebe debagera Praćenje lokalnih promenljivih.

Komandom *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.

198 Objasniti tehniku upotrebe debagera Praćenje stanja steka.

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

199 Objasniti tehniku upotrebe debagera Praćenje na nivou instrukcija i stanja procesora.

Neki debageri nam omogućavaju da pratimo instrukcije procesora. U okviru debagera GDB imamo komandu *(gdb) layout asm* koja prebacuje izgled TUI (text user interface) izgled u formu (isto TUI) koja prikazuje instrukcije procesora (registri, izvor, ...). [19]

200 Šta je dizajn softvera? Šta je arhitektura softvera? Objasniti sličnosti i razlike.

Dizajn softvera i arhitektura softvera su strukture softverskog sistema. Razlika je u tome što je arhitektura softvera posmatrana na relativno visokom nivou. Arhitektura se odnosi više na spoljni deo softvera, tj. kako komponente komuniciraju međusobno i predstavlja apstrahciju celog sistema. To znači da je svaka arhitektura softvera dizajn softvera, ali ne važi obrnuto. Neformalno rečeno, arhitekturu softvera ne zanima implementacija.

"Arhitektura predstavlja značajne odluke o dizajnu koje daju oblik sistemu, gde je značajnost određena cenom pravljenja izmena"

— Grady Booch, 2001

Pri pravljenju arhitekture softverskog sistema preduzimaju se neke specifične aktivnosti i uzimaju u obzir specifični aspekti problema:

- Veliki značaj širine posmatranja;
- Manji značaj pojedinosti.

201 Šta čini dizajn softvera? Navesti osnovne pojmove dizajna

Elementi:

1. komponente,
2. implementacija,
3. struktura podataka,
4. infrastruktura,
5. pomoćni alati,
6. i sve drugo što je deo razvojnog procesa ...

Osnovni pojmovi:

1. **apstrahovanje**
2. **dekompovanje**

202 Šta je apstrakcija? Objasniti ulogu apstrakcije u dizajnu softvera.

Apstrahovanje je razmatranje opštih karakteristika problema i rešenja uz zanemarivanje pojedinosti. Podrazumeva rešavanje problema na konceptualnom nivou. Ima sličnosti sa mehanizmom indukcije: Posmatranjem specifičnih slučajeva se teži ka pravljenju opšteg modela. Za apstrakciju se može reći da je **uopšteno rešenje**.

Primer: Niz bitova na disku je datoteka.

203 Šta je dekompozicija? Objasniti ulogu dekompozicije u dizajnu softvera.

Dekompovanje 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, gde se dekompozicijom apstraktnog sistema dolazi do njegove konkretne strukture. Ima sličnosti sa mehanizmom indukcije: Posmatranjem i razlaganjem opšteg slučaja se teži ka pravljenju konkretnijeg (preciznijeg) modela. Za dekompoziciju se može reći da je **razloženo rešenje**.

Primer: Jelovnik u restoranu se dekomponuje:

- jelovnik za glavno jelo;
- jelovnik za pića;
- jelovnik za dezert;
- ...

204 Navesti i ukratko objasniti 4 osnovna pojma dizajna softvera.

- **Modularnost:** Složene celine se dele na manje module. Na višem nivou apstrakcije se posmatra celina, a na nižem dekompovani 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 osnovnih kriterijuma pri dekompoziciji. Svaka komponenta mora da ima jasno prepoznatu i zaokruženu odgovornost.
- **Interfejsi:** Predstavlja tačku vezivanja komponente. Može da se posmatra kao deklaracija ponašanja komponente.

205 Navesti i ukratko objasniti bar 6 ključnih principa dizajna softvera.

1. **Princip jedinstvene odgovornosti:** Jedan modul bi trebalo da ima jednu namenu i samo jedan razlog za menjanje.
2. **Razdvajanje odgovornosti:** Različite komponente bi trebalo da imaju različite odgovornosti. Nijedna odgovornost ne bi trebalo da bude podeljena na više odgovornosti.
3. **Princip minimalnog znanja:** Jedna komponenta ne mora da zna kako druga funkcioniše. Dovoljno je da zna kako da je koristi.
4. **Princip inverzne zavisnosti:** Apstrakcija ne sme da zavisi od pojedinosti, a pojedinosti moraju da zavise od apstrakcije, iako se u procesu projektovanja apstrakcija oblikuje na osnovu pojedinosti.
5. **Princip acikličnih zavisnosti:** Izbegava se ciklična zavisnost komponenti. Složene zavisnosti otežavaju razumljivost i održavanje.
6. **Princip stabilne zavisnosti:** Smer zavisnosti bi trebalo da se poklapa sa smerom porasta stabilnosti. Nije dobro da drugi moduli zavise od manje stabilnih modula.
7. **Princip stabilne apstrakcije:** Modul bi trebalo da bude apstraktan koliko i stabilan. Drugi moduli bi trebalo da zavise od apstraktnih i stabilnih modula.
8. **Izbegavanje ponavljanja:** Nijedna funkcionalnost ne bi smela da se implementira više puta. Ako ima ponavljanja, to znači da rešenje nije dovoljno apstraktno.
9. **Izbegavanje suvišnog posla:** Predvideti samo ono što je danas potrebno, a ono što će možda biti potrebno nije potrebno.

206 Objasniti odnos pisanja koda i dizajniranja.

Pretpostavimo da je izvorni kod predstavlja dizajn softvera. Pisanje koda je u tom slučaju ekvivaletno dizajniranju. Tada je izgradnja (prevođenje) koda proces implementacije. Cena izgradnje je izuzetno niska (sve niža i niža sa bržim i jeftinijim računarima). Cena dizajniranja je relativno niska u odnosu na druge vrste projekata i dobijenu složenost.

Problem: Dizajn softvera lako i brzo postaje veoma složen. Programiranje je mnogo više od kodiranja. Programiranje obuhvata i projektovanje. Razvoj softvera je i zanatska i inžinjerska disciplina.

Kako raditi?

- **„Projekat pre koda“:**
  - Daje jasnu sliku projekta pre programiranja;
  - Omogućava programerima da imaju jasniju sliku šta prave;
  - Nudi jasnu i stabilnu sliku projekta;
  - Često se menja tokom razvoja, čak i potpuno;
  - Predstavlja ozbiljan, ali često neažuran vid dokumentacije.
- **„Projekat je kod“:**
  - Iskusni programeri mogu da istovremeno prilagode i kod projektovanju i projekat načinu kodiranja;
  - Štedi se vreme na projektovanju i može brže da se započne sa implementacijom pojedinačnih modula.
  - Može da dovodi do otežanog povezivanja komponenti;
  - Ne postoji jasna slika projekta u ranim fazama razvoja.

207 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 on pravi, onda je samo umereno detaljan.
- **Što je dizajn višeg nivoa, to je neophodno 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:** Šta god to bilo u konkretnom slučaju ...

Planiraju se:

- komponente,

- njihovi interfejsi,
- njihovi međusobni odnosi.

To obično spada u arhitekturu. Ne planira se detaljno implementacija komponenti (to obično spada u dizajn).

„Projektovanje“ se praktično izjednačava sa „oblikovanjem arhitekture“. Na taj način se izbegavaju problemi oba pristupa. Dobija se jasna velika slika dekompozicije sistema na komponente i jasno se definišu njihovi interfejsi. Ne gubi se suviše vremena na projektovanje pojedinosti u ranim fazama razvoja, a koje će se kasnije često menjati. Što je sistem složeniji, to se arhitektura pravi na više različitih nivoa apstrakcije, tj. „projektovanje“ mora da se radi detaljnije.

208 Navesti i ukratko objasniti obaveze softverskog arhitekte.

1. **Razumevanje svih aspekata razvoja:** Odnosi se na poznavanje, planiranje i praćenje razvojnog procesa, i uzimanje u obzir sva praktična ograničenja.
2. **Razumevanje sopstvene uloge:** Odnosi se na preuzimanje odgovornosti za ključne odluke bez zalaženja (iz ugla arhitekture) u nebitne detalje.
3. **Komuniciranje sa ulagačima:** Arhitektura mora da zadovoljava potrebe ulagača.
4. **Izvođenje rešenja iz poslovnih potreba:** Odnosi se na prepoznavanje i razumevanje potreba. Arhitektura je „slika“ poslovnog okruženja.
5. **Uticanje na zahteve:** Odnosi se na prepoznavanje kompromisa i predočavanje ulagačima. Pomaganje ulagačima da donesu ispravne odluke.
6. **Upravljanje rizicima i promenama:** Odnosi se na iterativno unapređenje i prilagođavanje arhitekture.
7. **Ponovljena upotreba:** Odnosi se na upotrebu postojećih dobara, čime se se štedi i novac.
8. **Dobro odmeravanje učešća:** Arhitekta mora da donese odluke u svom domenu. Ne sme da izađe van svog domena.
9. **Preciziranje arhitekture prema tehnološkim ograničenjima:** Arhitektura se uvek implementira u nekom tehnološkom kontekstu.
10. **Uvažavanje opštih okvira:** Svaka konkretna arhitektura je samo deo šireg poslovnog konteksta. Ona mora da se slaže sa ostalim elementima okruženja.

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

Aspekti arhitekture:

- **Šta?** Zahtevi koje softver mora da zadovolji.
- **Zašto?** Ključne odluke na osnovu kojih se oblikuje arhitektura.
- **Kako?** Komponente, interfejsi, odnosi. Na nižem nivou i dizajn i implementacija rešenja.

Ključni uticaji (međusobno suprostavljene 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);
- **Estetika:** Svi oni neformalni (iracionalni?) aspekti rešenja koji ga čine kvalitetnijim (nešto manje važna i jaka sila).

210 Prazno (radi usaglašavanja odgovora sa ispitnim pitanjima).

211 Šta su kohezija i spregnutost u kontekstu razvoja softvera?

Kohezija (eng. cohesion) je stepen međusobne povezanosti elemenata jednog modula.

Spregnutost (eng. coupling) je stepen međusobne povezanosti elemenata dveju različitih modula.

Dobro dizajniran softver se odlikuje visokom kohezijom i niskom spregnutošću komponenti. [20] [21]

212 Navesti vrste kohezije u kontekstu razvoja softvera.

- funkcionalna (Functional)
- sekvencijalna (Sequential)
- komunikaciona (Communicational)
- proceduralna (Procedural)
- vremenska (Temporal)

- logicka (Logical)
- koincidentna (Coincidental)

213 Objasniti funkcionalnu koheziju u kontekstu razvoja softvera.

Funkcionalna kohezija predstavlja povezanost delova jednog modula na osnovu međusobne funkcionalne zavisnosti u cilju ostvarivanja funkcije za koju je komponenta odgovorna. Svi delovi su prikupljeni sa jednim istim osnovnim ciljem. Svaki deo je potreban i nijedan deo nije višak. Predstavlja idealnu situaciju.
Napomena: Sledeći primer i primeri koju slede se odnose na koheziju klasa. Ove primere možemo posmatrati kao analogiju, gde klase igraju ulogu modula, a komponente igraja ulogu metoda.

```
class Plane
{
public:
    void takeoff();
    void fly();
    void land();
};
```

214 Objasniti sekvencijalnu koheziju u kontekstu razvoja softvera.

Koehezija je **sekvencijalna kohezija** ako su elementi komponente projektovani tako da izlaz jedne komponente predstavlja ulaz u drugu komponentu. Ne postoji puna funkcionalna zavisnost. Delovi sekvence potencijalno obavljaju različite poslove. Postoji potencijalno otvoren problem odgovornosti u odnosu na celovit posao. Ovaj tip kohezije se sasvim prirodno pojavljuje u funkcionalnim jezicima. Primer: protok podataka.

215 Objasniti komunikacionu koheziju u kontekstu razvoja softvera.

Kohezija je **komunikaciona kohezija** ako su elementi komponente prikupljeni zato što koriste iste podatke. Ne postoji funkcionalna zavisnost i delovi obično obavljaju različite poslove. Obično nisu dobro razdvojene odgovornosti:

- Jedna komponenta ima više odgovornosti;
- Jedna komponenta je podeljena na više kompenenti.

Primer: Ažurira se slog u bazi podataka i šalje se pisaču.

216 Objasniti proceduralnu koheziju u kontekstu razvoja softvera.

Kohezija je **proceduralna kohezija** ako su elementi komponenti prikupljeni zato što se koriste pri obavljanju nekog celovitog posla. Verovatno ne postoji funkcionalna zavisnost i komponente obavljaju potpuno raznorodne poslove (koji se obično obavljaju jedan za drugim). Obično nisu dobro razdvojene odgovornosti:

- Jedna komponenta ima više odgovornosti;
- Jedna odgovornost je podeljena na više komponenti.

Primer: Otvaranje datoteke, proveravanje ispravnosti, čitanje, ...
Primer: Pravljenje torte :)

```
class BakeCake
{
public:
    void addIngredients();
    void mix();
    void bake();
};
```

217 Objasniti vremensku koheziju u kontekstu razvoja softvera.

Kohezija je **vremenska kohezija** ako su elementi komponente prikupljeni zajedno zato što se koriste u „istom“ vremenskom periodu. Verovatno ne postoji funkcionalna zavisnost i često delovi obavljaju raznorodne poslove. Obično nisu dobro razdvojene odgovornosti:

- Jedna komponenta ima više odgovornosti;
- Jedna komponenta je podeljena na više kompenenti.

Primer: Različiti elementi inicijalnog sistema.

```
class InitFuns
{
public:
    void initDisk();
    void initPrinter();
    void initMonitor();
};
```

218 Objasniti logičku koheziju u kontekstu razvoja softvera.

Kohezija je **logička kohezija** ako su elementi komponente prikupljeni zajedno zato što imaju logički sličnu (ili čak istu) ulogu u sistemu. Najčešće ne

postoji funkcionalna zavisnost i delovi najčešće obavljaju raznorodne poslove koji mogu predstavljati alternativu jedni drugima. Obično nisu dobro razdvojene odgovornosti:

- Jedna komponenta ima više odgovornosti;
- Jedna komponenta je podeljena na više kompenenti.

Primer: Računanje površina različitih oblika.

```
class AreaFuns
{
public:
    double circleArea();
    double rectangleArea();
    double triangleArea();
};
```

219 Objasniti koincidentnu koheziju u kontekstu razvoja softvera.

Kohezija je **koincidentna kohezija** ako su elementi kohezije međusobno nespregnuti ili su veoma slabo spregnuti. Ne postoji nikakva (ili postoji sasvim niska) funkcionalna zavisnost. Delovi obavljaju raznorodne poslove i uopšte nisu razdvojene odgovornosti. Ovo je najniži nivo kohezije. Primer: Vraćanje datuma uz inicijalizaciju pisača.

```
class MyFuns
{
public:
    void initPrinter();
    double calcInterest();
    Date getDate();
};
```

220 Navesti osnovne karakteristike spregnutosti u kontekstu razvoja.

1. vrsta sprege
2. nivo sprege
3. širina sprege
4. način ostvarivanja sprege
5. intenzitet spregnutosti

221 Zašto je spregnutost komponenti softvera potencijalno problematična?

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

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

Želimo da umanjimo spregnutost da bismo mogli praviti izmene na jednom mestu bez toga da budemo time primorani da pravimo izmene i na drugim mestima.

222 Navesti i ukrakto objasniti vrste spregnutosti u kontekstu razvoja softvera.

- sprega logike
- sprega tipova
- sprega specifikacije

223 Objasniti spregu logike u kontekstu razvoja softvera.

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

- Komponente obavljaju različite delove istog posla.
- Komponenta A pretpostavlja da implentacija metoda B::M(...) počiva na nekom konkretnom algoritmu. Primer: A i B implementiraju komplementarne algoritme (kodiranje/dekodiranje, pisanje/čitanje, ...).
- Komponente dele pretpostavke o nekim konkretnim podacima. Primer: A zapisuje datoteku pod nekim imenom, a B je čita.

Sprega logike je visoko problematična i nepoželjna.

224 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 ili može biti **neodređena** ako komponenta A ne pravi instance tipa B (tj. može se u praksi dobijati na upotrebu instance podtipova). Primer: Komponenta A koristi klasu implementiranu u komponenti B.

Dobro implementirana sprega tipova ne predstavlja problem. Ovo posebno važi u slučaju kada je neodređena.

225 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 implicitnog polimorfizma;
- Upotreba referenci na metode objekata;
- Apstraktne bazne klase hijerarhija;
- Implementacija interfejsa.

Dobro implementirana sprega specifikacije ne predstavlja problem.

226 Navesti i ukratko objasniti nivoe spregnutosti u kontekstu razvoja softvera.

Spregnutosti se dele po nivou apstraktnosti delova koji se dele (od najjače ka najslabijoj): [20]

- po sadržaju (Content Coupling)
- preko zajednickih delova (Common Coupling)
- spoljašnja spregnutost (External Coupling)
- preko kontrole (Control Coupling)
- preko markera (Stamp Coupling)
- preko podataka (Data Coupling)
- preko poruka

227 Objasniti spregnutost po sadržaju u kontekstu razvoja softvera.

Spregnutost po sadržaju predstavlja otvoreno i neposredno korišćenje i/ili menjanje sadržaja jedne komponente od strane druge.
Opšti primer:

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

Konkretan primer:

```
class Pilot { ... };
class Plane
{
    double altitude, speed;
    friend class Pilot;
    // etc.
};
```

Problem:

- Narušena je enkapsulacija;
- Dovedene su u pitanje odgovornosti komponenti;
- Veoma je teško održavati komponentu od čijih internih aspekata implementacije zavisi druga komponenta.

Spregnutost po sadžaju je najjača i najnepoželjnija vrsta spregnutosti.

228 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.
Opšti primer:

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

Problem:

- 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.
- Ni odgovornosti nisu uspostavljene.
- Veoma je teško održavati ovako spregnute komponente.

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

229 Objasniti spoljašnju spregnutost u kontekstu razvoja softvera.

Spoljašnja spregnutost je na delu ako više komponenti upotrebljava isti od spolja nametnut koncept (interfejs, format podataka, komunikacioni kanal, ...).
Opšti primer:

```
... A::metod1 {
  ...external::oper1(...);
  ...external::oper2(...);
};
... B::metod2(...){
  ...external::oper3(...);
  ...external::oper4(...);
}
```

Problem:

- **Često ponavljanje koda:** Ako se koncept izmeni, moraju se menjati sve komponente koje ga upotrebljavaju.
- **Podeljene odgovornosti:** Poželjno je razmotriti učauravanje elemenata spoljašnjeg koncepta u novu komponentu (ili komponente), koju bi zatim koristile ovako spregnute komponente.

Spoljašnja spregnutost je sasvim prihvatljiva ako su spoljašnji koncepti relativno stabilni i pouzdani. Primer: slika implementira osnovne metode, a transformacije implementiraju složene operacije nad slikama.

230 Objasniti spregnutost preko kontrole u kontekstu razvoja softvera.

Spregnutost je **spregnutost 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 upustva za rad po kojima zatim samostalno postupa.

U odnosu na spolašnju spregnutost, kontroler je jedina ili primarna tačka kontrole.
Primer:

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

Primer: Kontrolisana klasa može da bude slika, a sve složene transformacije se implementiraju u drugoj klasi koja predstavlja kontroler.

Problem: Potencijalno nejasna podela odgovornosti. Ako je kontroler odgovoran za viši nivo operacija, onda to mora biti njegova jedina odgovornost. Često takvo rešenje ni po čemu nije bolje od integrisanih svih operacija u jedan

isti modul, tj. u oba slučaja potencijalno imamo module sa suviše odgovornosti.

Spregnutost preko kontrole je relativno jaka, ali može biti veoma korisna u nekim slučajevima.
Primer: Kontrolisana komponenta implementira elementarne postupke, a različite strategije obezbeđuju složene načine upravljanja radom komponente.

Primer sa slikom: Na nivou klasa može da bude dobro da se za svaku transformaciju pravi ova klasa. Na nivou komponenti, sve transformacije se obično okupljaju u jednu komponentu.

231 Objasniti spregnutost preko markera u kontekstu razvoja softvera.

Spregnutost je **spregnutost preko markera** ako više komponenti međusobno razmenjuje neku složenu strukturu podataka (marker) koju upotrebljavaju na različite načine. **Napomena:** Ako neposredno koriste iste podatke, onda je u pitanju spregnutost preko zajedničkih kanala.
Primer:

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

Problem:

- 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 spregnutosti 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.

232 Objasniti spregnutost preko podataka u kontekstu razvoja softvera.

Spregnutost je **spregnutost preko podataka** ako je jedna komponenta koristi interfejs druge komponente putem koga mu prenosi pojedinačne podatke.
Primer:

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

Problem: Zavisnost na nivou interfejsa.
Ovaj vid spregnutosti je među najnižim i sasvim prihvatlijv.

Spregnutost je **spregnutost preko poruka** ako jedna komponenta koristi interfejs druge komponente putem koga mu ne prenosi nikakve podatke. Primer:

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

Problem: Zavisnost na nivou interfejsa.
Ovaj vid spregnutosti je najniži i sasvim prihvatlijv.

233 Objasniti pojam širina sprege u kontekstu razvoja softvera.

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

- Uska sprega znači da su jasni i podela odgovornosti među komponentama i razlog njihove spregnutosti.
- Ako je sprega široka, to može da znači da:
  - Odgovornosti između ove dve komponente nisu dovoljno diferencirane. Možda je moguće suziti spregu implementiranjem nekih složenih celina koda u okviru komponente A.
  - Odgovornosti komponente A su preširoke. Možda je potrebno podeliti komponentu A na više delova.

234 Objasniti pojam smer sprege u kontekstu razvoja softvera

Sprega može biti:

- **Jednosmerna:** Komponenta A upotrebljava elemente komponente B, ali ne i obratno.
- **Dvosmerna:** Komponenta A upotrebljava elemente komponente B, ali i komponenta B upotrebljava elemente komponente A.

- **Cirkularna:** Ne postoji neposredno dvosmerna sprege dveju komponenti, ali postoji tranzitivna cirkularna spregnutost više komponenti.

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

- Usložnjava se podela odgovornosti među komponentama;
- Dovodi se u pitanje enkapsulacija.

Ipak, nekada dvosmera ili cirkularna spregnutost može da značajno pojednostavi dizajn. Primer: Inverzija odgovornosti kod obrasca „Posetilac“.

235 Objasniti pojam statička spregnutost u kontekstu razvoja softvera.

Statička spregnutost se ostvaruje pre izvršavanja programa i eksplicitno je iskazana u programskom kodu. Primeri:

- Nasleđivanje klase;
- Podatak klase A je objekat klase B;
- Argument metoda klase A je objekat klase B;
- U metodama klase A se pravi ili upotrebljava objekat klase B.

236 Objasniti pojam dinamička spregnutost u kontekstu razvoja softvera.

Dinamička spregnutost se ostvaraju u toku izvšavanja programa i nije eksplicitno iskazana kodom programa. Ovakva spregnutost može da zavisi od konfiguracije, ulaznih parametara i slično ...

Primer: Ako klasa A sadrži referencu na klasu B (statički), konkretan objekat (dinamički konfigurisan) može pripadati bilo kojoj klasi koja nasleđuje B.

237 Objasniti odnos statičke i dinamičke spregnutosti u kontekstu razvoja softvera.

Statička sprega je jača i nepoželjnija od dinamičke sprege.

- Deklaracija (interfejsa...) i implementacija neke statički spregnute komponente imaju neposrednog uticaja na implementaciju ostalih spregnutih komponenti.
- Promena dinamički spregnute komponente je lokalizovana tj. ako odgovara nasleđenom interfejsu, onda je sve u redu, a ako ne odgovara, onda je potrebno samo popraviti tu komponentu.

238 Objasniti pojam intenzitet spregnutosti u kontekstu razvoja softvera.

Intenzitet spregnutosti se može izraziti numerički na više načina. Primer:

- **Intenzitet spregnutosti dveju komponenti:**
  $coefSprege(A, B) = coefVrste(A, B) + coefNivoa(A, B) + ...$
- **Spregnutost jedne komponente sa sistemom:**
  $coefSprege(A) = sum_B(coefSprege(A, B))$
- **Ukupna spregnutost sistema:**
  $coefSpregeSistema = sum_A(coefSprege(A))$

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

Mera spregnutosti neke komponente sa sistemom se može izraziti na više načina:

- **Broj komponenti koje se referišu iz te komponente:** Broj elemenata drugih komponenti koje referišu iz te komponente.
- **Broj komponenti iz kojih se referiše ta komponente:** Broj elemenata te komponente koje se referišu iz drugih komponenti
- **Karakteristike posmatranih sprega.**
- ...

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

„Aksiome spregnutosti“:

- Sto je komponenta složenija, to je važnije da bude što manje spregnuta.
- Spregu je poželjno uvoditi na što jednostavnijim komponentama.

241 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

242 Objasniti karakteristike spregnutosti u slučaju arhitekture klijent-server.

Jedan od najčešćih i tipičnih oblika sprege:

- Komponenta server pruža usluge;
- Komponenta klijent koristi usluge.

Karakteristike sprege:

- 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.

243 Objasniti karakteristike spregnutosti u slučaju hijerarhije pripadnosti.

Relativno čest oblik sprege:

- Komponenta roditelj sadrži više dece koja obavljaju funkciju ili delove funkcije;
- Komponente deca obavljaju svaka svoj deo odgovornosti;
- Roditelj je odgovoran za koordinaciju.

Primer: prozor - dekorateri.
Karakteristike sprege:

- 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.

244 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 određenih poslova.

Primer: prozor-kontrole
Karakteristike sprege:

- 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.

245 Objasniti karakteristike spregnutosti u slučaju sprege putem interfejsa.

Tipičan oblik sprege u OO razvoju:

- Komponenta B deklarise (i implementira) interfejs;
- Komponenta A upotrebljava komponentu B posredstvom interfejsa.

Karakteristike sprege:

- Potencijalno jednosmerna;
- Širina zavisi od kvaliteta dizajna interfejsa;
- Nivo sprege je obično relativno nizak - preko poruka ili preko parametara;
- Sprega tipova ili specifikacije.

246 Objasniti karakteristike spregnutosti u slučaju sprege putem parametara metoda.

Čest oblik sprege u OO razvoju:

- Komponenta A deklarise metod koji kao parametar prima komponentu B;
- Komponenta BB, kao specijalizacija komponente B, se predaje na upotrebu komponenti A kao parametar deklarisanoj metodi.

Karakteristike sprege:

- Potencijalno jednosmerna;
- Širina zavisi od kvaliteta dizajna interfejsa;
- Nivo sprege je obično preko metoda;
- Obično sprega tipova ili specifikacija.

247 Objasniti odnos koncepta klase i pojma kohezije u kontekstu OO razvoja softvera.

U kontekstu OO programiranja, kohezija klase predstavlja stepen međupovezanosti elemenata jedne klase:

- Ako je nizak nivo kohezije u klasi, to znači da klasa nije dobro podeljena:
  - Ako postoji kohezija između nekih elemenata klase, ali ne svih, ili
  - ako se prepoznaje više različitih skupove elemenata klase unutar kojih postoji snažna kohezija.
- Moguće je da postoji problem prepoznavanja odgovornosti klase.
- Klasu je možda potrebno podeliti na više klasa.

248 Objasniti odnos koncepta klase i pojma spregnutosti u kontekstu 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.

249 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 se izdvajaju u potprograme,
- i dalje aspekti problema ostaju celina.

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

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

- Program obaveštava OS da je zainteresovan da reaguje na pritisak tastera od strane korisnika;
- OS prepoznaje dogadaj i salje programu poruku;
- Program (odgovarajući objekat) reaguje na poruku.

250 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 u fazi izvršavanja programa,
  - 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.

251 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 signale potrebno obraditi višestruko, dovoljno je dodati nove slotove koji će ih prihvatiti, a da se komponente koje emituju signale, kao i druge komponente koji već obrađuju te signale, ne moraju menjati.

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

- **Generator događaja:** objekat koji prepoznaje da je nastupio uslov koji predstavlja neki definisan dogadaj;
- **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.

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

Biblioteku QT koristimo za upravljanje događajima. Osnovni model objekta u QT biblioteci je **QObject**. Glavna karakteristika ovog modela je mogućnost komunikacije između objekata preko signala i slotova. Da bi se koristili signali i slotovi potrebno je uključiti makro **Q_OBJECT** u okviru QObject klase. Preporučuje se da se za sve klase koje nasleđuju QObject doda Q_OBJECT makro, nezavisno od toga da li zapravo koriste signala ili ne (neke funkcije mogu da imaju čudno ponašanje u suprotnom).

254 Objasniti koncept signala u kontekstu primene arhitekture zasnovane na dogadajima u okviru biblioteke QT.

Signali su metodi koji proizvode događaje. Deklarišu se u okviru sekcije signals kao metodi tipa *void* i 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.

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

Slotovi su metodi koji obrađuju događaje. Deklarišu se u okviru sekcije *private slots* ili *public slots*. Moraju biti istog tipa kao signali za čiju su obradu namenjeni.

256 Objasniti povezivanje signala i slotova u slučaju primene arhitekture zasnovane na događajima u okviru biblioteke QT.

Povezivanje se ostvaruje pomoću statičkog metoda QObject::connect.
Argumenti:

1. Pokazivač na objekat koji emituje signal;
2. Signal koji se emituje (navodi se pomocu makroa SIGNAL ili kao pokazivač na funkciju koja predstavlja signal);
3. Pokazivač na objekat koji hvata signal;
4. Slot kojim se signal hvata (navodi se pomoću makroa SLOT ili kao pokazivač na funkciju koja predstavlja slot).

Slot i odgovarajući signal moraju biti istog tipa (nije važno ime).
Zavisno od konfiguracije, nakon emitovanja signala se može sačekati i da se završi njegova obrada ili odmah nastaviti sa radom. Podrazumevano ponašanje je da se sačeka sa obradom.
Primer:

```
Counter a, b;
QObject::connect(&a, &Counter::valueChanged,
                 &b, &Counter::setValue);

a.setValue(12);     // a.value() == 12, b.value() == 12
b.setValue(48);     // a.value() == 12, b.value() == 48
```

Objašnjenje: Promena vrednosti objekta (brojača) *a* predstavlja događaj koji emituje signal *Counter::valueChanged*. Objekat *b* hvata signal u okviru slota *Counter::setValue* koji ažurira vrednost objekta *b*. Promena objekta *b* ne utiče na objekat *a*, jer nije tako definisano preko statičke *connect* metode.

Koja je razlika između korišćenja makroa SIGNAL i SLOT i korišćenja pokazivača na funkcije? Pre QT verzije 5 je postojao samo prvi način (makroa). Od QT verzije 5 moguće je koristiti i drugu varijantu.

257 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, uspostavljanje 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 programsko 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 okruženja za pravljenje ili menjanje korisničkih intefejsa i slično.

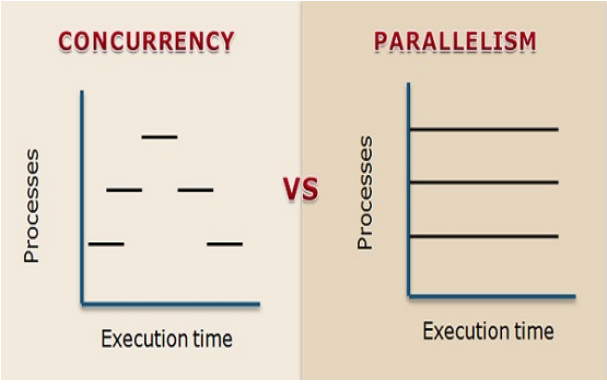
258 Šta je konkurento izvršavanje?

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

1. Posao T1 može da počne i završi pre početka T2.
2. Posao T1 može da počne i završi posle započinjanja posla T2.
3. Posao T1 može da počne pre započinjanja posla T2 i završi posle završavanja posla T1.
4. Posao T1 može da počne posle započinjanja posla T2 i završi pre završavanja posla T2.

259 Objasniti pojam paralelno izvršavanje.

Dva posla se izvršavaju paralelno ako postoji period vremena u kome su (doslovno) istovremeno aktivni. Paralelno izvršavanje je moguće samo ako imamo na raspolaganju više izvršnih jedinica (procesora ili jezgara).



Razlika i sličnosti između konkurentnog i paralelnog izvršavanja:

1. Konkurentnost podrazumeva neodređenost vremenskog odnosa, a paralelnost podrazumeva predodređenost vremenskog odnosa.
2. Konkurentni programi u krajnjem ishodu mogu da rade sekvencijalno ili deo po deo sekvencijalno, a paralelni programi moraju da rade istovremeno.
3. Konkurentni poslovi mogu, ali ne moraju da se izvršavaju paralelno. Paralelni poslovi mogu, ali ne moraju da budu konkurentni. Paralelni poslovi nisu konkurentni ako je unapred poznata njihova međusobna vremenska lociranost tokom čitavog perioda izvšavanja.
4. Programi su konkurentni ako su istovremeno u stanju izvršavanja. Programi su paralelni ako se istovremeno izvršavaju.
5. Konkurentno izvršavanje je moguće i na jednoj izvršnoj jedinici (procesoru). Paralelno programiranje zahteva više izvršnih jedinica (procesora ili jezgara).

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;
  - Primer: 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;
  - Primer: 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.

260 Objasniti pojam distribuirano izvršavanje.

Dva posla se izvršavaju distribuirano ako rade na različitim adresnim prostorima. Ovo je opštija definicija, koja teži da izjednači (apstrahuje) distribuiranost između više računara i distribuiranost između komponenti (procesora) istog računarskog sistema. Neke definicije su strože i podrazumevaju različite računarske sisteme, ali svi aspekti distribuiranja su uglavnom isti.

261 Objasniti pojam proces.

Proces je instanca programa koja se izvršava na računarskom sistemu. Obično se razmatra samo u kontekstu računarskih sistema koji imaju mogućnost izvršavanja više programa (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:

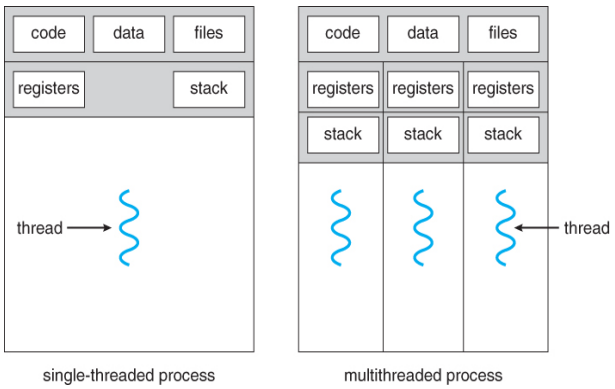
- Podatke o izvršavanju;
- Stanje izvršavanja (primer: spreman, radi, čeka, stoji, ...);
- Brojač instrukcija (adresa naredne instrukcije);
- Sačuvane vrednosti registara procesora;
- Informacije o upravljanju resursima;
- Informacije o memoriji (tablica stranica, podaci o alokaciji memorije, ...);
- Deskriptori datoteka;
- Ostali resursi, poput U/I zahteva i sličnog.

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 promašuju u kešu.

262 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. Nit je najmanja samostalna jedinica izvršavanja na procesoru.



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 izvršavanja procesa;

  - Ostali resursi, poput U/I zahteva i sl.

- Podaci o niti obuhvataju:
  - Stanje izvršavanja niti;
  - Brojac instrukcija;
  - Sačuvane vrednosti registara procesora.

263 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 jednog procesa, štedi se na resursima i dobija na performansama.

264 Objasniti sličnosti i razlike niti i procesa.

1. I procesi i niti se mogu izvršavati konkurentno i/ili paralelno;
2. Značajan broj problema pri konkurentnom izvršavanju se odnosi na isti način i na procese i na niti;
3. Samo procesi se mogu izračunavati distribuirano;
4. Sve niti jednog procesa rade istom adresnom prostoru;
5. Procesi ne dele među sobom resurse (bar ne neposredno);
6. Komunikacija između procesa se odvija isključivo kroz posebne mehanizme za međuprocesnu komunikaciju;
7. Nije potrebno eksplicitno staranje o eventualnom sukobljavanju oko resursa;
8. Niti jednog procesa dele resurse tog procesa;
9. Komunikacija između niti se najčešce odvija posredstvom deljenih resursa;
10. Neophodno je eksplicitno staranje o eventualnom sukobljavanjem oko resursa.

265 Objasniti kako se programiraju niti pomoću standardne biblioteke C++-a. Klase, metodi, ...

Klasa thread je implementirana u okviru C++ modula *thread*. Konstruktor za *thread* prima pokazivač na funkciju (ime funkcije) i redom njene parametre. Nit se automatski pravi i pokreće (ne postoji metod za započinjanje). [22]
Osnovni metodi:

- *get_id()* - Vraća identifikator niti;
- *joinable()* - Provera se da li je nit spojiva;
- *join()* - Spaja nit.

Primer:

```
#include <iostream>
#include <thread>

void some_function()
{
  // ...
}

void another_function(int x)
{
  // ...
}

int main()
{
  // kreira se objekat thread koji poziva some_function()
  std::thread first (some_function);
  // kreira se objekat thread koji poziva another_function(0)
  std::thread second (another_function, 0);

  std::cout << "main, both functions now execute concurrently...\n";

  // sinhronizacija niti:
  // ceka se da prva nit zavrsi
  first.join();
  // ceka se da druga nit zavrsi
  second.join();

  std::cout << "both functions completed.\n";

  return 0;
}
```

266 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 koristiti kao metodi):

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

Signali:

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

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

```
#include <QThread>

class MyThread : public QThread {

  Q_OBJECT

public:

  MyThread(...);

protected:

  void run();

};
```

267 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 kad se pozove metod start().

Dovršavanje:

- Kada nit završi 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 zaključanih 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 je veoma osetljiva operacija;
- Prekidanje niti preti da ugrozi 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;
- Primer: Nit koja završava može da postavlja neki deljeni signal, koji se zatim može lako proveravati od strane drugih niti.

POSIX (Portable Operating System Interface) je familija standarda definisana od strane IEEE ogranizacije sa namenom da se održi odgovarajuća kompatabilnost između operativnih sistema.

268 Objasniti detaljno operaciju pravljenja niti. Kako se implementira pomoću standardne biblioteke C++-a?

Pogledati prethodno pitanje sa primerom: „Objasniti kako se programiraju niti pomoću standardne biblioteke C++-a. Klase, metodi, ...“

269 Objasniti detaljno operaciju pravljenja niti. Kako se implementira pomocu 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 OS-a) kada se pozove metod void *start()* objekta klase QThread.

Napomena: Prilikom pokretanja niti se koristi metoda *start()*, a ne metoda *run()* koja se implementira u okviru klase koja nasleđuje QThread klasu. Prilikom pokretanja niti preko *start()* se kreira nit sa svojom memorijom i kodom, a onda se u okviru *start()* metode poziva metoda *run*.

270 Objasniti detaljno operaciju dovršavanja niti. Kako se implementira pomoću standardne biblioteke C++-a?

Nit se dovršava kada završi izvršavanje „glavne“ funkcije niti. Objekat niti se dovršava kad se izađe iz opsega u kojem je inicijalizovan.

Ako je nit spojiva (*joinable() == true*), poziva se *terminate()*. [23]

271 Objasniti detaljno operaciju dovršavanja niti. Kako se implementira pomocu biblioteke QT?

- Nit se dovršava kada se završi izvršavanje metoda void run() objekta klase QThread.
- Postoji metod *QThread::quit()* koji predstavlja opciju za dovršavanje niti. Ovaj metod ne radi ništa ako nit nema petlju za proveru događaja ili neki deo koda blokira petlju za proveru događaja.
- Postoji metod *QThread::terminate()*, ali korišćenje ove metode je loša praksa.

272 Objasniti detaljno operacije suspendovanja i nastavljanja niti. Kako se implementiraju standardne biblioteke C++-a?

Standardna biblioteka C++ nam nudi opciju da suspendujemo nit (u smislu da oslobodi CPU), kako bi se dao prioritet drugim nitima. Ovo se izvodi preko funkcije *void yield()* ili *sleep()*. [24]

273 Objasniti detaljno operaciju suspendovanja i nastavljanja niti. Kako se implementira pomocu biblioteke QT?

Suspendovanje niti nije moguće u QT.

274 Objasniti detaljno operaciju prekidanja niti. Kako se implementira pomoću standardne biblioteke C++-a?

- Nit se može prekinuti pozivanjem *terminate()* preko neke druge niti i nit na koja je referisana će biti prisilno prekinuta;
- Može da se pozove deskruktor niti **~thread()**.

275 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 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.

276 Objasniti detaljno operaciju čekanja niti. Kako se implementira pomoću standardne biblioteke C++-a?

Elementaran vid sinhronizovanja niti je kada jedna nit čeka da druga nit završi sa radom.

Ostvarivanje čekanja je relativno jednostavno, ali se preporučuje implementacija složenih mehanizama (primer: nit koja završava može da postavlja neki deljeni signal, koji je se zatim može lako proveriti od strane drugih niti).

Čekanje da se nit završi se izvodi pozivanjem metoda *void join()*. Metod *join()* završava rad tek kada odgavarajuća nit završi izvršavanje. Napomena: Na istoj niti se ne sme pozvati dva puta metod *join()*.

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

Biblioteka QT nema *join()* metodu za niti. Ali metoda *wait()* vrši sličnu funkciju. 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. Metod vraća *true* ako nit više nije aktivna (ili je završena ili nije ni započela rad) i *false* ako je nit jos aktivna.

278 Koji su najvažniji problemi pri pisanju konkurentnih programa?

- Deljenje resursa između jedinica izvršavanja;
- Komunikacija među jedinicama izvršavanja;
- Sinhronizacija jedinica izvršavanja.

279 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 se on mogao zaključati.

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

280 Objasniti podršku za mutekse u okviru standardne biblioteke C++-a.

Muteksi nude veoma jednostavan interfejs za signaliziranje ekskluzivnog pristupa kod kritičnih sekcija (onemugućava pristup ostalim nitima).
Osnovne operacije:

- **void lock():** Zaključava muteks ako je otklučan ili blokira izvršavanja.
- **bool try_lock():** Ako je muteks otključan, onda ga zaključava i vraća true, u suprotnom vraća false. Može da se koristi da nit radi jedan posao koji zahteva kritičnu sekciju, ako je muteks otključan ili da radi drugi posao koji ne zahteva kritičnu sekciju, ako je muteks zaključan.
- **void unlock():** Otključava muteks. Ako je muteks već otključan, onda nije definisano ponašanje.

[25]

281 Objasniti podrsku za mutekse u okviru biblioteke QT.

Muteksi su implementirani klasom **QMutex**.
Osnovni metodi:

- **Zaključavanje muteksa:** *void lock()*. Ako je već zaključan, čeka da se otključa pa ga zaključava. U svakom slučaju, vraća se tek kada je uspešno zaključan.
- **Otključava muteks:** *void unlock()*. Otključava muteks. Ako je zaključan od strane iste niti, otključa ga. Ako je zaključan od strane druge niti, proizvodi grešku. Ako nije zaključan ponašanje je nedefinisano.
- **Pokušaj zaključavanja:** *bool tryLock()* Ako je muteks otključan, zaključava ga i vraca *true*. Ako je već zaključan, ne radi nista i odmah vraca *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 je da se jednim 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.

282 Šta je, čemu služi i kako se koristi lock_guard iz standardne biblioteke C++-a? Objasniti detaljno.

Klasa lock_guard je okvir za muteks koji nudi pogodan RAII mehanizam za postavljanje vlasništva nad muteksom u okviru definisanog bloka. Kada se *lock_guard* kreira, on pokušava da zauzme vlasništvo nad tim muteksom. Kada se izađe iz odgovarajućeg bloka, *lock_guard* poziva svoj destruktor i muteks se oslobađa. [26]

```
#include <thread>
#include <mutex>
#include <iostream>

int g_i = 0;
std::mutex g_i_mutex;

void safe_increment()
{
    const std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;

    std::cout << std::this_thread::get_id() << ": " << g_i << '\n';

    // g_i_mutex se automatski oslobadja kada
    // izadje iz opsega
}

int main()
{
    std::cout << "main: " << g_i << '\n';

    std::thread t1(safe_increment);
    std::thread t2(safe_increment);

    t1.join();
    t2.join();

    std::cout << "main: " << g_i << '\n';
}
```

283 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 greške sa propuštanjem otključavanja.

Kada Koristimo QMutexLocker umesto QMutex? (ovo je analogno sa mutex i lock_guard u C++-u) Ako u funkciji, metodi ili bloku koda koji zahtevaju eskluzivan pristup postoji više mesta izlaska (primer: na više mesta se koristi *return*), potrebno je voditi računa o otključavanju muteks pre svakog izlaska. Alternativa je da se koristi *QMutexLocker* koji uzima vlasništvo nad muteksa i onemugućava pristup sve do izlaska iz opsega u kojem je *QMutexLocker* objekat definisan.

284 Šta su katanci? Kako se upotrebljavaju?

Muteksi se ponašaju kao specijalan slučaj katanaca. Oni 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.

285 Objasniti podršku za katance u okviru standardne biblioteke C++-a.

Katanci su definisani u okviru modula *mutex*. Muteksi se obično ne zaključavaju direktno, već koristeći *std::unique_lock* i *std::lock_guard* (bezbednije). [25]

286 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 milisekundama.
- *Kostruktor QReadWriteLock(RecursionMode mode = NonRecursive)* Opcioni argument može da bude QReadWriteLock::Recursive

287 Čemu služi klasa QReadLocker biblioteke QT? Objasniti detaljno.

Slično klasi QMutexLocker, služi da pojednostavi rad sa katancima. Dobar način da se preduprede greške sa propuštanjem otključavanja.

288 Čemu služi klasa QWriterLocker biblioteke QT? Objasniti detaljno.

Slično klasi QMutexLocker, služi da pojednostavi rad sa katancima. Dobar način da se preduprede greške sa propuštanjem otključavanja.

289 Šta je sinhornizacija? Šta se sve može sinhronizovati u konkurentnim programima?

Da bi se jedinice izvršavanja sinhrornizovale, 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.

290 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.

Binarni semafor je semafor koji ima samo jedan resurs tj. najviše jedna nit može a pristupa zaštićenim podacima u jednom trenutku.

Koja je razlika između binarnog semafora i muteksa? Muteks može

biti otključan samo od strane niti koja ga je zaključala, dok binarni semafor može biti otključan od strane bilo koje niti.

291 Objasniti podršku za semafore u okviru standardne biblioteke C++-a

Standardna biblioteka ima implementaciju za semafore od **C++20** standarda. Ukoliko se koristi ranija verzija C++ (što najverovatnije i jeste slučaj), onda se semafori mogu jednostavno implementirati preko C++ muteksa. [27]

292 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)*
  - Pokusava da smanji semafor za n.
  - Ako uspe vraća true, a inace vraća false.
- *bool tryAcquire(int n, int timeout)*
  - Pokusava da smanji semafor za n i čeka najduže *timeout* milisekundi.
  - Ako uspe vraća true, a inače false.
- *konstruktor QSemaphore(int n=0)*
  - pravi semafor i inicijalizuje brojač datom vrednošću.

293 Objasniti funkciju async i šablon future standardne biblioteke C++-a.

Šablonskoj funkciji *async()* prosleđujemo funkciju koju želimo da se izvrši zajedno sa parametrima te funkcije. Ova Funkcija vraća *future* objekat (klasa *future* je takođe šablonska). Osnovni metod koji koristimo kada radimo sa *future* objektima je metod *get()* koji blokira nit dok se ne postavi povratna vrednost ili dok se ne izbaci izuzetak. [28] [29]

```
#include <iostream>
#include <thread>
#include <future>

using namespace std;

int nestoBezveze( int n )
{
  this_thread::sleep_for( chrono::seconds(5));
  return n+100;
}


int main(int argc, char **argv)
{
  future<int> rezultat = async( nestoBezveze,5 );
  cout << "cekamo..." << endl;
  cout << rezultat.get() << endl;

  return 0;
}
```

294 Objasniti šablone future i promise standardne biblioteke C++-a

Šablonska klasa *promise* omogućava čuvanje vrednosti ili izuzetka koji će kasnije biti preuzeti asinhrono preko *future* objekta koji je kreiran preko *promise* objekta. Objekat *promise* je namenjen da se koristi samo jednom. [30]

```
#include <vector>
#include <thread>
#include <future>
#include <numeric>
#include <iostream>
#include <chrono>

// Racuna se suma brojeva vektora preko
// standardne accumulate funkcije
void accumulate(std::vector<int>::iterator first,
                std::vector<int>::iterator last,
                std::promise<int> accumulate_promise)
{
    int sum = std::accumulate(first, last, 0);
    accumulate_promise.set_value(sum);  // Notify future
}

void do_work(std::promise<void> barrier)
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    barrier.set_value();
}

int main()
{
    // Demonstracija koriscenja promise<int> za transmisiju rezultata
   izmedju niti.
    std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };
    std::promise<int> accumulate_promise;
    std::future<int> accumulate_future = accumulate_promise.get_future();
    std::thread work_thread(accumulate, numbers.begin(), numbers.end(),
                            std::move(accumulate_promise));

    // future::get() ceka da future dobije rezultat
    // Nepotrebno je pozivati wait() pre get()
    // accumulate_future.wait();
    std::cout << "result=" << accumulate_future.get() << '\n';
    work_thread.join();  // wait for thread completion

    // Demonstracija koriscenja promise<void> za signaliziranje stanje
   izmedju niti.
    std::promise<void> barrier;
    std::future<void> barrier_future = barrier.get_future();
    std::thread new_work_thread(do_work, std::move(barrier));
    barrier_future.wait();
    new_work_thread.join();
}
```

295 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)

296 Šta su potprogrami sa jedinstvenim povezivanjem?

Kažemo da je funkcija sa jedinstvenim povezivanjem ako ne sme biti istovremeno upotrebljivana u različitim nitima.
Mogući uzorci:

- U telu funkcije se upotrebljavaju globalni podaci ili drugi globalni resursi na način koji nije bezbedan;
- Ti glavni resursi se upotrebljavaju nezavisno od navedenih argumenata.

297 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 nad istim podacima.
Mogući uzorci:

- upotreba podataka ili resursa na način koji nije bezbedan.

298 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.

299 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 jedan metod je sa ponovljenim povezivanjem;
  - Nijednom podatku se ne može neposredno pristupati.
- **Bezbedna po niti:**

  - Ako su svi metodi bezbedni po niti;
  - Nijednom podatku se ne može neposredno pristupati.

300 Koje su najčesce 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.

301 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 sigurnim 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, ...

302 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, onda joj ne treba omogućavati 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.

303 Kako se bira gde se i kako postavljaju muteksi i katanci?

Ne stavljati katance i mutekse odokativno:

- Katanci, muteksi i slični mehanizmi za izolovanje će kod uciniti 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 verovatnocu čekanja niti;
- Povećanje broja katanaca stvara uslove za nastajanje mrtvih petlji.

Ne štititi samo kod ili samo podatke:

- Značajna unapređenja kvaliteta programa i nivoa performansi se mogu postići ako se zaključava prava vrsta resursa;
- Nekada je bolje štititi podatke, ali ne uvek.
  - Najčesce je dobro štititi podatke;
  - Ako je mnogo podataka koji se dele, dolazi do opasnosti od mrtvih petlji;

– Nekada je bolje štititi kod, ali ne uvek;
– Obično je bolje štititi 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.

304 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 (memory mapped files)
- datoteke

305 Objasniti razliku između komunikacije među procesima i komunikacije među nitima.

Komunikacija između procesa se može odvijati isključivo putem komunikacionih kanala obezbeđenih 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. atomičnosti operacija na deljenim resursima. Često je najbolje i za komunikaciju među nitima koristiti mehanizme namenjene za procese.

306 Šta je softverska metrika?

Metrika je nauka o merama i merenju.

Softverska metrika se bavi merenjem različitih karakteristika softverskih rešenja.

- Cilj je opisivanje stanja razvojnog projekta.
- Numeričkim opisivanjem različitih aspekata razvojnog projekta se stiču uporediva znanja o projektu.

307 Navesti najvažnije tipove softverskih metrika.

Neposredne (apsolutne) mere: Primer: broj jedinica koda ili broj linija koda.
Indirektne/izvedene mere: Primer: količnik drugih mera.
Intervalne mere: mera iskazana kao opseg vrednosti.
Normalne mere: iskazana rečima, npr. „jeste“, „nije“, „plavo“, ...
Uporedne mere: Primer: „veći od“, „manji od“, ...

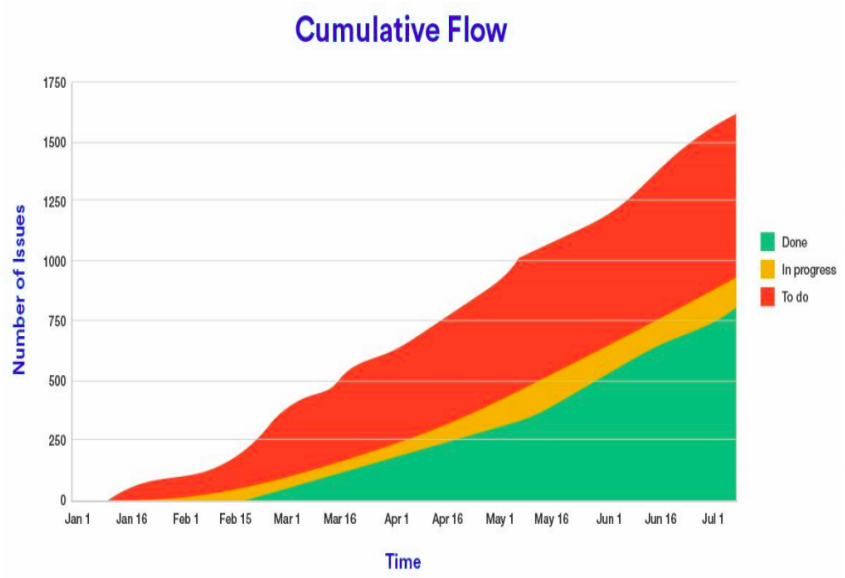
308 Objasniti vrste metrika u razvoju softvera.

Metrike praćenja razvoja softvera: Iskazuju u kojoj meri tok projekta prati planove. Odnose se na: proces razvoja i upotrebljene resurse (vreme, troškovi i sl.).
Metrike softvera: Iskazuju neke merljive odlike softvera. Odnose se na softver, kao proizvod razvojnog procesa.

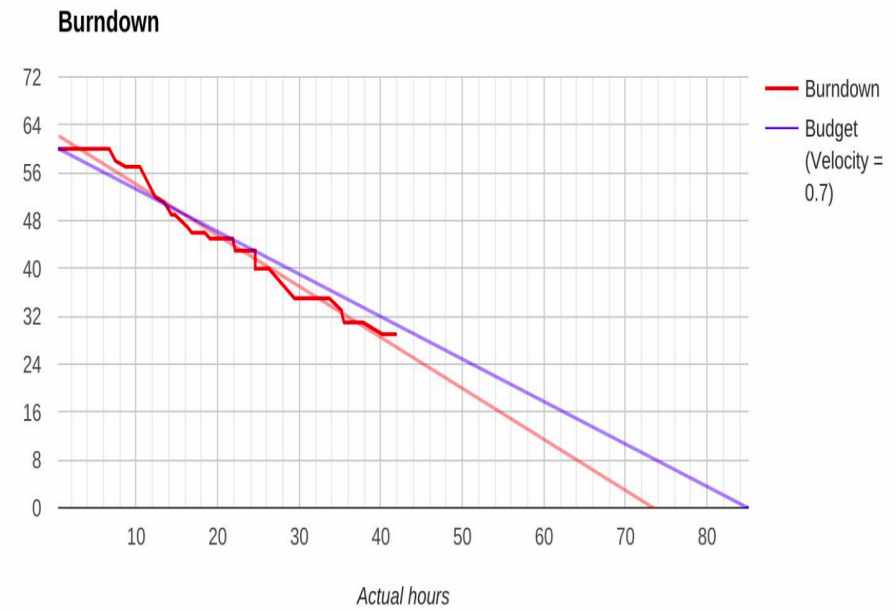
309 Navesti nekoliko metrika praćenja razvoja softvera. šta One opisuju?

Zovu se i **metrike upravljanja** zato što su važne pre svega upravljačkom timu:

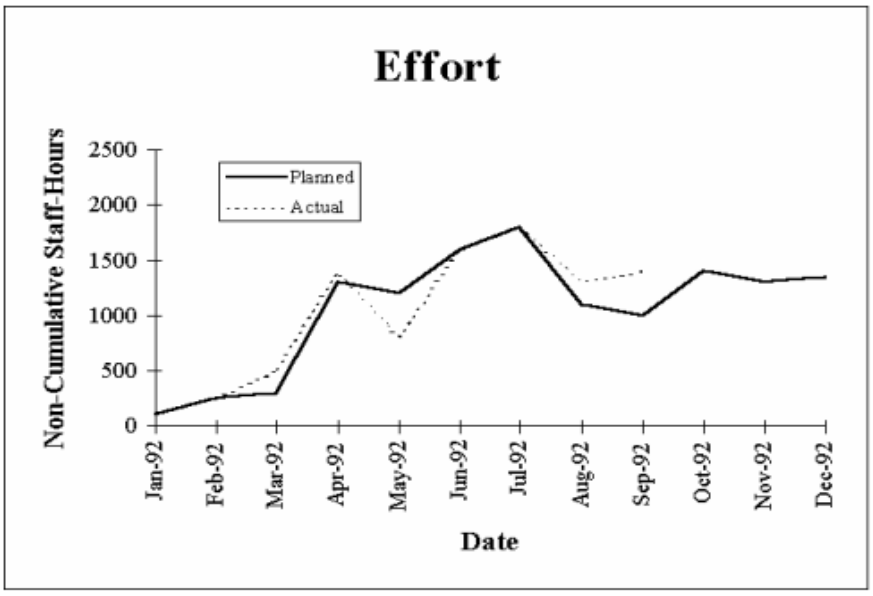
- **Mera napretka:**
  - Iskazuje uspešnost praćenja planova;
  - Predstavlja se broj planiranih i ostvarenih zadataka u odnosu na vreme;
  - Na kvalitet utiče ujednačenost celine zadataka.



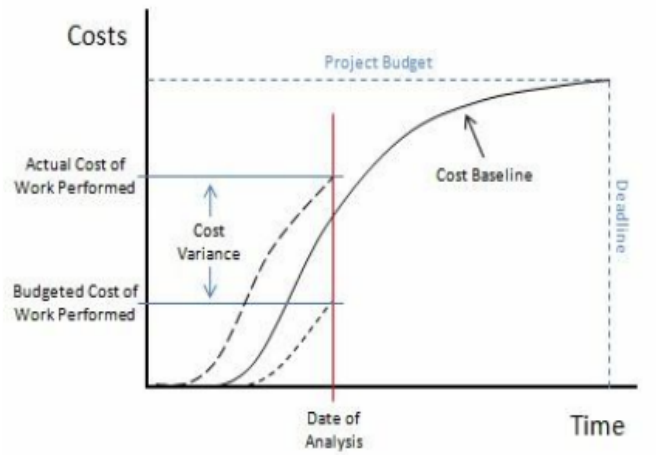
- **Mera završavanja:** Umesto ostvarenih poslova se predstavljaju preostali poslovi u odnosu na vreme. Ima bar dve linije tj. plan i ostvarenje.



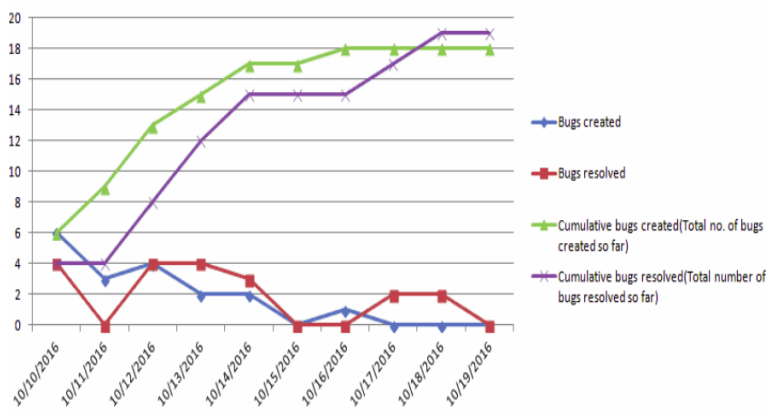
- **Mera napora:** Iskazuje planirani i ostvareni broj radnih sati u odnosu na vreme.



- **Troškovi:** Iskazuje planirani i ostvareni utrošak sredstava u odnosu na vreme. Dijagram sadrži:
  - projektovani ukupan budžet *(BAC)*;
  - planirana vrednost *(PV, planned value)* tj. planiran utrošak do tada
  - stvaran trošak *(AC, actual cost)* tj. stvaran utrošak do tada
  - dobijena vrednost *(EV, earned value)* tj. procenat obavljenog posla pomnožen sa projektovanim ukupnim budžetom



- **Problemi:** Ukazuje broj otvorenih i rešenih problema u odnosu na vreme. Može biti po danima ili kumulativno.



- **Stabilnost zahteva:** Iskazuje broj zahteva tokom vremena.
- **Stabilnost veličine:** Iskazuje veličinu softvera, obično iskazanu u broju jedinica koda, tokom vremena.

310 Navesti nekoliko metrika dizajna razvoja softvera. Šta one opisuju?

Metrike dizajna softvera 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.

311 Objansiti metriku stabilnost paketa.

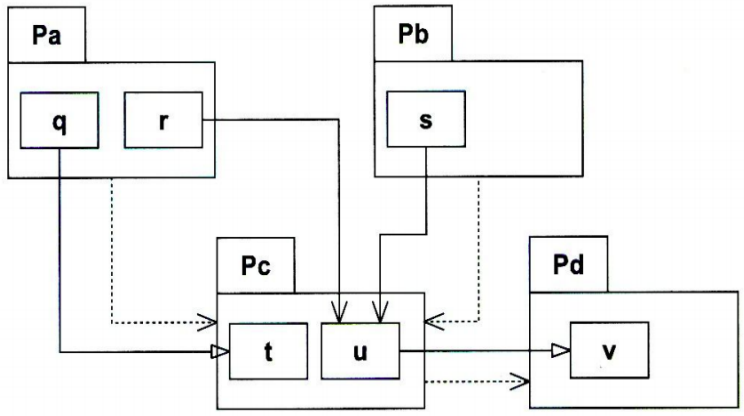
Stabilnost paketa predstavlja njegovu tendenciju da se ne menja tokom vremena.

- **Spregnutost prema paketu (eng. afferent coupling):** što ih je više, to će promene paketa biti teže izvodive, a time i ređe.
  C_a = broj klasa u drugim paketima koje zavise od klasa u posmatranom paketu
- **Spregnutost od paketa (eng. efferent coupling):** što ih je više, to će njegove promene česće biti potrebne.
  C_e = broj klasa u posmatranom paketu koje zavise od klasa u drugim paketima
- **Nestabilnost:** $I = \frac{C_e}{C_a + C_e}$. 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).

Primer: Određuje se nestabilnost paketa C tj. Pc.



Objašnjenje:

- $C_a = 3$, zavisnosti klasa: $q \to t, r \to u, s \to u$
- $C_e = 1$, zavisnosti klasa: $u \to v$
- $=> I = \frac{1}{4}$

312 Objansiti metriku apstraktnost paketa.

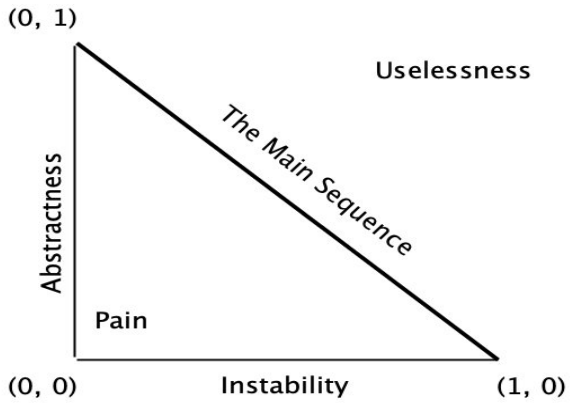
Apstraktnost paketa je relativna zastupljenost apstraktnih klasa u paketu:

- N_a :=**Broj apstraktnih klasa u paketu**
- N_c :=**broj klasa u paketu**
- **Apstraktnost** A**:** $A = \frac{N_a}{N_c}$ Ima opseg [0,1].

Princip odnosa stabilnosti i apstraktnosti: Paket treba da bude onoliko apstraktan koliko je stabilan. Drugim rečima, manje apstraktni paketi bi trebalo da zavise od apstraktnih, a ne obrnuto.

313 Objasniti odnos metrika stabilnosti i apstraktnosti paketa.

Poželjno je da odnos apstraktnosti i stabilnosti bude što bliže glavnoj sekvenci.
Udaljenost D se računa kao:
$D = \frac{|A+I-1|}{\sqrt{2}}$
Normalizovan oblik:
$D = |A + I - 1|$



314 Objasniti metriku funkcionalna kohezija paketa.

Jedan način izražavanja funkcionalne kohezije paketa je u obliku količnika broja zavisnosti među klasama paketa i broja paketa. Predstavlja intenzitet odnosa paketa sa sopstvenim 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.

315 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
- ...

316 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 je 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).

317 Šta je spremište? Šta sadrži? Kako je organizovano?

SKV (bilo da je centralizovan ili distribuiran) sadrži spremište. Spremište je mesto na kome se čuvaju različite verzije projekta. Obično se implementira pomoću odgovarajuće baze podataka.

318 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.

319 Objasniti pojam oznake u kontekstu upotrebe sistema za kontrolu verzija.

Često je pojedine verzije potrebno posebno označiti:

- Primer: verzija koja se javno publikuje;
- Ali i verzija koja predstavlja 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.

320 Objasniti grananje u kontekstu upotrebe sistema za kontrolu verzija.

Često je potrebno istovremeno razvijavati više različitih verzija koda:

- Primer: i nakon objavljivanja nove verzije potrebno je razvijati popravke za staru;
- Razvijanje novih funkcija za koje se ne zna kada će biti uvrstene 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 takode 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.

321 Šta su koflikti i kako se rešavaju u kontekstu upotrebe sistema za kontrolu verzija?

Često se dešava da se neki fajl menja od strane više 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 se ispravi neki problem koji je postojao i pre grananja, pa je potrbno ispravke preneti i u ostale grane.

322 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.

323 Objasniti primer strategije označavanja verzija.

Određivanje strategije označavanja verzija je sastavni deo upravljanja razvojnim projektom. Uobičajeno cetvorodelno označavanje verzije:
$< glavnaverzija > . < podverzije > [. < popravka > [. < revizija >]]$
Glavna verzija:

- Menja se kada su uvedene značajne izmene u odnosu na prethodnu verziju softvera;
- „Značajne 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
- xpack 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. Primer:

- 3.1a - alfa verzija, interna pregledna verzija;
- 3.1b - beta verzija, interna verzija koja može koristiti radi upoznavanja softvera ali nije jos dovršena;
- 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 uošenih nedostataka.

Postupak numerisanja:

- Obično se pre prvog publikovanja softver vodi sa glavnim brojem verzije „0“. Sve verzije koje prethodne verziji „1“ su „nezvanične“.

- Ne postoji koncenzus oko toga kako se interno numerisu 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 npr. 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 znacenja, 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 proglašava za javnu verziju 2.0.0

Numerisanje razvojnih verzija: Svaki od brojeva počinje od 0 i povećava se. Primer: verzija 3.2.0 je posle verzije 3.1.6. 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.

- Primer:
  - 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

324 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. Engleski termini:

- bug tracking system
- issue tracking system
- ticket system
- ...

325 Navesti i ukratko objasniti osnovne koncepte sistema Red-mine.

Savremeni sistemi za praćenje 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;
- ...

326 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.

327 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 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.

328 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 i kako se upotrebljava.

329 Navesti i ukratko objasniti osnovne opravdane i neopravdane motive za pravljenje dokumentacije.

Zato što to traži klijent:

- U pitanju je poslovna odluka (opravdano): Klijent obično ne ume da proceni potreban obim, kao ni pravi trenutak za pisanje dokumentacije (neopravdano).
- Iz ugla razvojnog tima je to legitiman razlog, ali je istovremeno i frustrirajuće (opravdano).
- Često proizvodi nepotrebne troškove, bez značajnih pozitivnih posledica ako nije praktično neophodna u trenutku kada se traži (neopravdano).

Da bi se formalizovale specifikacije (opravdano):

- Formalizovanje interfejsa za upotrebu komponenti: Važno za sve koji će razvijati i koristiti komponentu.
- Precizno objavljivanje ponašanja komponenti: Važno za sve koji će razvijati i koristiti komponentu.
- Konceptualan i implementacioni model komponente: Važno svim učesnicima u njenom razvoju.

Da bi se podržala komunikacija sa udaljenim timovima (opravdano):

- 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 slično;

Da bi se definisali ciljevi rada za neku drugu grupu:

- U nekim slučajevima može da bude dobro (opravdano):
  - 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 (neopravdano):
  - 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 (opravdano):

- 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) (opravdano);
- Klijenti često imaju stav da je dokumentacija pokazatelj uspešnosti razvoja, što je potpuno pogrešno. Dokumentacija se može napisati i o nečemu što nije čak ni dovoljno detaljno zamišljeno, a kamoli napravljeno (neopravdano).

Da bi se nešto temeljno razmotrilo (opravdano):

- 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 (neopravdano):

- 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 (neopravdano):

- Ako razvijalac nije dovoljno dobar, onda klijent može predati projekat i dokumentaciju drugom izvođacu, da bi nastavio posao (opravdano);
- 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 drugacije iskustvo, pristupe problemu i praksu.

330 Objasniti ulogu dokumentacije kao vida specifikacije zahteva projekta.

Pogledati prethodno pitanje.

331 Objasniti ulogu dokumentacije kao sredstva za komunikaciju.

Pogledati prethodno pitanje.

332 Objasniti ulogu dokumentacije u razmatranju nedoumica u projektu.

Pogledati prethodno pitanje.

333 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;
- Vec na sve segmente posla, od prve zamisli o softveru do gotovog softvera.

334 Šta obuhvata korisnička dokumentacija softvera?

1. **Opis čitavog sistema:** Uopšteni opis funkcija koje sistem pruža;

2. **Uputstvo za instalaciju:** Objašnjava kako se sistem priprema za rad, kako se prilagođava specifičnom okruženju, potrebama korisnika ili konkretnom računarskom sistemu;
3. **Vodič za početnike:** Pruža pojednostavljena objašnjenja kako započeti upotrebu sistema, obično u obliku tutorijala;
4. **Referentni priručnik:** Detaljan opis svih karakteristika i mogućnosti sistema i načina njihove upotrebe;
5. **Prilog o dopunama:** Opis izdanja pregledni izveštaj o svim izmenama i dopunama novog izdanja softvera;
6. **Skraćeni referentni pregled:** Pomoćni priručnik za brzo podsećanje;
7. **Upuststvo za administraciju:** Tehnički priručnik sa detaljnim objašnjenjima o mogućim naprednim prilagođavanjima specifičnim okolnostima i sa detaljnim opisom problema.

335 Šta obuhvata tehnička (sistemska) dokumentacija softvera?

1. **Vizija:** Objašnjava ciljeve sistema;
2. **Specifikacija i analiza zahteva:** Pruža detaljan opis zahteva ugovorenih između zainteresovanih strana (naručioci, klijenti, korisnici, projektanti, izvođači . . .)
3. **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.
4. **Opis implementacije - Detaljni opisi:**
   - Kako se pojedini elementi sistema modeliraju na konkretnim programskim jezicima;
   - Opisi značajnijih algoritama;
   - Specifikacije komunikacije.
5. **Plan testiranja softvera:** Opis protokola testiranja softvera (testovi jedinica koda i sve ostale vrste razvojnih testova);
6. **Rečnik podataka:** Sadrži rečnik termina i posebno opis osnovnih vrsta podataka koji se koriste u sistemu.
7. . . .

336 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 dokumentacija je 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 retko, 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).

337 Šta su alati za unutrašnje dokumentovanje programskog koda? Zašto su potrebni i po čemu se suštinski razlikuju od održavanja spoljašnje dokumentacije?

Unutrašnja dokumentacija objašnjava kako kod funkcioniše (primer: komentari u kodu), a spoljašnja dokumentacija objašnjava kako softver koristi.

Alati za unutrašnje dokumentovanje programskog koda služe za automatsko generisanje skeleta dokumentacije, što značajno olakšava posao.

338 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#, P HP , 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.

Primeri anotacija:[31]

- Javadoc stil:

```
/**
 * ... text ...
 */
```

- QT stil:

```
/*!
 * ... text ...
 */
```

- C++ stil (1):

```
///
/// ... text ...
///
```

- C++ stil (2):

```
//!
//! ... text ...
//!
```

- Uokvireni stil (1):

```
/*********************************************//**
*  ... text
**********************************************/
```

- Uokvireni stil (2):

```
//////////////////////////////////////////////////
/// ... text ...
//////////////////////////////////////////////////
```

- Širi opis (preko komande \brief)

```
/*! \brief Brief description.
 *         Brief description continued.
 *
 *  Detailed description starts here.
*/
```

- Dokumentacija članova(više opcija):

```
int var; /*!< Detailed description after the member */

int var; /**< Detailed description after the member */

int var; //!< Detailed description after the member
         //!<

int var; ///< Detailed description after the member
         ///<

int var; //!< Brief description after the member

int var; ///< Brief description after the member
```

- Dokumentacija parametara funkcije:
  - @param za informaciju o parametrima i [in], [out], [in,out] za smer

```
void foo(int v /**< [in] docs for input parameter v. */);
```

339 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.

340 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

341 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?

342 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?

343 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.

344 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 je 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 predvidas da će biti potrebno!"

345 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.

346 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.

347 Objasniti tehnike optimizacije „odbacivanje nepotrebne preciznosti“ i „tablice unapred izračunatih vrednosti“.

„Odbacivanje nepotrebne preciznosti“: Smanjivanje preciznosti u pokretnom zarezu i smanjivanje opsega celih brojeva može da smanji vreme izvršavanja i za više od 50%, posebno ako je dužina podatka inicijalno veća od dužine procesorske reči ili širine magistrale podataka.

„Tablice unapred izračunatih vrednosti“: 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.

348 Objasniti tehnike optimizacije „integracija petlji“, „izmeštanje inavarijanti izvan petlje“ i „razmotavanje petlji“.

„Integracija petlji“: Smanjivanje broja petlji (integracija), umesto dve uzastopne petlje pravljenje jedne sa složenijim korakom (u slučaju jednostavnih koraka) može da doprinese i do 35% .

„Izmeštanje inavarijanti izvan petlje“: Sve ono što može, izračunava se van petlje:

for(int i=0; i<limit(a,b,c); i++)... U tom slučaju koristiti pomoćne promenlive ili promeniti smer integracije.

„Razmotavanje petlji“: Smanjivanje broja ponavljanja uz povećanje složenosti koraka petlje može da doprinese i do 50%, ali može i da uspori izvršavanje u nekim slučajevima.

349 Objasniti tehnike optimizacije „smanjiti broj argumenata funkcije“ i „izbegavati globalne promenljive“.

„Smanjiti broj argumenata funkcije“: Ako funkcija ima manji broj argumenata, veća je verovatnoca 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 okviru strukture koja se prenosi po adresi.

„Izbegavati globalne promenljive“: Lokalne primenljive mogu da se optimizuju zapisivanjem u registrima, a globalne ne smeju, jer se nikada ne zna da li ih jos neko koristi (potprogram, druge niti i slično).

350 Objasniti tehnike optimizacije „upotreba umetnutih funkcija“ i „eliminacija grananja i petlji“.

„Upotreba umetnutih funkcija“: U slučaju jednostavnih funkcija (npr. swap) može da smanji vreme izvršavanja i do 50%.

„Eliminacija grananja i petlji“: U nekim slučajevima se grananja ili petlje mogu zameniti nešto složenijim izračunavanjem bez grananja i petlji. Može da se dobije i po nekoliko puta efikasniji 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 pojedinacno brojanje.

351 Objasniti tehnike optimizacije „zamenjivanje dinamickog uslova statickim“ 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.

„Snižavanje složenosti operacije“: Neke operacije mogu da se zamene efikasnijim operacijama Primer: umesto a*8 možemo da napišemo a « 3.

352 Objasniti tehnike optimizacije „redosled proveravanja uslova“ i „izbor rešenja prema najčešćem slučaju“.

„Redosled proveravanja uslova“: Ako se proverava veći broj uslova i postupa u skladu sa više invarijanti, redosled proveravanja uslova se mora odrediti tako da prosečan broj provera bude što manji.

„Izbor rešenja prema najčešćem slučaju“: 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 quicksort.

353 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 mnozenje. Medutim, većina prevodilaca to može da uradi umesto nas. Štaviše, to neće uraditi ako je procesor takav da to nije brze.

Smanjivanje koda nije uvek i njegovo optimizovanje: Dobar primer je upravo eliminacija petlji.

Optimizovanje tokom inicijalnog kodiranja: Predstavlja najćešći 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.

354 Navesti i ukratko objasniti tri tehnike optimizacije specifične za programski jezik C++.

1. 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.
2. Upotrebljavati reference umesto pokazivača (jednako efikasno, a mnogo čistije).
3. 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.
4. Iz delova koda koji se optimizuju izbaciti rukovanje izuzecima. Obrada izuzetaka je relativno neefikasna.

355 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 alokacije i dealokacije (new i delete);
- Upotreba šablona funkcija i metaprogramiranja.

356 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 vreme provedeno u samom kodu. Mogu da mere i zauzeće memorije, upotrebu keš memorije, razne druge stvari. Primeri:

- GNU gprof
- Valgrind

357 Prazno (radi usaglašavanja odgovora sa ispitnim pitanjima).

358 Šta je korisnički interfejs? Šta je grafički korisnički interfejs?

Korisnički interfejs (UI) predstavlja način na koji osoba interaguje sa softverom ili hardverom. Poželjno je da korisnički interfejs bude intuitivan za korisnika.

Grafički korisnički interfejs (GUI) predstavlja korisnički interfejs sa grafičkim kontrolama, gde korisnik može da interaguje sa softverom koristeći miš ili tastaturu. Primer: Način koji interagujemo sa operativnim sistemom na *Windows*-u.

359 Koje su osnovni kvaliteti i slabosti grafičkih korisničkih interfejsa?

Prednosti:

- Ne zahteva memorisanje imena komandi;
- Ne zahteva memorisanje parametara komandi;
- Uglavnom se lakše nauči korišćenje softvera na taj način;
- *Drag-and-drop* svojstvo;

Mane:

- Dodatna cena za razvoj;
- Dodatno vreme za razvoj;
- Uglavnom sporiji rad nego preko komandi;
- Zahteva više memorije;

360 Od čega zavisi upotrebljivost korisničkog interfejsa?

Korisnički interfejs treba da bude:

- **Čist:** Bitne informacije su očigledne. Olakšava učenje i smanjuje pravljenje grešaka. od strane klijenta.

- **Konzistentan:** Klijent bi trebalo da može da iskoristi prethodno naučeno za rešavanje novih zadataka.
- **Jednostavan:** Lako se uči.
- **Kontrolisan od strane klijenta:** Klijent kontroliše interfejs, ne računar.
- **Direktan:** Uslovno posledične veze su jasno vidljive.
- **Praštajuć:** Klijent sme da pravi greške.
- **Daje povratne informacije:** Informiše korisnika o stanju zadatka.
- **Estetičan:** Vizuelno privlačan i jasan način prezentovanih informacija.[32]

361 Koji su osnovni ciljevi pri oblikovanju korisničkog interfejsa?

Pogledati prethodno pitanje.

362 Kako se testira korisnički interfejs?

Korisnički interfejs se se testira preko testova prihvatljivosti, gde podrazumevamo da je pre toga odrađeno jedinično testiranje, integraciono testiranje i sistemsko testiranje.

Literatura

[1] geeksforgeeks: copy assignment. at: https://en.cppreference.com/w/cpp/language/copy_assignment.

[2] cppreference: copy assignment. at: https://en.cppreference.com/w/cpp/language/copy_assignment.

[3] stackoverflow: The rule of three. at: https://stackoverflow.com/questions/4172722/what-is-the-rule-of-three.

[4] cppreference: templates. at: http://www.cplusplus.com/doc/oldtutorial/templates.

[5] guru99: incremental model. at: https://www.guru99.com/what-is-incremental-model-in-sdlc-advantages-disadvantages.html.

[6] CatOnMat: polymorphism. at: https://catonmat.net/cpp-polymorphism.

[7] Visuel Paradigm: UML. at: https://www.visual-paradigm.com/guide/.

[8] Medium: Abstract Factory. at: https://medium.com/@hitherejoe/design-patterns-abstract-factory-39a22985bdbf#:~:text=In%20the%20case%20of%20the,be%20at%20the%20same%20time.

[9] Agile Manifesto: twelve principles. at: https://agilemanifesto.org/principles.html.

[10] Smart Sheed: four values and twelve principles. at: https://www.smartsheet.com/comprehensive-guide-values-principles-agile-manifesto.

[11] WikiC2: Why call it extreme? at: https://wiki.c2.com/?WhyCallItXp.

[12] Agile Alliance: Extreme Programming. at: https://www.agilealliance.org/glossary/xp.

[13] Airbrake: Extreme Programming. at: https://airbrake.io/blog/sdlc/extreme-programming.

[14] Extreme Programming: Acceptance Tests. at: http://www.extremeprogramming.org/rules/functionaltests.html.

[15] CppUnit: Assert macros. at: http://cppunit.sourceforge.net/doc/cvs/group___assertions.html.

[16] c2 wiki: AAA pattern. at: https://wiki.c2.com/?ArrangeActAssert.

[17] Catch2 Github: Assertions. at: https://github.com/catchorg/Catch2/blob/master/docs/assertions.md.

[18] Refactoring Guru: Refactoring. at: https://refactoring.guru/refactoring.

[19] GDB: Commands. at: https://sourceware.org/gdb/current/onlinedocs/gdb/TUI-Commands.html#TUI-Commands.

[20] geeksforgeeks: Coupling and Cohesion. at: https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/.

[21] sjsu: Design Goals and Principles. at: http://www.cs.sjsu.edu/faculty/pearce/oose/metrics/metrics.htm.

[22] cppreference: Thread. at: http://www.cplusplus.com/reference/thread/thread/.

[23] cppreference: Thread. at: https://en.cppreference.com/w/cpp/thread/thread/~thread.

[24] cppreference: Yield. at: https://en.cppreference.com/w/cpp/thread/yield.

[25] cppreference: Mutex. at: https://en.cppreference.com/w/cpp/thread/mutex.

[26] cppreference: Lock Guard. at: https://en.cppreference.com/w/cpp/thread/lock_guard.

[27] cppreference: Semaphore. at: https://en.cppreference.com/w/cpp/thread/counting_semaphore.

[28] cppreference: Async. at: http://www.cplusplus.com/reference/future/async/.

[29] cppreference: Future. at: http://www.cplusplus.com/reference/future/future/.

[30] cppreference: Promise. at: https://en.cppreference.com/w/cpp/thread/promise.

[31] Doxygen: Comment Blocks. at: https://www.doxygen.nl/manual/docblocks.html.

[32] UnixWore: UI. at: http://uw714doc.sco.com/en/SDK_vtcl/vtclgN.style_goodui.html#:~:text=A%20good%20interface%20makes%20it,premise%20of%20good%20UI%20design.