Razvoj softvera

1. Koji od pokazivača p1, p2 i p3 nije ispravno definisan u primeru:

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

p2 nije pokazivač - on je tipa int jer prilikom deklarisanja promenljivih * se vezuje za ime, a ne za tip promenljive.

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

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

Važi da je dužina s1 jednaka 4, a dužina s2 jednaka 3. Razlog za ovo je terminirajuća nula ‘\0’ koja se automatski dodaje u slučaju definisanja kao kod s1.

3. Šta će ispisati ovaj program?

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

Program ispisuje 3.

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

Primarno polimorfizam, kako hijerarhijski (nasleđivanje), tako i parametarski (šabloni).

5. Koji je redosled uništavanja objekata u primeru:

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

Prvo se uništava n (eksplicitno pozvan destruktor unutar main), zatim k (po izlasku iz opsega važenja promenljive, tj. funkcije main), i na kraju a (globalna promenljiva, ima statički vek trajanja, tj. živa je do završetka izvršavanja programa).

6. Da li je neka (koja?) od narednih linija koda neispravna?

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

Neispravna je 4. linija. Naime, tip promenljive p je u tom slučaju short**, a mi pokušavamo da mu dodelimo vrednost tipa short*, pa imamo neslaganje tipova.

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

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

Alokacija (na steku), a zatim inicijalizacija (eksplicitnim pozivom konstruktora bez argumenata klase Lista).

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;
}
```

Poziva se dva puta - za promenljive a i b. Promenljiva c je samo referenca na a, promenljiva p je dinamički alocirana i njen destruktor nikad nije pozvan (delete), a promenljiva q je pokazivač na b, pa je time pozvan destruktor pokazivača na A, a ne destruktor klase A.

9. Šta su šabloni funkcija?

Šablon funkcije je funkcija koja upotrebljava parametarske (simboličke) tipove u svojoj deklaraciji (tip povratne vrednosti i argumenata) i/ili implementaciji (moguća je upotreba parametarskih tipova i u samom telu funkcije). Moguća je i apstrakcija konstanti na sličan način kao i sa tipovima. Ovo je jedan od osnovnih konstrukata (pored šablona klasa) koji omogućava parametarski polimorfizam u jeziku C++.

10. Kako se pišu i koriste šabloni funkcija?

```
template<typename T>

T maksimum(T x, T y) {
    return x > y ? x : y;
}

int main() {
    // Kompajler sam zaključuje da je tip parametra T int
    cout << maksimum(2, 3) << endl;
    // Kompajler ne može da zaključi da li tip za T treba da bude int ili double
    // pa je potrebno eksplicitno navesti tip
    cout << maksimum(4, 9.8) << endl;
    return 0;
}
```

U slučaju da smo imali apstrakciju neke konstante u šablonu funkcije, to bi izgledalo ovako:

```
template<typename T, int K>

T f(T x) {
    return K * x;
}

cout << f<int, 10>(5) << endl;
```

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

```
template<typename T>

double sr_vrednost(T x, T y) {
    return (x + y) / 2.0;
}
```

12. Šta su šabloni klasa?

Šabloni klasa su definicije klasa koje koriste apstrakciju nekih tipova (u vidu simboličkih, tj. parametarskih tipova) ili konstanti. One su jedan od osnovnih konstrukata (pored šablona funkcija) koji omogućavaju parametarski polimorfizam u jeziku C++.

13. Kako se prevode šabloni klasa?

Prevođenje se radi iz više koraka. U prvom prolazu se standardno proverava sintaksna ispravnost definicije klase. Nakon toga, vrši se instanciranje šablona klase čime prevodilac prevodi strukturu klase i uočava probleme u njoj. Međutim, ovaj korak ne obuhvata i prevođenje instanci metoda klase - ono se odlaže sve do njihove konkretne upotrebe u kodu, kada se konačno izvršava i njihovo prevođenje (jedna od glavnih posledica ovoga je što greške prilikom definisanja šablona mogu ostati prikrivene).

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

Neophodno je da metod bude deklarisan sa ključnom rečju virtual, a dovoljno je da mu se pristupa preko reference ili preko pokazivača.

15. Šta je virtualna funkcija?

Virtualna funkcija je metod koji se može prevazići (override) u izvedenim klasama, što omogućava dinamičko vezivanje poziva tog metoda. Metod se deklariše kao virtualni uz pomoć ključne reči virtual.

16. Šta je čisto virtualna funkcija?

Čisto virtualna funkcija je apstraktni metod koji se mora implementirati u okviru izvedenih klasa čije objekte želimo da instanciramo. Klasa koja sadrži bar jedan čisto virtualni metod je apstraktna i ne može se instancirati.

17. Šta je apstraktna klasa?

Apstraktna klasa je klasa koja ima barem jedan čisto virtualni metod. Ona se ne može instancirati, ali se mogu deklarisati pokazivači i reference na objekte tog tipa. Konkretne, izvedene klase moraju implementirati sve njene čisto virtualne metode da bi se instancirale.

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

Umetnute funkcije su funkcije koje će MOŽDA biti razmotane tokom prevođenja da bi se zaobišlo kreiranje dodatnih stek okvira, što bi umanjilo performanse (slično pisanju makroa umesto funkcija u C-u). Umetnute

funkcije se deklarišu uz ključnu reč (specifikator) inline. Ovo predstavlja samo sugestiju kompilatoru da izvrši razmotavanje, ono nije zagarantovano.

```
inline int maksimum(int x, int y) {
    return x > y ? x : y;
}
```

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

Umetnuti metodi su funkcije-članice neke klase koje će MOŽDA biti razmotane tokom prevođenja da bi se zaobišlo kreiranje dodatnih stek okvira, što bi umanjilo performanse (slično pisanju makroa umesto funkcija u C-u). Umetnuti metodi se deklarišu uz ključnu reč (specifikator) inline. Ovo predstavlja samo sugestiju kompilatoru da izvrši razmotavanje, ono nije zagarantovano.

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

- Gubitak uloženih sredstava, odnosno plaćena cena neuspeha.
- Izgubljeno vreme koje opet povlači gubitak materijalnih sredstava.
- Posledice po čitav poslovni sistem.

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

- Prekoračenje troškova - možda ćemo na kraju ipak ostvariti profit, ali trenutno smo u gubitku jer trošimo više nego što smo planirali.
- Prekoračenje vremenskih rokova - ono može izazvati dodatne troškove usled produženog razvoja i/ili kašnjenja puštanja u rad (zamislimo kašnjenje pri implementaciji informacionog sistema Olimpijade).
- Neupotrebljiv rezultat - moguće je da neki zahtevi nisu ispunjeni ili čak da je sistem implementiran potpuno u skladu sa zahtevima, ali da opet ne odgovara stvarnim potrebama (primer je implementacija novog informacionog sistema carine u Australiji, ali nemogućnost migracije na njega pošto bi ona zahtevala prekid rada trenutnog sistema na duže vreme, što je nedopustivo). Moguće je i da neki detalji zahteva nisu dovoljno dobro precizirani, pa se tokom upotrebe ispostavi da softver nije dobar.
- Odustajanje od projekta usled bilo kog od ostalih razloga.
- Katastrofalne greške - katastrofalni bagovi mogu izazvati ne samo materijalnu štetu, već u nekim slučajevima izazvati opasnost i po ljudske živote (primer je pogrešna doza zračenja određena softverom uređaja za terapiju zračenjem). U pitanju su različite greške kao što su prekoračenje, neusklađene merne jedinice i slično.

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

Neupotrebljiv rezultat je jedna vrsta neuspeha pri razvoju softvera. Rezultat je neupotrebljiv ukoliko je implementiran u skladu sa zahtevima, ali opet ne odgovara stvarnim potrebama. Aspekti neupotrebljivosti su:

- neupotrebljiv korisnički interfejs (neintuitivni izgled korisničkog interfejsa, spor odziv, nepouzdanost, nepostojanje hardverskog odziva i slično).
- neostvariva ili loša (u realnim uslovima) procedura korišćenja - na primer, sistem u prodavnici ne dopušta da cena predmeta bude 0, pa "gratis" predmeti obično koštaju 1 dinar.

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

- Nerealni ili nejasni ciljevi projekta - napravi mi "nešto" što će mi utrostručiti godišnji profit.
- Neprecizna procena potrebnih resursa
- Loše definisani zahtevi - moguće je da je klijent želeo jedno, a razvijaoci su shvatili to potpuno drugačije. Ovo je često usko povezano sa slabom komunikacijom.
- Slaba komunikacija između klijenta, razvijaoca i korisnika - slaba komunikacija dovodi do nerazumevanja zahteva, kao i potrebe za čestim retroaktivnim izmenama sistema koji se razvija.

- Softver je dugo razvijan (npr. 5 godina) i sada više nije upotrebljiv iako je razvijen tačno prema zahtevima.

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

Na čijoj god strani se nalazi problem, razvijaoc je dužan da proba to da reši. Problem može nastati na strani:

- klijenta (ulagača) - nerealni ili nejasni ciljevi projekta (klijent ni sam ne zna šta hoće, primer sa "napravi mi nešto i utrostruči mi godišnji profit"), neusklađenost ciljeva i strategije (razvijamo softver za sektor koji će biti rasformiran, u suštini uzaludan rad), neusklađenost u politici ulagača (informacioni sistem fakulteta - studentska služba hoće nešto, profesori hoće nešto drugo, studenti nešto potpuno treće), komercijalni pritisak (razvojni tim se požuruje da bi se što pre ostvario profit), otpor korisnika prema primeni novog softvera i slično.
- razvijaoca - svi problemi na strani razvijaoca potiču od nedovoljne stručnosti tima. U ovo spadaju slabo vođenje projekta, neprecizna procena potrebnih resursa, nesposobnost da se iznese složenost projekta, upotreba nezrelih tehnologija, slabo izveštavanje o stanju projekta, neupravljani rizici i slično.
- obe strane - slaba komunikacija između klijenta, razvijaoca i korisnika (dovodi do retroaktivnih izmena, gubitka vremena, loše definisanih ciljeva), nepoverenje između klijenta i razvijaoca (novije metodologije razvoja se trude da reše ovaj problem), loše definisani zahtevi i slično.

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

Nerealni ili nejasni ciljevi projekta (klijent ni sam ne zna šta hoće, primer sa "napravi mi nešto i utrostruči mi godišnji profit"), neusklađenost ciljeva i strategije (razvijamo softver za sektor koji će biti rasformiran, u suštini uzaludan rad), neusklađenost u politici ulagača (informacioni sistem fakulteta - studentska služba hoće nešto, profesori hoće nešto drugo, studenti nešto potpuno treće), komercijalni pritisak (razvojni tim se požuruje da bi se što pre ostvario profit), otpor korisnika prema primeni novog softvera i slično.

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

Svi problemi na strani razvijaoca potiču od nedovoljne stručnosti tima. U ovo spadaju slabo vođenje projekta, neprecizna procena potrebnih resursa, nesposobnost da se iznese složenost projekta, upotreba nezrelih tehnologija, slabo izveštavanje o stanju projekta, neupravljani rizici i slično.

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

Slaba komunikacija između klijenta, razvijaoca i korisnika (dovodi do retroaktivnih izmena, gubitka vremena, loše definisanih ciljeva), nepoverenje između klijenta i razvijaoca (novije metodologije razvoja se trude da reše ovaj problem), loše definisani zahtevi i slično.

28. Objasniti kako pristupi planiranju mogu dovesti do problema.

Ekstremi pri planiranju najčešće dovode do problema. Sa jedne strane imamo nedovoljno planiranje koje se ogleda kroz nedovoljnu analizu problema i loše i neprecizno definisanje zahteva koje za posledicu najčešće ima veliki broj retroaktivnih korekcija, slabu upotrebljivost rešenja i probijanje rokova. Sa druge strane imamo preterano planiranje koje se ogleda kroz previše duboku i obimnu analizu, preširoko ulaženje u implementaciju i preambicioznost. Posledica ovakvog pristupa su uglavnom kasno uočavanje propusta (nekada tek nakon puštanja softvera u rad), otežana ili nemoguća tranzicija sa starog softvera na novi, probijanje rokova i smanjena fleksibilnost tokom razvijanja.

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 to postane pretnja uspešnom dovršivanju projekta.

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

Manjak osoblja, nerealni rokovi i budžet, razvoj pogrešnih funkcija i interfejsa, preterivanje, neprekidni niz izmena u zahtevima, slabosti eksterno realizovanih poslova, slabosti u eksterno nabavljenim komponentama, slabe performanse u realnom radu, rad na granicama računarskih nauka, tj. sa novim netestiranim tehnologijama.

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
- Davanje prednosti objektima u odnosu na procese ("u žiži su objekti, a ne procesi")
- Pravljenje prototipova

32. Objasniti koncept inkrementalnog razvoja.

Ideja je da više ne posmatramo projekat kao jedan veliki monolit (što je dovodilo do toga da se do završetka projekta promene okolnosti i zahtevi), već da razvijanje projekta ide deo po deo. Ovo se može raditi na dva načina. Prvi način je da se projekat inkrementalno nadograđuje, tj. da se postepeno dodaju funkcionalnosti, a drugi način je "evolutivni" razvoj projekta gde se odmah razvija ceo projekat, ali ne do krajnjih detalja, već se vremenom dalje oblikuje i popravlja. Dobit ovakvog koncepta razvoja su povećano poverenje klijenta (jer sada možemo da pokažemo rezultate pre završetka celog projekta) i lakše planiranje i implementacija usled manjih i jednostavnijih poslova. Međutim, potrebno je biti oprezan u planiranju koraka za razvoj. Ukoliko se u prvim koracima potpuno zanemare kasniji, postoji rizik da će biti potrebne veće izmene kasnije ili potpuno zanemarivanje nekih funkcionalnosti, ali ako se uzme previše koraka u obzir, onda će razvoj izgledati kao prvobitni, monolitni razvoj.

33. Objasniti koncept određivanja koraka prema rokovima.

Ovaj koncept je usko vezan za koncept inkrementalnog razvoja. Naime, ideja je da se za svaki inkrementalni korak prvo odrede budžet i rokovi, a zatim da se odrede poslovi koji će biti obavljeni u svakom koraku tako da odgovaraju zadatim ograničenjima. Ovim pristupom se smanjuje rizik od probijanja budžeta i rokova i uspostavlja se redovan ritam isporučivanja novih verzija (što povećava poverenje klijenta). Glavni rizik je isti kao i kod inkrementalnog razvoja - mogućnost zanemarivanja nekih funkcionalnosti pri kraju razvoja projekta.

34. Objasniti koncept pojačane komunikacije među subjektima.

Ideja je da se u planiranje uključe sve vrste subjekata (razvijaoci, klijent, korisnici) u što većem broju. Ovaj koncept se često kombinuje sa određivanjem koraka prema rokovima (klijent odlučuje šta je najbitnije za njega u toj iteraciji i poslovi se zatim određuju u skladu sa tim). Jedan od glavnih rizika ovakvog pristupa je taj što klijent može da počne sa stalnim iznošenjem novih zahteva, bez obzira na to što nekim treba da da prioritet, a neke treba da odlaže. Prednost je bolje razumevanje zahteva i bolje upoznavanje sa upotrebom softvera od strane svih subjekata.

35. Objasniti koncept davanja prednosti objektima u odnosu na procese.

Umesto procesa, projekat posmatramo kroz objekte, tj. pokušavamo da prepoznamo zajedno i strukturu i ponašanje nekih elemenata softvera. Prednost ovog koncepta je veća stabilnost (objekti su stabilniji od procesa, tj. izmene u načinu poslovanja manje utiču na objekte nego na procese). Na primer, ako pravimo softver za polaganje prijemnog ispita, a on bude otkazan, uklonićemo samo objekat koji opisuje prijemni ispit i njemu prateće procese, ali i dalje imamo objekte koji predstavljaju studente. Ovakav pristup je prilagođeniji inkrementalnom razvoju. Međutim, ako posmatramo samo objekte, tj. ako potpuno zanemarimo procese, možemo napraviti velike propuste u arhitekturi sistema koji se kasnije veoma teško ispravljaju.

36. Objasniti koncept pravljenja prototipova.

Ideja je da se u komunikaciji sa klijentom njemu isporučuje nešto što liči na finalni softver u tom koraku (prototip). Ovo pojačava komunikaciju sa klijentom i smanjuje se mogućnost greške usled pogrešne komunikacije (klijent, čak iako je netehnička osoba, može uočiti potencijalne nedostatke ukoliko mu se prikaže prototip). Treba voditi računa o tome da se prototipima ne posvećuje previše vremena u razvoju.

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

Nastale su početkom 80-ih kada više nije bilo dovoljno strukturno projektovanje i programiranje. U tom periodu projekti su postajali sve veći, a time i potreba da posmatramo objekte (tj. entitete) u sistemu koji imaju svoja stanja i ponašanja, da povećamo nivo apstrakcije (što dovodi do višestruke upotrebljivosti softvera) i modularnosti, da uvedemo u upotrebu vizualne korisničke interfejse, komunikaciju sa drugim računarima, da skratimo razvojne cikluse, itd.

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

Davanje prednosti objektima u odnosu na procese - posmatramo zajedno strukturu i ponašanje nekih elemenata softvera, što dovodi do veće stabilnosti. Sve objektno orijentisane metodologije se odlikuju skraćivanjem trajanja razvojnih ciklusa - uvodimo inkrementalni razvoj. Pojačana komunikacija među subjektima u svim fazama razvoja - uključujemo sve vrste subjekata u proces planiranja (razvijaoce, klijenta, korisnike).

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

Objekat je entitet koji ima definisanu strukturu i ponašanje u okviru sistema. Coad, Yourdon (1990): objekat je apstrakcija nečega u domenu problema, koja opisuje sposobnost sistema da o tome čuva informaciju, interaguje sa time ili oboje.

40. Šta je klasa? Atribut? Metod?

Klasa je opis skupa objekata koji dele istu strukturu i ponašanje, tj. dele iste atribute, operacije, metode, odnose i semantiku. Atribut je jedno imenovano polje klase. Skup atributa opisuje strukturu klase (i objekata koji joj pripadaju). Metod je funkcija definisana u okviru klase. Skup metoda opisuje ponašanje klase (i objekata koji joj pripadaju).

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

- Enkapsulacija
- Interfejs
- Polimorfizam
- Nasleđivanje

42. Objasniti koncept enkapsulacije.

Glavna ideja u konceptu enkapsulacije je da je struktura objekta njihova interna stvar i da ona treba biti sakrivena od spoljašnjeg sveta (atributima se može pristupati samo posredno, kroz interfejs). Glavne prednosti enkapsulacije su razdvajanje interfejsa od implementacije, veća apstrakcija strukture objekta, kao i teže (ili nemoguće) dovođenje objekta u nedozvoljeno stanje.

43. Objasniti koncept interfejsa.

Ideja interfejsa je da objekat treba da pruža spoljašnjim korisnicima samo skup metoda putem kojih se sa njim može komunicirati. Takav skup metoda naziva se javni interfejs objekta (klase) i on se najčešće oblikuje tako da omogućava obavljanje jednog celovitog posla. Glavna prednost je sakrivanje potencijalno komplikovane implementacije od korisnika.

44. Objasniti koncept polimorfizma.

Koncept polimorfizma podrazumeva da se jednom napisan kod može upotrebljavati za različite vrste objekata. Glavna prednost koju koncept polimorfizma uvodi je veća apstraktnost i veća (ponovna) upotrebljivost koda. Osnovne vrste polimorfizma su:

- hijerarhijski - implementiramo metod za bazu hijerarhije, a sve ostale klase će naslediti te funkcionalnosti. Naziva se nasleđivanje.
- parametarski - metode ne rade sa konkretnim tipovima podataka, već koriste "šablone" umesto tipova.
- implicitni - nigde ne preciziramo tip objekata već pišemo funkciju koja potencijalno radi sa bilo kojim objektom ako je to moguće.

45. Objasniti koncept nasleđivanja i odgovarajuće odnose.

Nasleđivanje je koncept koji omogućava hijerarhijski polimorfizam. Ono je zapravo ekvivalentno uvođenju refleksivne i tranzitivne relacije (parcijalne relacije poretka) "jeste" između klasa. Za klasu A se kaže da "jeste" klasa B akko svaki objekat klase A ima sve osobine koje imaju i objekti klase B. U ovom slučaju se takođe kaže i da je klasa A "izvedena" iz klase B, a klasa B je "osnovna" klasa za klasu A. Nasleđivanjem se eksplicitno označava sličnost među klasama.

46. Kroz koje faze je prošao razvoj OO metodologija?

Početni koraci do 1997. godine, oblikovanje UML-a od 1995. do 2005. godine i post-UML koraci od 2000. godine.

47. Koje su karakteristike prve faze razvoja OO metodologija?

Period početnih koraka razvoja OO metodologija je karakterizovan velikim brojem različitih notacija i metodologija, ali ni jedna od njih dovoljno široka i kompletna.

48. Koje su karakteristike druge faze razvoja OO metodologija?

Period oblikovanja UML-a je karakterizovan sa nekoliko dubljih metodologija koncentrisanih na različite faze razvoja kao i ujednačavanjem notacije (na kojoj je bio akcenat). Kao produkt ove faze dobijen je UML (Unified Modeling Language).

49. Koje su karakteristike treće faze razvoja OO metodologija?

Post-UML period je karakterisan ujednačenom notacijom za projektovanje (u vidu UML-a), širokim shvatanjem procesa razvoja, potpunom posvećenosti metodologijama i razvojem i upotrebom agilnih metodologija.

50. Šta je UML?

UML, tj. objedinjeni jezik za modeliranje, je prvenstveno grafički jezik koji se koristi za modeliranje i vizuelizaciju softverskih sistema. UML dijagrami su često sastavni deo projektne dokumentacije.

51. Koje (tri) vrste dijagrama postoje u UML-u? Objasniti.

- Strukturni dijagrami - opisuju statičku (nezavisnu od vremena) strukturu softverskog sistema i njegovih delova.
- Dijagrami ponašanja - bave se dinamičkom prirodom softverskog sistema, tj. opisuju ponašanje objekata ili podsistema u kontekstu toka vremena.
- Dijagrami interakcije - oni su podgrupa dijagrama ponašanja gde je vremenski tok opisan eksplicitnije nego na ostalim dijagramima i uključuju interakciju dva ili više objekata.

52. Navesti strukturne dijagrame UML-a.

- Dijagram klasa
- Dijagram objekata
- Dijagram paketa

- Dijagram složene strukture
- Dijagram komponenti
- Dijagram isporučivanja
- Dijagram profila

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

Dijagram klasa je strukturni UML dijagram čija je osnovna namena opisivanje statičke strukture modela softvera, tj. strukturu klasa i njihove međusobne odnose. Osnovni elementi dijagrama klasa su klase koje su predstavljene pravougaonicima.

54. Kako se predstavljaju odnosi između klasa na dijagramu klasa? Koji odnosi postoje?

Odnosi između klasa se predstavljaju linijama koje, u zavisnosti od vrste odnosa, imaju različite simbole na krajevima. Odnosi koji se predstavljaju su:

- Nasleđivanje - puna linija, strelica u obliku praznog trougla na strani bazne klase.
- Implementacija interfejsa - isprekidana linija, strelica u obliku praznog trougla na strani interfejsa.
- Agregacija - puna linija, prazan romb na strani klase koja sadrži delove.
- Kompozicija - puna linija, pun romb na strani klase koja sadrži delove.
- Asocijacija - puna linija, bez dodatnih simbola.
- Navigacija - jednosmerna asocijacija, puna linija, strelica na jednom kraju.
- Zavisnost - isprekidana linija, strelica na jednom kraju.

55. Objasniti kardinalnosti odnosa na dijagramu klasa - šta predstavljaju i kako se označavaju?

Kardinalnost odnosa predstavlja broj objekata koji mogu učestvovati u tom odnosu. Kardinalnost se navodi suprotno od klase za koju naznačavamo koliko njenih objekata može učestvovati u odnosu, u formatu od..do (npr. 10..20 - 10 do 20).

56. Koje su vrste dijagrama klasa i po čemu se razlikuju?

Različite vrste dijagrama klasa prikazuju drugačije elemente u zavisnosti od značajnosti u nekom konkretnom kontekstu. Razlikujemo sledeće osnovne vrste:

- Kompletan dijagram klasa - sadrži sve elemente (ime klase, atribute, metode, ograničenja, odnose), relativno retko se koristi zbog potencijalne pretrpanosti informacijama.
- Dijagram modela domena - prvenstveno se bavi strukturom objekata (prikazuje atribute), a zapostavlja njihovo ponašanje (ne prikazuje metode).
- Dijagram interfejsa klasa - prvenstveno se bavi ponašanjem (prikazuje javne metode, atributi se zanemaruju).
- Dijagram implementacije - slično kao dijagram interfejsa, samo što prikazuje sve metode (i javne, i privatne).

57. Koja je uloga i šta su osnovni elementi dijagrama komponenti?

Dijagram komponenti predstavlja vezu između funkcionalnih ili fizičkih komponenti softvera. Slično kao i kod klasa, komponente treba da imaju jednu odgovornost i da nju obavljaju samostalno. Osnovni elementi dijagrama su komponente i interfejsi kojima se one povezuju. Komponente se označavaju pravougaonikom sa stereotipom slagalice u gornjem desnom uglu, a interfejsi malim krugom sa polukružnom linijom prema interfejsu. U slučaju da postoji zavisnost između komponenti, a ne želimo da istaknemo interfejse, onda se one povezuju isprekidanom strelicom.

58. Koja je uloga i šta su osnovni elementi dijagrama objekata?

Dijagram objekata služi da bolje predstavi dinamičku prirodu odnosa između objekata i kao takav se često koristi kao dopuna dijagramu klasa. On sadrži nazive klasa, objekata, imena i vrednosti atributa i odnose među objektima u jednom trenutku vremena. Objekti su predstavljeni pravougaonicima koji u prvom delu sadrže naziv objekta i njegovu klasu, a u drugom spisak atributa (ne nužno svih) i njihovih vrednosti u tom trenutku.

59. Koja je uloga i šta su osnovni elementi dijagrama isporučivanja?

Dijagram isporučivanja predstavlja elemente fizičke arhitekture softverskog sistema i način raspoređivanja softverskih komponenti na te fizičke elemente. On sadrži čvorove (serveri, klijenti, uređaji), softverske i hardverske podsisteme ili komponente, kao i linije komunikacije između uređaja i međusobne veze podsistema.

60. Koja je uloga i šta su osnovni elementi dijagrama paketa?

Dijagram paketa opisuje kako su elementi logičkog modela (klase, komponente, slučajevi upotrebe, ...) organizovani u pakete, kao i međuzavisnost tih paketa. Elementi unutar jednog paketa bi trebalo da su semantički povezani i da se zajedno menjaju. Dijagram sadrži nazive i granice paketa i njihove međusobne zavisnosti (može sadržati i klase u paketima).

61. Navesti dijagrame ponašanja UML-a.

- Dijagram aktivnosti
- Dijagram stanja
- Dijagram slučajeva upotrebe
- Dijagrami interakcije (kao podvrsta, obuhvataju dijagram sekvence, dijagram komunikacije, dijagram vremena i pregledni dijagram interakcije)

62. Koja je uloga i šta su osnovni elementi dijagrama aktivnosti?

Dijagram aktivnosti opisuje poslovne procese višeg i nižeg nivoa i tokove podataka između njih. On sadrži procese, tokove podataka između procesa, grananja i čvorišta, uslovne tačke i početne i završne tačke. Može se grupisati i po trakama aktera, tako da se vidi koji od subjekata je zadužen za koju aktivnost. Sličan je dijagramskim tehnikama opisivanja algoritama.

63. Koja je uloga i šta su osnovni elementi dijagrama stanja?

Dijagram stanja opisuje kako se stanje jednog objekta menja kroz interakcije u koje objekat ulazi. Dijagram stanja sadrži sva stanja u kojima se posmatrani objekat može naći, pri čemu su početno i završno stanje posebno označena, kao i sve moguće prelaze iz jednog u drugo stanje i imena događaja koji odgovaraju tim prelazima.

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

Dijagram slučajeva upotrebe opisuje jednu funkcionalnu celinu sistema i kao takav ima ulogu da na vizualan način prikaže elemente funkcionalne specifikacije sistema. Njegovi elementi su slučajevi upotrebe, akteri koji učestvuju u njima i njihovi međusobni odnosi. Dodatno, on može sadržati i pakete i podsisteme. Važno je napomenuti da svaki slučaj upotrebe mora biti propraćen njegovim opisom u tekstualnom obliku.

65. Objasniti odnose između slučajeva upotrebe na dijagramima slučajeva upotrebe.

Odnosi između slučajeva upotrebe se označavaju isprekidanim strelicama. Uobičajeni odnosi su "include" koji označava da slučaj od kog ide strelica obuhvata čitav slučaj prema kome ide strelica (npr. upis godine ---include--> izbor predmeta) i "extends" koji označava da je slučaj od kog ide strelica moguće proširenje drugog slučaja u vidu neobaveznog dodatka (npr. uputstvo ---extends--> upis godine).

66. Šta obuhvata opis jednog slučaja upotrebe?

Mora sadržati bar naziv, aktere, kratak opis, preduslove (uslovi koji moraju biti ispunjeni da bi akteri mogli da započnu slučaj upotrebe), postuslove (uslovi koji moraju biti ispunjeni nakon završetka slučaja upotrebe), opis

toka slučaja upotrebe, opis posebnih slučajeva, dijagrame koji tačnije opisuju slučaj upotrebe (najčešće dijagram aktivnosti ili sekvence).

67. Navesti dijagrame interakcija UML-a.

- Dijagram sekvence
- Dijagram komunikacije
- Dijagram vremena
- Pregledni dijagram interakcija

68. Koja je uloga i šta su osnovni elementi dijagrama komunikacije?

Dijagram komunikacije ilustruje objekte, njihove međusobne odnose i poruke koje razmenjuju, sa posebnom pažnjom posvećenom strukturnoj organizaciji tih objekata. Sadrži objekte (ponekad i subjekte), poruke koje oni razmenjuju, komentare i napomene.

69. Koja je uloga i šta su osnovni elementi preglednog dijagrama interakcije?

Pregledni dijagram interakcija je varijanta dijagrama aktivnosti (opis poslovnih procesa i toka podataka između njih) u kojoj je akcenat na upravljanju procesa ili sistemom. Svaki čvor, tj. aktivnost u dijagramu može da predstavlja neki drugi dijagram interakcija ili aktivnosti. Sadrži objekte, manje dijagrame aktivnosti ili interakcija, slučajeve upotrebe, tok odvijanja procesa, grananja i spajanja, početak i kraj.

70. Koja je uloga i šta su osnovni elementi dijagrama sekvence?

Dijagram sekvence služi za opisivanje toka odvijanja slučaja upotrebe. Pomoću njega se predstavlja koji subjekat ili objekat je zadužen za koji korak slučaja upotrebe, kao i kojim redom se ti koraci odvijaju i kako se prenose odgovornosti sa jednog na drugi subjekat ili objekat. Na najnižem nivou apstrakcije je predstavljena implementacija u vidu poziva metoda (tj. slanja poruka između objekata), pa je sama implementacija u kodu trivijalna na osnovu takvog dijagrama sekvence. Vertikalna dimenzija u dijagramu sekvence predstavlja tok vremena (od vrha ka dnu). Horizontalno se dijagram deli na prostore koji odgovaraju pojedinačnim objektima. Svaka kolona u vrhu ima identifikaciju objekta, a ispod nje se nalazi isprekidana vertikalna linija koja predstavlja životni vek tog objekta. Na toj liniji (životnog veka) se crtaju uski pravougaonici koji predstavljaju okvire aktivnosti objekta koji započinju prijemom poruke, a završavaju se kada se ta primljena poruka obradi. Poruke koje se šalju između objekata se predstavljaju strelicama.

71. Šta je projekat softvera?

Projekat softvera je plan prema kome će se implementirati softver.

72. Šta čini projekat softvera?

Projekat softvera čine razni dokumenti i nacrti iz kojih se može razumeti šta se pravi, zašto se pravi i kako se pravi. Vrste tih dokumenata i nacrta zavise od razvojne metodologije koja se koristi, ali neki primeri su vizija, specifikacija zahteva, analiza rizika, životni ciklus, kadrovski plan, plan infrastrukture, model domena, implementacioni model, plan dokumentovanja, plan testiranja i slično.

73. Šta je metodologija razvoja softvera?

Razvojna metodologija predstavlja način posmatranja i pristupa celom projektu. Izabrana metodologija preporučuje kojim modelima treba pridavati veći značaj, u kojim fazama razvoja se prave određeni modeli, specijalnosti razvijalaca koji prave te modele i drugo.

74. Objasniti ukratko kako izgleda životni ciklus projekta u klasičnim razvojnim metodologijama.

U klasičnim razvojnim metodologijama, životni ciklus projekta bi podrazumevao striktno određeni redosled obavljanja određenih poslova. Faze projektovanja su bile jasno razdvojene od faza implementacije, pa je postojao

niz faza kroz koji je projekat prolazio od početka do završetka (npr. analiza -> projektovanje -> kodiranje -> testiranje -> održavanje).

75. Šta je model vodopada?

Model vodopada je model životnog ciklusa koji se obično primenjivao u klasičnim razvojnim tehnologijama. Podrazumevao je niz razdvojenih faza kroz koje bi projekat prolazio u jednom smeru, bez mogućnosti vraćanja unazad (npr. analiza -> projektovanje -> kodiranje -> testiranje -> održavanje). Ovo je dovodilo do raznih problema, naročito u slučaju da je kasno uočena greška u nekoj ranijoj, već završenoj, fazi (najviše zbog toga što su često sasvim različiti timovi radili na svakoj od faza).

76. Kakvo je mesto projektovanja u agilnim razvojnim metodologijama (ukratko)?

Za razliku od klasičnih metodologija, u agilnim metodologijama faze razvoja više nisu strogo razdvojene. Ovo dovodi do toga da se analiziranje, projektovanje i implementacija rade onda kada je to potrebno, a ne u unapred propisanim fazama. Ovakav način razvoja zahteva da razvojni tim bude u stanju da vrši sve što je potrebno u projektu (analiza, projektovanje, implementacija, ...).

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

Dizajn softvera se može shvatiti skoro kao sinonim za projekat softvera, tj. plan po kome će se implementirati softver. Međutim, on često ne obuhvata elemente projektovanja koji su bliži poslovnoj strani problema (vizija, formalizacija zahteva), kao ni elemente planiranja samog razvojnog procesa i infrastrukture. Dakle, fokus je na opisivanju i modeliranju funkcionalnih i strukturnih elemenata softvera. Arhitektura softvera predstavlja strukturu softverskog sistema na apstraktnom nivou. Prema ovim definicijama, dizajn je opštiji pojam od arhitekture, tj. svaka arhitektura je dizajn, ali nije svaki dizajn arhitektura. Štaviše, jedan od kriterijuma za prepoznavanje arhitekture je cena - arhitektura predstavlja značajne odluke o dizajnu koje daju oblik sistemu, gde je značajnost određena cenom pravljenja izmena.

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

Apstrahovanje je uopštavanje karakteristika nekog posmatranog problema i njegovih mogućih rešenja. Dobrim apstrahovanjem dobijamo na opštosti rešenja (a time i potencijalnu mogućnost za njegovu ponovnu upotrebu).

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

Dekompozicija je postupak uvođenja sve više pojedinosti u uopšteni model softvera kroz prepoznavanje manjih celina koje čine taj sistem. Ovim postupkom se dobija konkretnija i jasnija struktura rešenja.

80. Koje su osnovne vrste dekomponovanja?

Funkcionalno dekomponovanje, logičko dekomponovanje i, u poslednje vreme, dekomponovanje prema promenljivosti.

81. Šta je funkcionalna dekompozicija?

Funkcionalna dekompozicija je podela celine na manje komponente koje predstavljaju neke funkcionalne celine. U idealnom slučaju, svaka komponenta bi trebalo da ima prepoznatljivu ulogu u većem sistemu i da obavlja samo jednu funkciju tako da je ona za nju u potpunosti odgovorna. Komponente međusobno sarađuju preko interfejsa, ali tako da je (bar bi tako trebalo da bude) njihova saradnja što manjeg obima.

82. Šta je dekompozicija prema promenljivosti?

U slučaju dekompozicije prema promenljivosti, ideja je da se prepoznaju glavne tačke promenljivosti (mesta na kojima može doći do izmena usled promena okolnosti ili prilagođavanja dela softvera za druge primene) i glavne ose promenljivosti (pravci u kojima se promene distribuiraju - najčešće odgovaraju zavisnostima delova sistema

od tačaka promenljivosti) i da se zatim celina podeli na delove tako da se tačke promenljivosti što više razdvoje, a ose promenljivosti što više grupišu.

83. Šta je logička dekompozicija?

Logička dekompozicija predstavlja podelu celine na pakete (ili druge strukturne celine) koji grupišu delove implementacije koji se zajedno razvijaju ili imaju izražene unutrašnje međuzavisnosti strukture ili ponašanja.

84. Objasniti odnos apstrahovanja i dekomponovanja.

Iako su apstrahovanje i dekomponovanje naizgled suštinski suprotni postupci, potrebna je usklađena i naizmenična upotreba oba koncepta da bi se dobio dobar dizajn. Apstrahovanje daje na opštosti rešenja, a dekomponovanje na preciznosti njegove strukture. Suprotnost ova dva koncepta je najizraženija kada se primenjuje funkcionalno komponovanje (apstrakcija) i funkcionalno dekomponovanje.

85. Koje su poželjne osobine softvera, iz ugla projektovanja?

Najvažnije osobine su ispravnost i efikasnost, mada je ovo više iz ugla korisnika i razvojnog tima, a ne iz ugla projektovanja. Pored toga, ovaj put iz ugla projektovanja, imamo i fleksibilnost, proširivost i modularnost softvera. Dodatne bitne karakteristike su razdvojenost odgovornosti komponenti, preciznost i jasnoća interfejsa, doslednost enkapsulacije, visoka kohezija i niska spregnutost.

86. Šta je fleksibilnost softvera?

Fleksibilnost softvera je potencijal da se on ili njegovi delovi uz relativno male izmene prilagode promenama u okruženju ili da se primenjuju u sasvim novim okolnostima.

87. Šta je proširivost softvera?

Proširivost softvera je potencijal da se značajan broj dopuna implementira samo uz dodavanje novih elemenata softvera uz eventualne minimalne izmene u postojećim delovima softvera.

88. Šta su kohezija i spregnutost u kontekstu razvoja softvera?

Kohezija i spregnutost su apstraktne mere koje se koriste za ocenjivanje kvaliteta softvera. Kohezija je stepen međusobne povezanosti elemenata jednog modula, a spregnutost je stepen međusobne povezanosti elemenata različitih modula.

89. Navesti vrste kohezije u kontekstu razvoja softvera.

Poređane redom od najviših i najpoželjnijih vrsta ka najslabijim i najmanje poželjnim:

- Funkcionalna kohezija
- Sekvencijalna kohezija
- Komunikaciona kohezija
- Proceduralna kohezija
- Vremenska kohezija
- Logička kohezija
- Koincidentna kohezija

90. Objasniti funkcionalnu koheziju u kontekstu razvoja softvera.

Funkcionalna kohezija se odnosi na povezanost delova modula (metoda unutar klase, klasa unutar komponente, ...) na osnovu međusobne funkcionalne zavisnosti, a u cilju ostvarivanja funkcije za koju je taj modul odgovoran. U kontekstu funkcionalne dekompozicije, okupljanje svih delova komponente u istu komponentu sa jednim ciljem tako da ni jedan od tih delova nije suvišan bi donosilo visoku funkcionalnu koheziju. Ovo je najviša i najpoželjnija vrsta kohezije.

91. Objasniti sekvencijalnu koheziju u kontekstu razvoja softvera.

Sekvencijalna kohezija znači da su elementi modula projektovani tako da rezultat jednog elementa predstavlja ulazne podatke nekog drugog elementa unutar tog modula. Ovakva kohezija, ukoliko se zaobiđe nejasna odgovornost u odnosu na posao kao celinu (koja nastaje ukoliko delovi sekvencijalne obrade rade skroz različite poslove) i ako elementi nemaju javne interfejse koji koriste drugi moduli, je sasvim prihvatljiva.

92. Objasniti komunikacionu koheziju u kontekstu razvoja softvera.

Kohezija je komunikaciona ako su elementi modula prikupljeni zato što koriste iste podatke. Iako je često da ovakav modul ima više odgovornosti koje su dalje podeljene na više modula, ovakav vid kohezije može biti prihvatljiv.

93. Objasniti proceduralnu koheziju u kontekstu razvoja softvera.

Kohezija je proceduralna ukoliko su elementi modula prikupljeni zato što se koriste pri obavljanju nekog celovitog posla. Za razliku od sekvencijalne kohezije, ovde elementi ne rade sekvencijalno. Često se događa da nema funkcionalne zavisnosti između elemenata i da oni obavljaju skroz različite poslove, što ukazuje na nejasno razdvojene odgovornosti. Proceduralna kohezija se ponekad može opravdati u pravljenju logičkih celina poput paketa.

94. Objasniti vremensku koheziju u kontekstu razvoja softvera.

Vremenska kohezija znači da su elementi modula prikupljeni zato što se koriste u istom periodu vremena - npr. prikupljanje elemenata inicijalizacije nekog sistema (oni mogu raditi potpuno drugačije poslove, ali se svi koriste u periodu inicijalizacije). Poslovi koje ti elementi obavljaju su najčešće nepovezani, što ukazuje na nejasnu podelu odgovornosti.

95. Objasniti logičku koheziju u kontekstu razvoja softvera.

Kohezija je logička ako su elementi modula prikupljeni u celinu zato što imaju logički sličnu (ili istu) ulogu u sistemu. Primer ovoga bi bilo stavljanje različitih operacija čitanja i parsiranja za različite formate dokumenata u isti modul. Ovakvi moduli najčešće imaju veći broj odgovornosti koje mogu, ali ne moraju, istovremeno biti podeljene na više modula.

96. Objasniti koincidentnu koheziju u kontekstu razvoja softvera.

Kohezija je koincidentna ako su elementi modula međusobno potpuno odvojeni ili veoma slabo povezani. U ovom slučaju, elementi modula su praktično slučajno okupljeni na isto mesto (npr. zbog pisanja istog dana ili istim alatom, zbog upotrebe sličnih interfejsa, ...). Odgovornosti uopšte nisu razdvojene u ovom slučaju i ovakav vid kohezije je nedopustiv.

97. Navesti osnovne karakteristike spregnutosti u kontekstu razvoja softvera.

- Vrsta sprege
- Nivo sprege
- Širina sprege
- Smer spregnutosti
- Način ostvarivanja sprege
- Intenzitet spregnutosti

98. Zašto je spregnutost komponenti softvera potencijalno problematična?

Visoka spregnutost označava jaku povezanost nekog modula sa drugim modulima. Ukoliko je interfejs između takvih modula širok (nije čist i jednostavan), to ukazuje na loše izvedenu dekompoziciju i podelu poslova. Visoka

sprega takođe dovodi do veće mogućnosti da su potrebne izmene u zavisnim modulima nakon promene nekog drugog, spregnutog modula.

99. Navesti i ukratko objasniti vrste spregnutosti u kontekstu razvoja softvera.

- Sprega logike - najnepoželjnija vrsta sprege, odlikuje se deljenjem informacija (jedan modul neposredno pristupa informacijama koje se održavaju od strane drugog modula) ili pretpostavki (jedan modul mora da vodi računa o pretpostavkama o načinu implementacije drugog modula) između modula.
- Sprega tipova - ponekad prihvatljiva vrsta sprege, odlikuje se korišćenjem tipa u jednom modulu koji je definisan u drugom modulu.
- Sprega specifikacije - najpoželjnija vrsta sprege, apstraktnija od sprege tipova po tome što se pretpostavlja da nije poznat konkretan tip koji se koristi, već postoje samo pretpostavke o njegovom interfejsu.

100. Objasniti spregu logike u kontekstu razvoja softvera.

Sprega logike je najnepoželjnija vrsta sprege. Odlikuje se deljenjem informacija (jedan modul neposredno pristupa informacijama koje se održavaju od strane drugog modula) ili pretpostavki (jedan modul mora da vodi računa o pretpostavkama o načinu implementacije drugog modula) između modula. U oba slučaja se narušava princip enkapsulacije.

101. Objasniti spregu tipova u kontekstu razvoja softvera.

Sprega tipova je ponekad prihvatljiva vrsta sprege. Odlikuje se korišćenjem, uz puno poznavanje njihovih specifičnosti, tipova u jednom modulu koji su definisani u drugom modulu. Ovakva sprega može biti određena (komponenta koja korsiti tip čak i pravi objekte tog tipa) ili neodređena (objekti tog tipa se samo koriste, a pravljenje se prepušta matičnoj komponenti). Poželjnija je neodređena. Ukoliko je enkapsulacije dobro urađena na tipu o kojem je reč, ova vrsta sprege može biti prihvatljiva.

102. Objasniti spregu specifikacije u kontekstu razvoja softvera.

Sprega specifikacije je najpoželjnija vrsta sprege, apstraktnija od sprege tipova po tome što se pretpostavlja da nije poznat konkretan tip koji se koristi, već samo pretpostavke o njegovom interfejsu. Ona nastupa kada modul koji koristi neki tip iz drugog modula zapravo koristi apstraktni tip koji radi sa bilo kojim konkretnim tipom koji odgovara datoj specifikaciji. Ovo se najčešće ostvaruje uz pomoć polimorfizma.

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

Poređani redom od najjačeg do najslabijeg:

- Spregnutost po sadržaju - otvoreno i neposredno pristupanje i/ili menjanje sadržaja jedne komponente od strane druge komponente.
- Spregnutost preko zajedničkih delova - dve ili više komponenti neposredno pristupaju nekim deljenim podacima koji nisu njihov sastavni deo.
- Spoljašnja spregnutost - upotreba deljenog i sa strane nametnutog koncepta (interfejs, format podataka, komunikacioni protokol, ...) od strane više komponenti.
- Spregnutost preko kontrole - komponenta upravlja radom druge komponente, pri čemu upravljana komponenta ne može da funkcioniše samostalno, tj. bez spoljašnje kontrole.
- Spregnutost preko markera - više komponenti međusobno razmenjuje neku složenu strukturu podataka koju koriste na različite načine.
- Spregnutost preko podataka - komponenta koristi interfejs druge komponente putem koga mu prenosi pojedinačne podatke.
- Spregnutost preko poruka - komponenta koristi interfejs druge komponente bez prenošenja podataka.

104. Objasniti spregnutost po sadržaju u kontekstu razvoja softvera.

Spregnutost po sadržaju predstavlja otvoreno i neposredno pristupanje i/ili menjanje sadržaja jedne komponente od strane druge komponente. U ovom slučaju, enkapsulacija je narušena i u pitanje se dovode odgovornosti komponenti. Izmene u jednoj komponenti u slučaju ovakve sprege najčešće zahteva izmene i u zavisnoj komponenti, pa je održavanje veoma teško. Ovo je najviši nivo spregnutosti i time i najnepoželjniji.

105. Objasniti spregnutost preko zajedničkih delova u kontekstu razvoja softvera.

Spregnutost preko zajedničkih delova znači da dve ili više komponenti neposredno pristupaju nekim deljenim podacima koji nisu njihov sastavni deo. U ovom slučaju, iako to nije očigledno, enkapsulacija nije ni uspostavljena, jer suštinski ove komponente pristupaju jedna drugoj obostrano jer tretiraju zajedničke delove kao svoje podatke. Ovaj nivo spregnutosti je visok i veoma je nepoželjan.

106. Objasniti spoljašnju spregnutost u kontekstu razvoja softvera.

U slučaju spoljašnje spregnutosti, više komponenti koristi jedan deljeni i sa strane nametnuti koncept (interfejs, format podataka, komunikacioni protokol, ...). Problem u ovom slučaju je taj što često dolazi do ponavljanja istog ili veoma sličnog koda koji koristi taj spoljni koncept, pa ako dođe do promena u njemu, onda će vrlo verovatno biti potrebne izmene i u komponentama koje ga koriste. U slučaju da je taj koncept stabilan i ako ima relativno uzak interfejs, onda ovakav nivo spregnutosti može biti prihvatljiv (u suprotnom se spoljni koncept može umotati u novu komponentu koja ima uži interfejs).

107. Objasniti spregnutost preko kontrole u kontekstu razvoja softvera.

Spregnutost preko kontrole podrazumeva da jedna komponenta upravlja radom druge komponente, pri čemu ta komponenta ne može da funkcioniše samostalno, tj. bez spoljašnje kontrole. Primer ovoga bi bio kad bi kontrolisana komponenta implementirala niži nivo neke klase operacija, a viši nivo upravljanja prepustila kontrolerima (npr. niži nivo operacija sa slikama i kontroler koji primenjuje te operacije i time dobija složene). Potencijalni problemi su nejasna podela odgovornosti ukoliko komponenta kontroler ima druge odgovornosti osim upravljanja kontrolisanom komponentom, kao i problemi ukoliko jednu komponentu kontroliše više kontrolera (promena interfejsa kontrolisane komponente bi napravila propagaciju izmena kroz kontrolere). Ako se ovi problemi zaobiđu, ovaj nivo spregnutosti je najčešće prihvatljiv.

108. Objasniti spregnutost preko markera u kontekstu razvoja softvera.

U slučaju spregnutosti preko markera, više komponenti međusobno razmenjuje neku složenu strukturu podataka (tj. marker) koju upotrebljavaju na razne načine. U slučaju da marker nema ponašanje, već se njegovi podaci neposredno koriste, ipak je u pitanju spregnutost preko zajedničkih delova, a ako marker ima definisano ponašanje koje se uvek koristi na sličan način, to liči na spregnutost preko kontrole. Potencijalni problemi sa ovim nivoom spregnutosti su nedefinisana odgovornost markera (neki aspekti ponašanja su prepušteni drugim komponentama), kao i mogućnost sukoba nadležnosti među komponentama koje ga koriste ako marker nije strogo enkapsuliran (što je najčešći slučaj). Spregnutost preko markera je prihvatljiva u slučajevima da se ovi problemi zaobiđu.

109. Objasniti spregnutost preko podataka u kontekstu razvoja softvera.

Spregnutost preko podataka podrazumeva da jedna komponenta koristi interfejs druge komponente putem koga mu prenosi pojedinačne podatke. Ukoliko su podaci koji se prenose složeni, onda ovo liči na spregnutost preko zajedničkih delova. Ovaj nivo spregnutosti je jedan od najnižih i najčešće sasvim prihvatljiv. Problemi mogu da nastanu iz loše definisanih interfejsa ili različitog tumačenja podataka koji se razmenjuju.

110. Objasniti pojam širine sprege u kontekstu razvoja softvera.

Širina sprege između dve komponente odgovara broju elemenata jedne komponente (objekata, podataka, metoda, poruka, ...) koje upotrebljava druga komponenta. U slučaju da je spregnutost dvosmerna, onda širina sprege zavisi i od smera. Poželjno je da sprega bude što uža (jer što je uža, to je manja verovatnoća da se izmene u jednoj od komponenti propagiraju na drugu). Široka sprega može sugerisati da nije jasna podela

odgovornosti između te dve komponente (u kom slučaju možemo implementirati neke složenije celine u okviru prve komponente radi sužavanja njenog interfejsa) ili da je odgovornost neke komponente preširoka (u kom slučaju možemo tu komponentu podeliti na više manjih).

111. Objasniti pojam smer sprege u kontekstu razvoja softvera.

Spregnutost između dve komponente može biti jednosmerna ili dvosmerna. Ako je jednosmerna, tada jedna komponenta koristi elemente druge, ali ne i obrnuto. Ako je dvosmerna, onda svaka od dveju komponenti koristi elemente one druge. Spregnutost može biti i cirkularna (tranzitivna cirkularna spregnutost više od dve komponente). Najpoželjnije je da sprega bude jednosmerna zbog uvećane složenosti sistema koju ostale dve vrste donose.

112. Objasniti pojam statička spregnutost u kontekstu razvoja softvera.

Statička spregnutost je eksplicitno izražena u programskom kodu, tj. ostvaruje se pre samog izvršavanja programa (npr. klasa B nasleđuje klasu A ili neki podatak klase A je objekat klase B). Statička spregnutost je intenzivnija od dinamičke i kao takva često ima odlike spregnutosti tipova.

113. Objasniti pojam dinamička spregnutost u kontekstu razvoja softvera.

Dinamička spregnutost se ostvaruje u toku izvršavanja programa, tj. nije eksplicitno iskazana u programskom kodu, već npr. zavisi od konfiguracije, ulaznih parametara i slično (npr. razvoj vođen događajima u Qt-u). Dinamička spregnutost je manje intenzivna od statičke, pa time i poželjnija. Ona najčešće ima odlike spregnutosti specifikacije.

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

Statička spregnutost je intenzivnija i time nepoželjnija od dinamičke. Pošto je statička spregnutost relativno čvrsto uspostavljena u samom kodu, ona najčešće ima odlike spregnutosti tipova, a dinamička spregnutosti specifikacije.

115. Objasniti pojam intenzitet spregnutosti u kontekstu razvoja softvera.

Intenzitet spregnutosti je složena karakteristika koja u obzir uzima sve ostale (vrsta sprege, nivo sprege, širina sprege, smer sprege i način ostvarivanja sprege). Važi da logička sprega ima veći intenzitet od sprege tipova, koja ima veći intenzitet od sprege specifikacije. Viši nivo spregnutosti i širina sprege donose veći intenzitet. Takođe, važi da je cirkularna sprega većeg intenziteta od dvosmerne, a ona od jednosmerne, kao i to da je statička sprega većeg intenziteta od dinamičke.

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

Da bi dobili meru sprege, možemo posmatrati npr. broj drugih komponenti (ili broj elemenata drugih komponenti) koje se referišu iz neke komponente, broj drugih komponenti iz kojih se referiše ta komponenta (ili broj elemenata te komponente koji se referišu iz drugih komponenti), karakteristike sprege i slično. Što se tiče računanja intenziteta sprege, numeričku ocenu bi mogli da dobijemo, na primer, na sledeći način:

- Spregnutost dve komponente: 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))

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

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

Ova pravila se mogu objasniti na sledeći način - što je komponenta složenija, to je ona podložnija izmenama, pa ako je ona jako spregnuta, izmene će se propagirati kroz zavisne (spregnute) komponente.

118. Navesti nekoliko uobičajenih načina spregnutosti komponenti.

- Arhitektura klijent-server
- Hijerarhija pripadnosti
- Cirkularna spregnutost
- Sprega putem interfejsa
- Sprega putem parametara metoda

119. Objasniti karakteristike spregnutosti u slučaju arhitekture klijent-server.

Komponenta server pruža usluge, a komponenta klijent te usluge koristi. Ovakva sprega je jednosmerna, poželjno uska (server treba da pruža samo jedan tip usluga), obično relativno nizak nivo sprege (preko poruka ili preko podataka). Vrsta sprege je sprega tipova ili specifikacija, a nizak intenzitet sprege preporučuje ovu vrstu sprege kao idealnu.

120. Objasniti karakteristike spregnutosti u slučaju hijerarhije pripadnosti.

Komponenta roditelj sadrži i vrši koordinaciju nad više komponenti dece, koje zapravo obavljaju funkciju ili delove funkcije, tako da svaka komponenta dete obavlja svoj deo odgovornosti. Ovakva sprega je obično jednosmerna, relativno uska, ima relativno nizak nivo sprege (preko poruka ili preko podataka), vrsta sprege je sprega tipova ili specifikacija.

121. Objasniti karakteristike spregnutosti u slučaju cirkularne spregnutosti.

Komponenta roditelj sadrži više komponenti dece, gde svako dete zna ko mu je roditelj i pristupa mu radi određenih poslova. Ovakva sprega je dvosmerna, potencijalno široka, obično ima relativno nizak nivo sprege (preko poruka ili preko podataka). Vrsta sprege je sprega tipova ili specifikacija.

122. Objasniti karakteristike spregnutosti u slučaju sprege putem interfejsa.

Komponenta B deklariše i implementira interfejs, a komponenta A koristi komponentu B preko tog interfejsa. Ovakva sprega je potencijalno jednosmerna, širina joj zavisi od dizajna interfejsa, nivo sprege je najčešće relativno nizak (preko poruka ili preko podataka), a vrsta sprege je sprega tipova ili specifikacija.

123. Objasniti karakteristike spregnutosti u slučaju sprege putem parametara metoda.

Komponenta A deklariše metod koji kao parametar prima komponentu B, a komponenta BB, koja je specijalizacija komponente B, se predaje na upotrebu komponenti A kao parametar tog metoda. Ovakva sprega je potencijalno jednosmerna, širina joj zavisi od dizajna interfejsa, nivo sprege je obično preko markera, a vrsta je obično sprega tipova ili specifikacija.

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

Kohezija klase predstavlja stepen međupovezanosti elemenata te klase. Poželjno je da stepen kohezije bude što viši, a ukoliko je nizak, to ukazuje na to da klasa nije dobro oblikovana. Tada je moguće da postoji kohezija između nekih elemenata klase, ali ne svih, ili da unutar nje postoji više različitih skupova elemenata između kojih postoji snažna kohezija. U ovom slučaju je moguće da postoji problem prepoznavanja odgovornosti klase, pa je klasu možda potrebno podeliti na više klasa.

125. Objasniti odnos koncepta klase i pojma spregnutosti u kontekstu OO razvoja softvera.

Spregnutost u ovom slučaju predstavlja stepen međupovezanosti različitih klasa. Poželjno je da nivo spregnutosti bude što manji (ali, naravno, ne i da uopšte ne postoji, jer kako bi onda klase komunicirale). Ukoliko je nivo spregnutosti visok, to ukazuje na to da odgovornosti nisu dobro podeljene između klasa, pa je možda potrebno neke klase spojiti, a neke drugačije podeliti.

126. Šta je obrazac za projektovanje? Čemu služi?

Obrazac za projektovanje je višestruko primenljivo konceptualno rešenje koje odgovara nekoj značajnoj klasi problema. Obrasci za projektovanje služe da daju smernice za rešavanje konceptualno ekvivalentnih (ekvivalentnih na nekom umerenom nivou apstrakcije) problema tako da se očuvaju ili poboljšaju dobre karakteristike projekta i da se izbegnu potencijalni problemi.

127. Koji su osnovni elementi obrazaca za projektovanje? Objasniti ih.

- Naziv obrasca - naziv se bira tako da u što manje reči što bolje opiše problem, rešenje i posledice primene obrasca. Ovo znatno olakšava komunikaciju.
- Problem koji se obrascem rešava - detaljan, ali opet dovoljno apstraktan, opis problema za koji obrazac daje rešenje.
- Rešenje problema - detaljan apstraktan opis elemenata koji čine dizajn rešenja, kao i njihovih odgovornosti i međusobnih odnosa.
- Posledice rešenja - različiti rezultati i ocene primene obrasca.

128. Objasniti ime, kao element obrasca za projektovanje.

Ime obrasca treba da u što manje reči što bolje opiše problem, rešenje i posledice primene obrasca. Zadavanjem imena obrascima se olakšava komunikacija, razmena mišljenja i diskutovanje o konceptualnom rešenju, kao i pisanje i čitanje projektne dokumentacije. Imena takođe omogućavaju i pravljenje kataloga i rečnika obrazaca.

129. Objasniti problem, kao element obrasca za projektovanje.

Problem koji se rešava obrascem treba da bude detaljno ali istovremeno i dovoljno apstraktno opisan. On je osnovno sredstvo za prepoznavanje slučajeva u kojim neki obrazac može da se primeni. Poželjno je da opis obuhvata i konkretne primere, kao i opise elemenata problema (pojmova, odnosa, ...) i odgovarajuće dijagrame. Može obuhvatati i spisak uslova koje je potrebno ispuniti da bi obrazac bio uspešno primenljiv.

130. Objasniti rešenje, kao element obrasca za projektovanje.

Rešenje problema predstavlja detaljan apstraktan opis elemenata koji čine dizajn rešenja, kao i njihovih odgovornosti, međusobnih odnosa i toka saradnje. Rešenje ne sme biti ograničeno na opisivanje određenog konkretnog projekta ili implementacije, jer obrazac mora da predstavlja uputstvo koje se može primeniti u različitim slučajevima.

131. Objasniti posledice, kao element obrasca za projektovanje.

Posledice rešenja obuhvataju različite rezultate i ocene primene obrasca. One mogu biti značajne pri razmatranju pogodnosti primene nekog rešenja u odnosu na druga potencijalna rešenja.

132. Navesti šta sve obuhvata opis jednog obrasca za projektovanje.

Ime obrasca, alternativna imena, klasifikacija (omogućava bolje snalaženje u katalogu), namena, motivacija (doprinosi), primenljiivost, preduslovi za primenu, logička struktura rešenja (predstavljena dijagramima i pratećim tekstom, najčešće UML), entiteti rešenja (klase ili objekti koji imaju važnu ulogu u obrascu), saradnja između učesnika u rešenju (opis tokova poruka između objekata, najčešće se koristi UML za opis), posledice primene rešenja, implementacija (uputstvo za primenu obrasca), poznati primeri korišćenja, primeri koda, pregled srodnih obrazaca.

133. Kako su klasifikovani obrasci za projektovanje? Navesti po jedan primer od svake vrste obrazaca.

- Gradivni obrasci - npr. unikat (Singleton)
- Strukturni obrasci - npr. dekorater (Decorator)
- Obrasci ponašanja - npr. posetilac (Visitor)

134. Objasniti namenu gradivnih obrazaca za projektovanje.

Gradivni obrasci služe da apstrahuju postupak pravljenja i povezivanja novih objekata. Oni pokušavaju da od korisnika sakriju što veći deo složenosti pravljenja i povezivanja napravljenih objekata, kao i međusobne odnose objekata i klasa, a ponekad i sam tip tih objekata, a to postižu pružanjem odgovarajućeg interfejsa za pravljenje, povezivanje i korišćenje objekata korisniku, što dalje omogućava da se različiti objekti sa složenim ponašanjem grade na relativno jednostavan način.

135. Navesti bar četiri gradivna obrazaca za projektovanje.

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

136. Objasniti namenu strukturnih obrazaca za projektovanje.

Strukturni obrasci rešavaju probleme organizovanja objekata i klasa u veće funkcionalne celine.

137. Navesti bar pet strukturnih obrazaca za projektovanje.

- Adapter (Adapter)
- Most (Bridge)
- Dekorater (Decorator)
- Fasada (Facade)
- Sastav (Composite)

138. Objasniti namenu obrazaca ponašanja.

Obrasci ponašanja kao jednu od glavnih namena imaju enkapsuliranje složene prirode kontrole i prebacivanje složenosti upotrebe na povezivanje objekata. Oni to rade pomoću uspostavljanja odgovarajućih odnosa među klasama i objektima kroz odgovarajuće načine razmene poruka između njih.

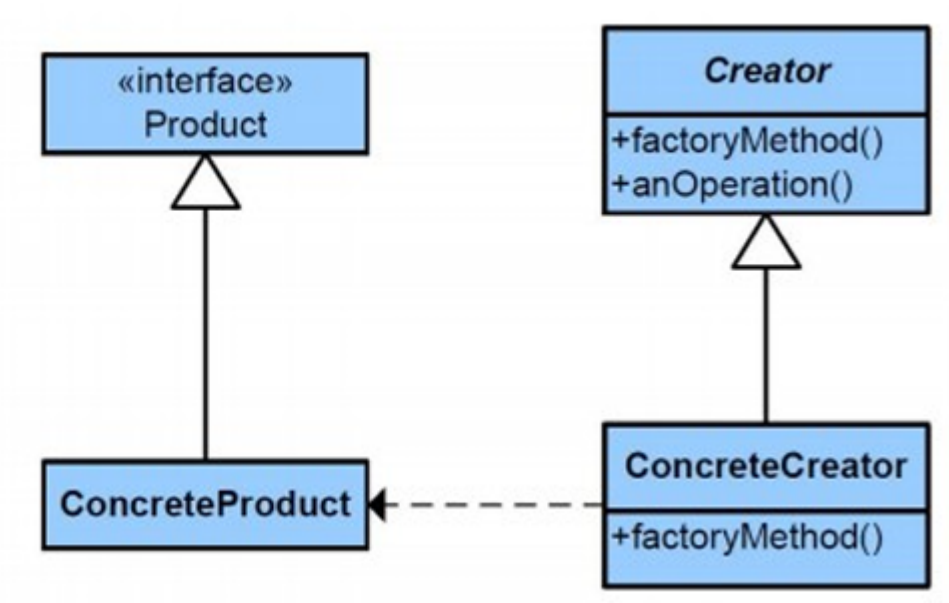
139. Navesti bar sedam obrazaca ponašanja.

- Posetilac (Visitor)
- Posmatrač (Observer)
- Podsetnik (Memento)
- Posrednik (Mediator)
- Iterator (Iterator)
- Strategija (Strategy)
- Šablonski metod (Template Method)

140. Objasniti kada se i kako primenjuje obrazac Proizvodni metod (Factory Method).

Obrazac Proizvodni metod se može koristiti kada je potrebno apstrahovati pravljenje različitih objekata u različitim okolnostima. Ovo se radi tako što se definiše interfejs za pravljenje objekata unutar klase "kreatora" (u vidu apstraktnog metoda), a njenim potklasama se prepušta da taj metod i implementiraju (tj. da naprave i vrate odgovarajući objekat).

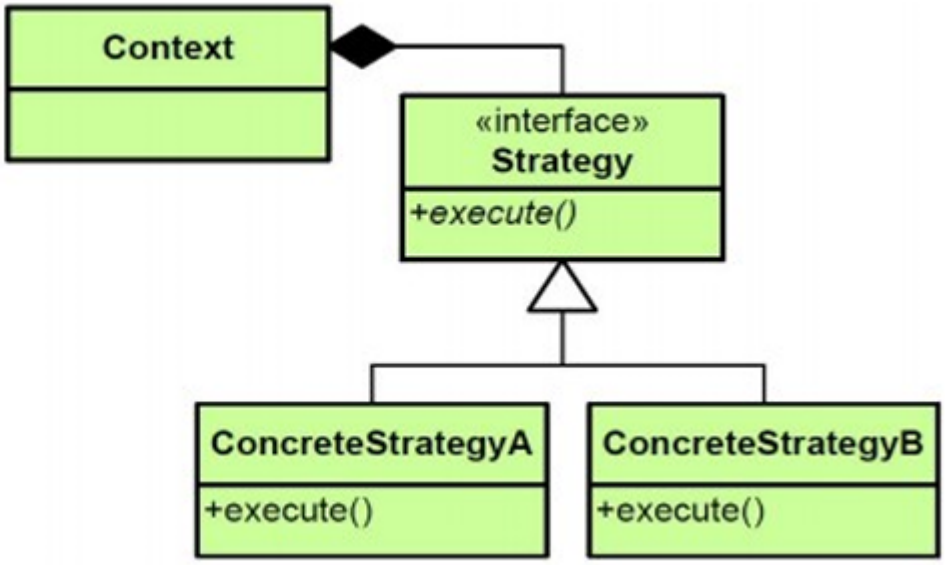
141. Skicirati dijagram klasa obrasca za projektovanje Proizvodni metod (Factory Method).



142. Objasniti kada se i kako primenjuje obrazac Strategija.

Obrazac Strategija se može koristiti ukoliko je potrebno primeniti sličnan ili isti algoritam, ali sa različitim konkretnim koracima. On omogućava dinamički izbor algoritma i može biti alternativa složenim proverama stanja. Ovo se radi tako što se definiše apstraktna klasa algoritama čiji koraci nisu fiksni, a zatim se implementiraju konkretne klase (u vidu hijerarhije klasa), a izbor jedne od njih se prepušta kontekstu.

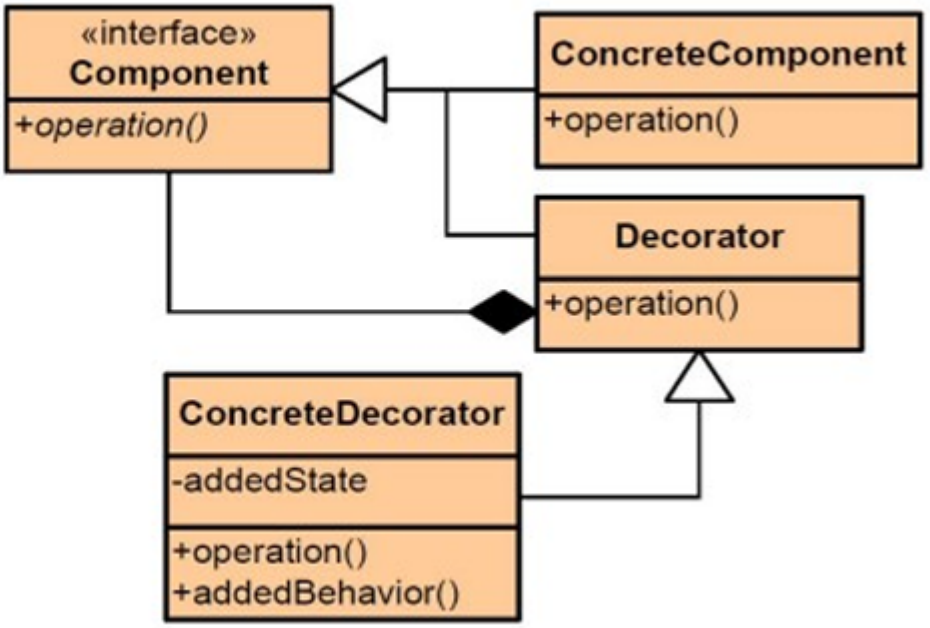
143. Skicirati dijagram klasa obrasca za projektovanje Strategija.



144. Objasniti kada se i kako primenjuje obrazac Dekorater.

Obrazac Dekorater se može primeniti u slučaju da želimo da dinamički dodamo odgovornosti i/ili karakteristike objektima. Ovaj obrazac sprečava potencijalnu eksploziju u veličini klasne hijerarhije ukoliko bi se za svaku kombinaciju karakteristika i odgovornosti pravila posebna klasa. Ovo se radi tako što dekoracija obavlja samo deo posla koji ume, a sve ostalo se prepušta objektu koji dekoriše.

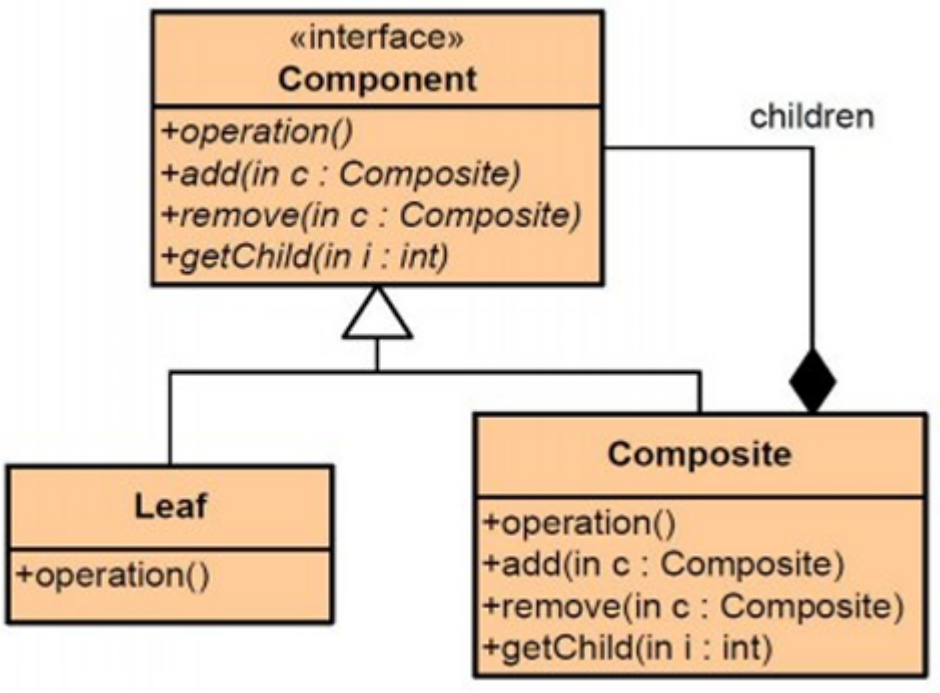
145. Skicirati dijagram klasa obrasca za projektovanje Dekorater.



146. Objasniti kada se i kako primenjuje obrazac Složeni objekat (Sastav, Composite).

Obrazac Sastav se primenjuje kada je potrebno da u hijerarhiji klasa postoji složeni objekat koji je kompozicija više objekata iz iste klasne hijerarhije. To se radi tako što klasa složenog objekta, tzv. sastav, nasleđuje baznu klasu te hijerarhije. On uz to ima i privatnu kolekciju objekata iste te hijerarhije i javne metode za osnovne operacije sa tom kolekcijom (npr. dodavanje i brisanje elemenata iz nje). Takođe implementira i sve potrebne metode hijerarhije.

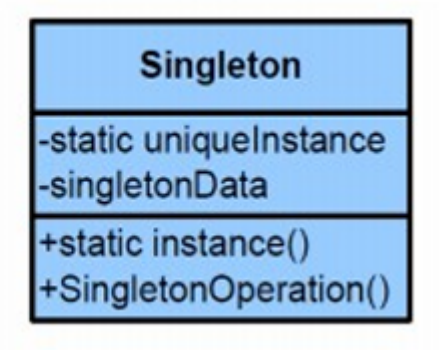
147. Skicirati dijagram klasa obrasca za projektovanje Složeni objekat (Sastav, Composite).



148. Objasniti kada se i kako primenjuje obrazac Unikat (Singleton).

Obrazac Unikat se koristi ukoliko je u programu potrebno da postoji tačno jedan objekat neke klase. Ovo je čest slučaj za neke upravljačke ili kontrolne objekte i slično. Ovo se izvodi tako što se staranje o tom objektu, tzv. unikatu, prepušta samoj klasi unikata, a to se radi tako što se napravi jedan statički metod koji vraća referencu na uvek isti (statički) primerak objekta te klase, a sakrivaju se svi konstruktori tako da je nemoguće instancirati drugi objekat te klase.

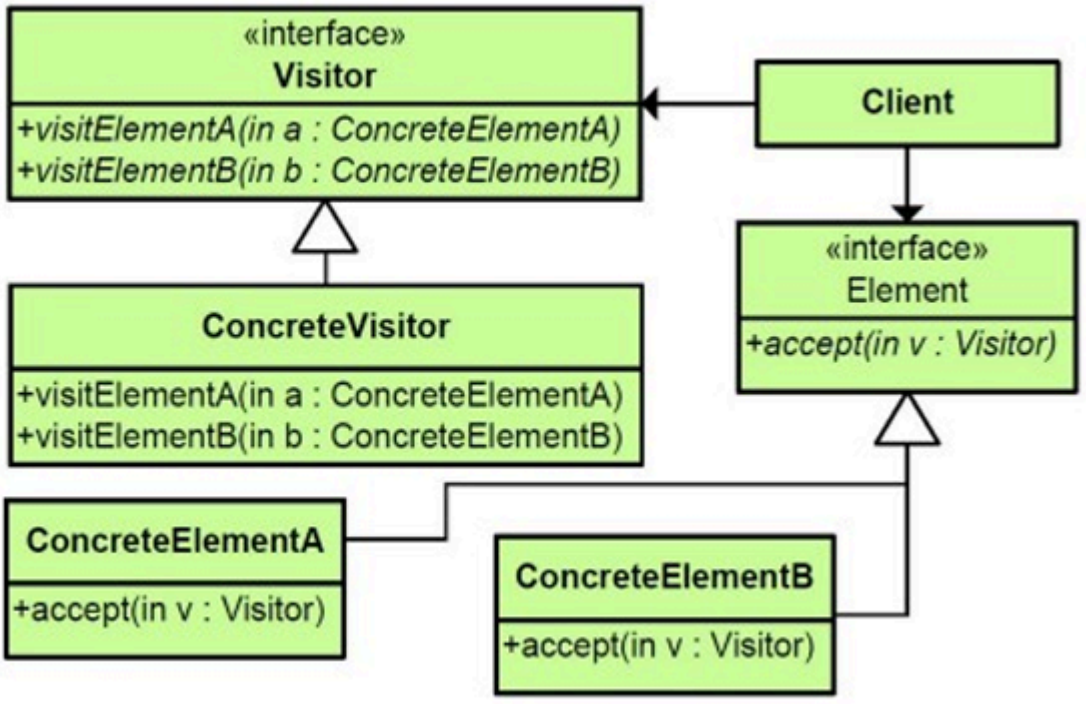
149. Skicirati dijagram klasa obrasca za projektovanje Unikat (Singleton).



150. Objasniti kada se i kako primenjuje obrazac Posetilac (Visitor).

Obrazac Visitor se može upotrebiti ukoliko želimo da razdvojimo operacije koje treba da se izvršavaju na elementima neke strukture od implementacije samih elemenata te strukture (npr. obilaženje grafova, apstraktnog sintaksnog stabla i slično). Zahvaljujući njemu, moguće je dodavanje i menjanje operacija bez menjanja same implementacije strukture, što se postiže prebacivanjem odgovornosti izvođenja operacija na posetioca.

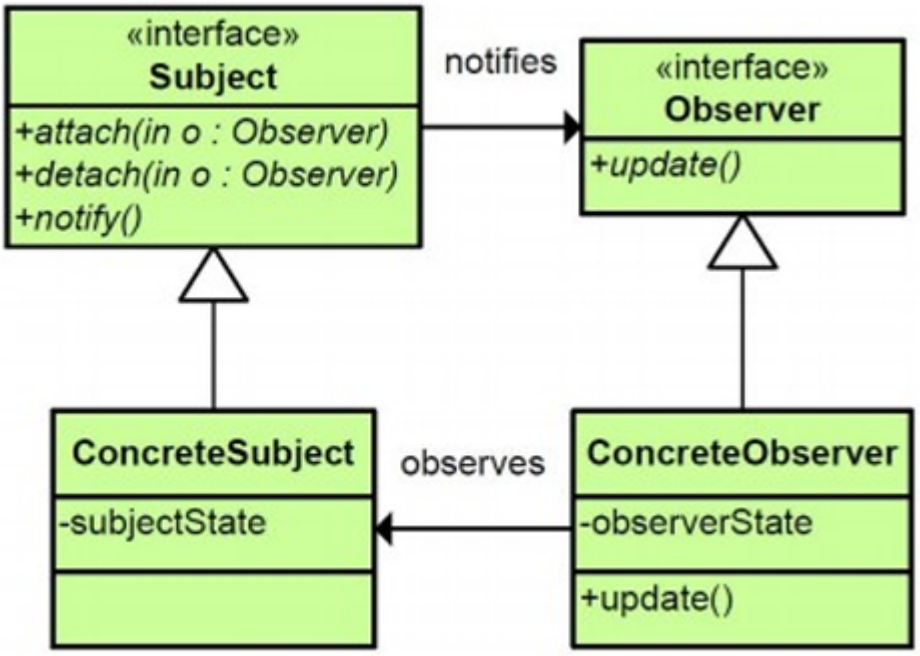
151. Skicirati dijagram klasa obrasca za projektovanje Posetilac (Visitor).



152. Objasniti kada se i kako primenjuje obrazac Posmatrač (Observer).

Često je u sistemu potrebno pratiti stanje nekih objekata i reagovati na promene tog stanja. U tom slučaju može se koristiti obrazac Posmatrač. Ideja je da bude moguće povezivanje više posmatrača sa jednim posmatranim objektom. Ovo se postiže kreiranjem dve klasne hijerarhije - jedna hijerarhija predstavlja objekte čije stanje se može pratiti, a druga predstavlja posmatrače koji prate to stanje. Posmatrani objekti imaju mogućnost da pridruže sebi posmatrače i sami održavaju listu posmatrača koji su im pridruženi. Ukoliko dođe do promene, obaveštavaju sve posmatrače iz te liste.

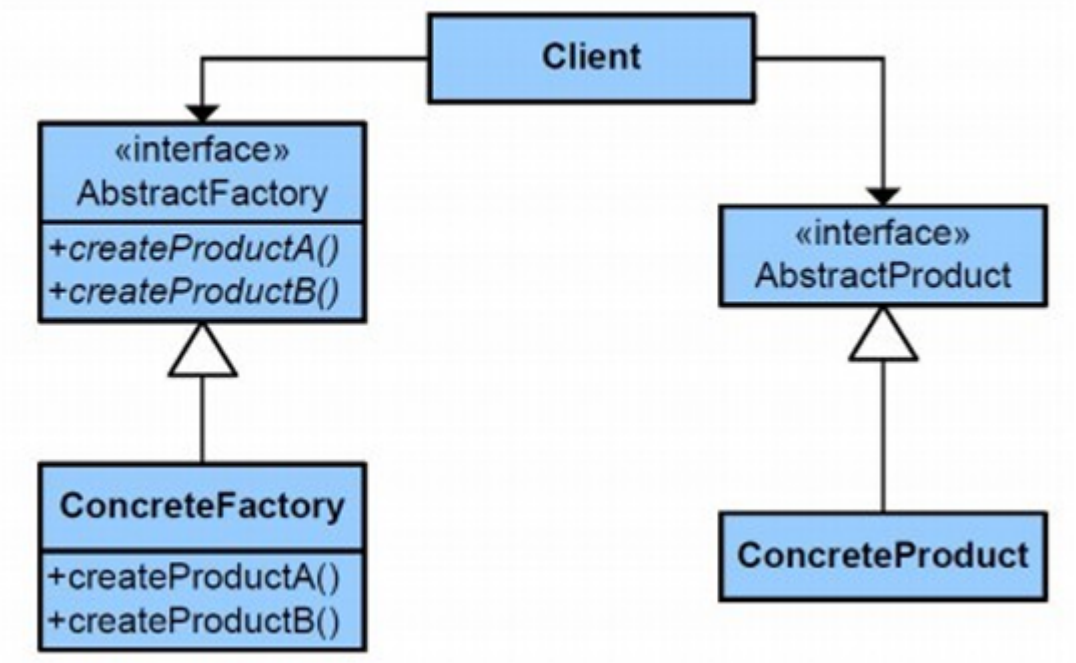
153. Skicirati dijagram klasa obrasca za projektovanje Posmatrač (Observer).



154. Objasniti kada se i kako primenjuje obrazac Apstraktna fabrika (Abstract Factory).

Obrazac Apstraktna fabrika je dobar kandidat za primenu ukoliko je potrebno napraviti familiju sličnih objekata čiji izbor zavisi od nekih okolnosti, kao npr. platforme. Ovaj obrazac često povećava prenosivost programa (npr. pravimo elemente korisničkog interfejsa (npr. dugme), ali pravljenje svakog se razlikuje u zavisnosti od platforme (Windows, Linux, ...)). Ideja je da imamo klasnu hijerarhiju fabrika, koja u osnovi ima apstraktnu fabriku koja pruža interfejs koji ćemo koristiti (napraviDugme(), ...), a nasleđuju je konkretne fabrike (npr. fabrikaWindows, fabrikaLinux) koje implementiraju sve potrebne metode za kreaciju objekata.

155. Skicirati dijagram klasa obrasca za projektovanje Apstraktna fabrika (Abstract Factory).



156. Navesti skupove principa dizajniranja softvera koje smo obradili.

- Ključni principi OO dizajna
- Principi dodeljivanja odgovornosti
- Principi oblikovanja celina

157. Navesti ključne principe OO dizajna.

- Princip jedinstvene odgovornosti (Single Responsibility Principle)
- Princip otvorenosti i zatvorenosti (Open-Closed Principle)
- Princip zamenljivosti (Liskov Substitution Principle)
- Princip razdvajanja interfejsa (Interface Segregation Principle)
- Princip inverzne zavisnosti (Dependency Inversion Principle)

158. Objasniti princip jedinstvene odgovornosti.

"Klasa bi trebalo da ima samo jedan razlog da se menja". Princip se ne mora nužno primenjivati samo na klasama, već to može biti bilo koj drugi strukturni element. Takođe, važno je napomenuti da ovaj princip implicira da klasa treba imati samo jednu odgovornost. Primer narušavanja ovog principa je sledeći - neka je data klasa hijerarhija elemenata nekog programskog jezika koja se može koristiti za predstavljanje apstraktnog sintaksnog stabla. Ukoliko bi svakom elementu dodali metode za proveru ispravnosti, generisanje mašinskog koda i slično, narušili bi jedinstvenu odgovornost te klase, a to je izgradnja interne reprezentacije programskog koda. Umesto ovog pristupa, mogli bi koristiti obrazac za projektovanje Posetilac.

159. Objasniti princip otvorenosti i zatvorenosti.

"Elementi softvera bi trebalo da budu otvoreni za proširivanje i zatvoreni za menjanje". Otvorenost za proširivanje se odnosi na to da bi ponašanje strukturnog elementa trebalo biti proširivo. Zatvorenost za menjanje se odnosi na to da se proširivanje izvodi bez promene postojećeg koda, a primarno interfejsa i postojećeg ponašanja tog elementa. Ovaj princip se obično postiže pomoću uobičajenih OOP tehnika, konkretnije nasleđivanja, dinamičkog vezivanja metoda i definisanja apstraktnih klasa. Na primer, ukoliko želimo da dodamo novo ponašanje, to možemo uraditi dodavanjem nove klase u neku apstraktnu hijerarhiju klasa.

160. Objasniti princip zamenljivosti.

Princip zamenljivosti glasi "Podtipovi moraju da mogu da zamene bazne tipove". Ovaj princip je osnova za hijerarhijski polimorfizam. Međutim, iako se čini da je on možda suvišan, ovaj princip uopšte nije jednostavan. Uzmimo za primer baznu klasu Pravougaonik i klasu koja je nasleđuje Kvadrat. Dodavanjem metoda postaviSirinu u klasu Pravougaonik došlo bi do problema - ako bi unutar klase Kvadrat menjali samo njegovu širinu, on više ne bi bio kvadrat nakon primene ovog metoda, pa bi princip zamenljivosti bio narušen. Drugo potencijalno rešenje bi bilo da u okviru klase Kvadrat u tom metodu menjamo i širinu i visinu, što bi dovelo do toga da se njegova površina menja drugačije od površine za klasu Pravougaonik, pa je princip zamenljivosti opet narušen. Sledi sledeće - da bi se ispoštovao princip zamenljivosti, pri dodavanju metoda u baznu klasu, mora važiti da su preduslovi tog metoda u izvedenoj klasi isti ili blaži nego u baznoj, a da su postuslovi tog metoda u izvedenoj klasi isti ili jači nego u baznoj klasi. Rešenje prethodnog problema bi bilo uvođenje metoda postaviSirinuIVisinu koji izbacuje izuzetak u slučaju neispravnih argumenata.

161. Objasniti princip razdvajanja interfejsa.

"Klijenti (korisnici) ne bi trebalo da budu primorani da zavise od metoda (interfejsa) koje ne koriste". Pod korisnicima, misli se na strukturne elemente programa koji koriste neke interfejse. Princip suštinski nalaže da složene interfejse, koji služe za obavljanje više poslova, treba razbiti na više manjih interfejsa. Narušavanje ovog principa često ukazuje na lošu podelu odgovornosti među klasama ili na preplitanje odgovornosti između delova hijerarhije klasa (ili čak i različitih hijerarhija).

162. Objasniti princip inverzne zavisnosti.

"Moduli visokog nivoa ne smeju da zavise od modula niskog nivoa. I jedni i drugi bi trebalo da zavise od apstrakcija". Jedna, alternativna, definicija glasi "Programirati prema interfejsima, a ne prema implementacijama". Upotreba ovog principa često uvodi nove, apstraktne, elemente programa, što može predstavljati faktor povećanja složenosti programa, ali dobit je daleko veća zbog čistijih zavisnosti i lakšeg razumevanja i održavanja koda.

163. Navesti principe dodeljivanja odgovornosti (dizajn softvera).

- Princip Informacioni ekspert
- Princip Stvaralac
- Princip Visoka kohezija
- Princip Niska spregnutost
- Princip Kontroler
- Princip Polimorfizam
- Princip Izmišljotina
- Princip Indirekcija
- Princip Izolovane promenljivosti

164. Objasniti princip dizajna softvera Informacioni ekspert.

"Odgovornost za obavljanje posla dodeljivati klasi koja ima informacije neophodne za obavljanje tog posla". Ideja ovog principa je da ponašanje (metodi koji obavljaju neki posao) i znanje (informacije potrebne za obavljanje tog posla) budu locirani na istom mestu. Poštovanje ovog principa dovodi do manje zavisnosti između različitih strukturnih elemenata programa, što je poželjno. Ako ne bi poštovali ovaj princip, klasa koja treba da obavi posao bi morala nekako da komunicira sa klasom koja ima potrebne informacije za taj posao, što dovodi do njihove zavisnosti, koja je nepoželjna.

165. Objasniti princip dizajna softvera Stvaralac.

"Odgovornost za pravljenje objekta klase A dodeliti klasi B ako važi bar jedno od:

- instanca klase B predstavlja kompoziciju instanci klase A;
- instanca klase B referiše instance klase A;
- instanca klase B blisko koristi instance klase A;
- instanca klase B ima informacije za inicijalizaciju instanci klase A i prenosi ih pri pravljenju".

Glavni motiv ovog principa je smanjenje broja zavisnosti, a to se postiže tako što se odgovornost za pravljenje instance neke klase stavlja na mesto gde se ta klasa svakako već upotrebljava i gde već postoji zavisnost od te klase (u suštini primena dekomponovanja prema promenljivosti).

166. Objasniti princip dizajna softvera Visoka kohezija.

"Dodeljivati odgovornosti tako da kohezija ostane visoka". Kohezija je mera međusobne spregnutosti elemenata neke strukturne celine (što, za visok nivo kohezije, ukazuje na to da je ta celina zadužena za jedan posao, što je poželjna osobina), pa zbog toga treba težiti da je održavamo na visokom nivou za svaku strukturnu celinu.

167. Objasniti princip dizajna softvera Niska spregnutost.

"Dodeljivati odgovornost tako da spregnutost ostane niska". Spregnutost je mera međusobne povezanosti elemenata različitih strukturnih celina programa (što, ukoliko je ona visoka, ukazuje na veliku međusobnu zavisnost elemenata, što naravno treba izbeći), pa težimo da je održimo na što nižem nivou.

168. Objasniti princip dizajna softvera Kontroler.

"Odgovornost za primanje ili obrađivanje poruka o sistemskim događajima dodeljivati klasi koja:

- predstavlja ceo sistem, uređaj ili podsistem (tzv. fasadni kontroler);
- predstavlja scenario slučaja upotrebe u kome se pojavljuje sistemski događaj (kontroler sesije ili slučaja upotrebe)".

Pod sistemskim događajem se podrazumevaju sve ulazne ili izlazne operacije koje nisu neposredan proizvod rada programa koji pišemo (npr. akcije korisnika putem korisničkog interfejsa, akcije uređaja, drugih povezanih

programa itd.). Ideja je da se reagovanje na takve akcije grupiše na mestu odakle se upravlja aktivnostima i komunikacijom u konkretnom slučaju upotrebe.

169. Objasniti princip dizajna softvera Polimorfizam.

"Kada se neka vrsta ponašanja menja prema tipovima, onda je potrebno odgovornosti za odgovarajuće ponašanje rasporediti po tim tipovima, primenom polimorfizma". Ovaj princip se odnosi na ulogu polimorfizma, tj. omogućavanje da sa istim interfejsom konkretni objekti različitih tipova ispoljavaju različito ponašanje. Princip se odnosi i na hijerarhijski i na parametarski polimorfizam. Glavni motiv je što se zavisnost ne pravi prema svakom od konkretnih tipova, već samo prema njihovoj zajedničkoj apstrakciji (baznoj klasi u slučaju hijerarhijskog polimorfizma).

170. Objasniti princip dizajna softvera Izmišljotina.

"Tesno povezan i zaokružen skup odgovornosti dodeliti veštački uvedenoj klasi, koja ne predstavlja koncept iz domena problema - koja je izmišljena da bi omogućila visoku koheziju, nisku spregnutost i višestruku upotrebu". U suštini, ovaj princip sugeriše da projektant u nekim slučajevima ne mora da tačno i isključivo sledi strukture iz domena koji se modelira, već može i da "izmišlja" nove koncepte ukoliko će to pomoći u postizanju poželjnih karakteristika programa. Obično se koristi kada praćenje postojećih elemenata i koncepata iz prostora domena dovodi do problematičnih zavisnosti. Rezultat ovog principa je najčešće uvođenje nove klase koja ne odgovara neposredno ni jednom strukturnom elementu domena, ali obuhvata aspekte ponašanja koji su čvrsto međusobno povezani, a slabo povezani sa ostalim konceptima i strukturama. Primer su obrasci Fasada, Adapter, Posetilac i Dekorater.

171. Objasniti princip dizajna softvera Indirekcija.

"Dodeliti odgovornosti objektu koji je posrednik između drugih komponenti ili servisa, tako da oni ne moraju da budu neposredno spregnuti". Ideja je da se potencijalno komplikovane međusobne zavisnosti (Naročito cirkularne i dvosmerne) pojednostave uvođenjem posrednika između problematičnih komponenti, tako da se svaka komunikacija između njih obavlja preko tog posrednika. Uglavnom se teži da implementacija posrednika zavisi od interfejsa komponenti, a da komponente zavise samo od interfejsa posrednika, što dovodi do toga da se izmene u interfejsu nekih komponenti propagiraju samo na posrednika, ali ne i na ostale komponente.

172. Objasniti princip dizajna softvera Izolovane promenljivosti.

"Dodeliti odgovornosti tako da se oblikuje stabilan interfejs oko prepoznatih tačaka predvidive promenljivosti ili nestabilnosti". Ideja ovog principa je da sve nestabilne elemente sistema treba izolovati od okoline apstraktnim interfejsom za koji je veoma mala šansa da će se promeniti (stabilan je). Ovde se primenjuje dekomponovanje prema promenljivosti, a fokus je na tačkama promenljivosti (koje treba izolovati).

173. Navesti principe oblikovanja celina (dizajn softvera).

U principe grupisanja spadaju:

- Princip ekvivalentnosti izdanja i upotrebe
- Princip zajedničke zatvorenosti
- Princip zajedničke upotrebe

U principe razdvajanja spadaju:

- Princip stabilne zavisnosti
- Princip stabilne apstrakcije
- Princip acikličkih zavisnosti

174. Objasniti princip ekvivalentnosti izdanja i upotrebe.

"Granula ponovljive upotrebe je granula objavljivanja". Pod objavljivanjem se misli na zvanično objavljivanje formalne specifikacije funkcionalnosti i interfejsa neke strukturne celine programa, bilo unutar razvojnog tima ili šire. Princip ekvivalentnosti izdanja i upotrebe nalaže da bi trebalo da postoji obostrana tesna veza između ponovljive upotrebe i objavljivanja, tj. treba da važi sledeće:

- Ako bi neka celina trebalo da se koristi na više mesta u programu, onda je neophodno da se u nekom trenutku objavi njena specifikacija (funkcionalnost i interfejs).
- Ako nešto objavimo, onda moramo da računamo da se to koristi već od trenutka objavljivanja.

Princip implicira da objavljena celina, zbog ponovljive upotrebe, mora imati što stabilniji interfejs i što bolju enkapsulaciju.

175. Objasniti princip zajedničke zatvorenosti.

"Delovi celine bi trebalo da budu zajedno i podjednako zatvoreni u odnosu na iste vrste promena. Ako celina mora da se menja, onda se pretpostavlja da moraju da se menjaju i (svi) delovi te celine, ali ne i druge celine". Ovaj princip se često odnosi na slučajeve komunikacione kohezije (delovi celine koriste iste podatke) i u suštini sugeriše da može biti dobro delove grupisati u celinu zato što imaju zajedničke faktore promenljivosti. Primer bi bio skup funkcija (metoda) za formatiranje izveštaja - one mogu biti međusobno relativno nezavisne, ali sve zavise od specifikacije formata izveštaja. Princip zajedničke zatvorenosti kaže da bi bilo dobro grupisati te funkcije na jedno mesto, npr. u jednu klasu (u ovom slučaju, međutim, ne bi ostvarivali funkcionalnu koheziju).

176. Objasniti princip zajedničke upotrebe.

"Klase u paketu se koriste zajedno. Ako se koristi jedna od klasa u paketu, onda se koriste sve". Ovaj princip se prvenstveno odnosi na logičku strukturnu dekompoziciju i pakete - u suštini, kaže da je u isti paket potrebno staviti klase koje se koriste zajedno (npr. klase čvorova apstraktnog sintaksnog stabla). Ovaj princip se takođe može primeniti i na funkcionalnu dekompoziciju, uglavnom na komunikacionu, proceduralnu ili sekvencijalnu koheziju unutar komponente.

177. Objasniti princip stabilne zavisnosti.

"Smer zavisnosti bi trebalo da se poklapa sa smerom porasta stabilnosti". Ovaj princip sugeriše da bi zavisnosti između elemenata programa trebalo da idu od manje stabilnih prema stabilnijim elementima, a ne suprotno (element je stabilan ukoliko relativno mali broj faktora može da dovede do potrebe da ga menjamo). Ovo je i logično, jer ukoliko dođe do potrebe da se neki element menja, postoji verovatnoća da će se promene propagirati i kroz od njega zavisne elemente, pa zato želimo da element od kojeg zavise drugi elementi bude što stabilniji.

178. Objasniti princip stabilne apstrakcije.

"Celina bi trebalo da bude apstraktna onoliko koliko je stabilna". Ovaj princip sugeriše da apstraktni delovi programa ne bi trebalo da zavise od nestabilnih delova programa (a ako baš moraju da zavise od nečega, onda to treba da budu još apstraktniji delovi programa), ali i da nestabilni delovi programa treba da zavise od apstraktnih. Ključan aspekt ovog principa je da povezuje stabilnost i apstraktnost, pa zajedno sa principom stabilne zavisnosti sugeriše da bi smer zavisnosti trebao da se poklapa sa smerom porasta apstraktnosti.

179. Objasniti princip acikličkih zavisnosti.

"Ne dopuštati ciklične zavisnosti paketa". Iako se prvenstveno odnosi na pakete, ovaj princip se može primeniti i na funkcionalne elemente programa (poput komponenti i klasa), mada tada se ciklične funkcionalne zavisnosti ponekad ne mogu skroz eliminisati, pa je potrebno modifikovati program tako da imaju što manji uticaj na stabilnost delova programa. Glavni problem kod ciklične zavisnosti je mogućnost propagacije promena kroz cikl. Ona se obično rešava tako što se promeni smer zavisnosti između neka dva elementa koji učestvuju u njoj (npr. kroz princip inverznih zavisnosti ili princip izmišljotina).

180. Šta je agilni razvoj softvera?

Agilni razvoj softvera je familija razvojnih metodologija koja je nastala krajem 20. veka. Ima mnogo sličnosti sa OO metodologijama, ali ima značajno drugačiji pristup u planiranju u udnosu na klasične OO metodologije. Iako neki principi agilnog razvoja mogu izgledati nesistematično i neuredno, to zapravo uopšte nije tačno.

181. Navesti osnovne pretpostavke Manifesta agilnog razvoja.

- Pojedinci i saradnja su ispred procesa i alata
- Funkcionalan softver je ispred iscrpne dokumentacije
- Saradnja sa klijentom je ispred pregovaranja oko ugovora
- Reagovanje na promene je ispred praćenja plana

Iako su vrednosti sa leve strane važnije, one sa desne ne treba potpuno zapostaviti.

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

Pojedinci (ljudi), tj. članovi razvojnog tima, su najvažniji i presudni činilac uspešnog razvoja. Iz tog razloga, jako je važno da tim bude sastavljen od sposobnih pojedinaca sa dobrim međusobnim odnosima. Iako su procesi i alati veoma bitni, loš razvojni proces može učiniti razvojni tim manje produktivnim, a dobar ih može učiniti još boljim. Preveliko davanje pažnje alatima i stavljanje ljudi u drugi plan može biti još gore od potpunog odsustva alata.

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

Krajnji cilj razvoja softvera je softver, a ne dokumentacija. Dokumentacija projekta je bezvredna ukoliko sam softver ne radi. Međutim, dokumentacija se ne sme zapostaviti - softver bez prateće dokumentacije je propast (kako za korisnike i vlasnike, tako i za one koji će održavati taj softver). Potrebno je odrediti pravu meru u obimu i preciznosti dokumentacije, kao i vremena potrebnog za njeno pisanje. Važno je napomenuti i da je prerano napisana dokumentacija najčešće nestabilna i podložna promenama.

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

U svim agilnim metodologijama teži se da se pri sklapanju posla ugovorom prvenstveno odrede međusobni odnosi, a ne kompletan obim poslova. Naime, najčešće je veoma teško odrediti tačne troškove izrade celog projekta i obim poslova pre početka rada na njemu, pa se teži da se, umesto detaljnog dogovora oko celog projekta pri sklapanju ugovora, detalji određuju nakndano, kako bude tekao razvoj. Ovo se često obavlja u iteracijama (što odgovara tome što većina agilnih metodologija počiva na nekom vidu iterativnog razvoja).

185. Objasniti pretpostavku agilnog razvoja da je reagovanje na promene ispred praćenja plana.

Agilne metodologije prihvataju da su promene neminovne. Klijent može promeniti već iskazane stavove, da se promene okolnosti poslovanja, da se pojave neki novi zadaci ili da se neki prethodno oblikovani zadaci promene. Iz tog razloga u agilnim metodologijama važi da je reagovanje na promene važnije od praćenja plana. Međutim, plan je i dalje jako bitan, ali predlaže se da on bude detaljan samo za kratak period, najčešće za jednu narednu iteraciju, a da na duge staze bude zacrtana samo vizija (tj. glavni cilj razvoja).

186. Navesti bar 8 principa agilnog razvoja softvera.

- Najviši prioritet je zadovoljiti klijenta kroz brzo i neprekidno isporučivanje vrednog softvera.
- Uvek otvoreno prihvatati promene, čak i u kasnim fazama razvoja. Agilni razvoj softvera uvažava promene kao sredstvo postizanja kvaliteta za klijenta.
- Isporučivati softver koji radi, što češće, sa intervalom od par nedelja do par meseci, pri čemu se kraći intervali više cene.
- Poslovni ljudi i razvijaoci moraju svakodnevno da sarađuju na projektu.
- Zasnivati projekat na motivisanim pojedincima. Pružiti im okruženje i potrebnu podršku i imati poverenja u njih da će obaviti posao.
- Najefikasniji način za prenošenje informacija timu i unutar tima je razgovor licem u lice.

- Softver koji radi je osnovno merilo napretka.
- Agilni razvoj softvera promoviše uzdržani razvoj. Sponzori, razvijaoci i korisnici bi trebalo da budu u stanju da neograničeno dugo održavaju ujednačen ritam.
- Neprekidno posvećivanje pažnje tehničkoj doteranosti i dobrom dizajnu podiže agilnost.
- Jednostavnost, kao umetnost maksimizovanja količine posla koji se ne obavlja, je od suštinskog značaja.
- Najbolje arhitekture, zahtevi i projekti potiču iz samoorganizovanih timova.
- U redovnim intervalima tim mora da sagleda svoj rad i mogućnosti unapređivanja svog ponašanja i efikasnosti.

187. Navesti bar 3 metodologije agilnog razvoja softvera.

- Ekstremno programiranje (XP - Extreme Programming)
- Skram (Scrum)
- Agilno modeliranje
- Agilan objedinjen proces (AUP - Agile Unified Process)
- Otvoren objedinjen proces (OpenUP - Open Unified Process)

188. Šta je ekstremno programiranje?

Ekstremno programiranje je jedna od najpoznatijih, mada više ne i jedna od najzastupljenijih, agilnih metodologija. Sačinjena je od više jednostavnih i međusobno nezavisnih metoda kojima se preciznije opisuje način ostvarivanja principa agilnog razvoja softvera. Iako više nije među najzastupljenijim metodologijama, ostavila je veliki uticaj na druge metodologije sa svojim metodama.

189. Navesti bar 8 metoda ekstremnog programiranja.

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

190. Objasniti metod ekstremnog programiranja "Klijent je član tima".

Ekstremno programiranje zastupa stav da je neophodno da klijenti i razvijaoci rade zajedno da bi se postigla kvalitetna saradnja. Najefikasniji način da se to ostvari je da predstavnik klijenta bude stalno prisutan, praktično kao ravnopravan član razvojnog tima. Ovo omogućava ubrzano dobijanje odgovora na pitanja, neformalno saopštavanje potreba za izmenama, podizanje poverenja klijenta u razvojni tim i intenzivno uključenje klijenta u definisanje testova prihvatljivosti. Međutim, jedan od potencijalnih problema kod ovakvog pristupa je mogućnost klijenta da saopštava veliki broj želja za izmenama, što može da uspori rad.

191. Objasniti metod ekstremnog programiranja "Korisničke celine (User Stories)".

Korisničke celine predstavljaju okvirnu, relativno površnu (pošto ih piše klijent, najčešće nisu preterano tehnički precizne), specifikaciju nekog zahteva. Njihova osnovna namena je da pomognu pri pravljenju okvirnog plana i praćenja toka projekta, pa se kao takve koriste kao osnovni element za pravljenje planova iteracija. Najčešće im se dodeljuju prioriteti i grube procene potrebnih resursa za implementaciju. Detalji se razmatraju tek kad dođe vreme na implementaciju.

192. Objasniti metod ekstremnog programiranja "Kratki ciklusi".

U ekstremnom programiranju praktikuje se redovna isporuka softvera (obično na svake 2 nedelje). Da bi se ovo postiglo, razvojni ciklus razlikuje pojam iteracije i izdanja. Iteracija je kratak ciklus tokom koga se radi na izabranom skupu korisničkih celina, sa ciljem da se te celine i završe da bi klijent mogao da testira nove verzije softvera. Izdanje je duži ciklus razvoja i obuhvata nekoliko iteracija, npr. od 4 do 6. Osnovni cilj im je zaokruživanje većih celina i dovršavanje verzije softvera koja bi trebalo da ide u produkciju. Kratke iteracije pomažu u stvaranju većeg poverenja između razvojnog tima i klijenta, kao i lakše reagovanje na potencijalne izmene.

193. Objasniti metod ekstremnog programiranja "Testovi prihvatljivosti".

Testovi prihvatljivosti predstavljaju primarni oblik dokumentovanja merila ispravnosti i kvaliteta implementacije korisničke celine. Testove prihvatljivosti određuje, a ako je moguće i tehnički definiše, sam klijent. Pišu se neposredno pre implementacije (pri planiranju odgovarajuće iteracije) ili tokom implementacije korisničke celine. Svakako, moraju biti spremni pre kraja iteracije. Poželjno je pisati testove na nekom skript jeziku radi automatizacije testiranja - potrebno je pokrenuti sve testove prilikom svake naredne izgradnje sistema.

194. Objasniti metod ekstremnog programiranja "Programiranje u paru".

Ideja je da se sav produkcioni kod piše u parovima od po dva programera. Oba programera rade na istoj radnoj stanici, ali tako što jedan piše kod, a drugi u hodu kritički posmatra napisan kod, proverava da li je sve u redu i daje predloge. Uloge ta dva člana treba menjati često, a takođe je potrebno i menjati sam sastav timova povremeno (npr. jednom dnevno). Razlog za ovo je taj što je poželjno da tokom jedne iteracije, svaki član tima radi bar jednom sa svim ostalim članovima tima, jer na taj način svako će donekle biti upoznat sa ostalim delovima sistema za koje nisu specijalizovani.

195. Objasniti metod ekstremnog programiranja "Razvoj vođen testovima".

Ideja je da se produkcioni kod piše sa ciljem da zadovolji testove jedinica koda. Ovo podrazumeva prvo pisanje testova, a zatim koda koji će omogućiti da se svi testovi prođu. Preporučuje se da se pisanje testova i koda brzo smenjuje, tj. da se ne piše odjednom veliki broj testova, pa zatim mnogo koda, već je bolje da se piše nekoliko testova i onda kod koji će ih zadovoljiti, i tako u krug. Postojanje testova omogućava laku proveru ispravnosti, naročito nakon izmena u kodu (npr. zbog refaktorisanja).

196. Objasniti metod ekstremnog programiranja "Kolektivno vlasništvo".

Pod kolektivnim vlasništvom podrazumeva se da je svaki deo napisanog koda odgovornost svih članova tima. Svi članovi tima (smenjujući se u parovima), rade na svim delovima i nivoima softvera. Svaki član tima (i svaki par) ima pravo da proveri i unapredi bilo koji deo koda, a ne samo onaj koji su oni lično razvijali.

197. Objasniti metod ekstremnog programiranja "Neprekidna integracija".

Kolektivno vlasništvo nad kodom (svako je odgovoran za svaki deo koda) dovodi do toga da više parova radi nad delovima istog koda, pa je potrebno koristiti sisteme za kontrolu verzija koda (Git, SVN, ...). Neprekidna integracija podrazumeva da se postavljanje (integracija) izmenjenog koda na server za kontrolu verzije koda vrši ne samo kad se završi neka velika, složena celina, već dosta češće - npr. nakon svake iteracije zadovoljavanja napisanih testova. Pre svake integracije, potrebno je proveriti ispravnost koda, tj. da li se kod prevodi i da li prolaze svi testovi. Da bi se ovo olakšalo, koriste se alati za neprekidnu integraciju koji ovaj proces automatizuju.

198. Objasniti metod ekstremnog programiranja "Uzdržan ritam".

Agilni razvoj softvera promoviše ustaljen ritam rada i izbegavanje prekovremenog rada (do kojeg često dolazi usled prekoračenja rokova). Ekstremno programiranje čak i potpuno zabranjuje prekovremeni rad. Jedini izuzetak je poslednja nedelja izdanja, kada se očekuje da tim bude potpuno posvećen kvalitetu i popravljanju uočenih nedostataka. Uzdržavanje od prekomernog rada rasterećuje članove tima i pomaže u održavanju kvalitetnih odnosa unutar tima, što za uzvrat dovodi do manjeg broja grešaka i veće iskorišćenosti radnog vremena, pa time i veće produktivnosti.

199. Objasniti metod ekstremnog programiranja "Otvoren radni prostor".

Ova praksa propisuje da bi razvojni tim trebalo da radi u jednoj, dovoljno velikoj prostoriji. Ovo bi trebalo da olakša komunikaciju između članova tima, kao i lakše programiranje u paru. Ovaj način rada ima veliki broj protivnika (npr. zbog smanjene privatnosti, velike gužve itd.), pa se predlažu kompromisi u vidu manjih prostorija sa tri do pet radnih stolova.

200. Objasniti metod ekstremnog programiranja "Igra planiranja".

U ekstremnom programiranju, odgovornost u planiranju je podeljena. Naime, klijent odlučuje o značajnosti karakteristika softvera, šta je potrebno, šta nije, kao i o prioritetima karakteristika (ovo se evidentira u obliku korisničkih celina). Razvojni tim odlučuje koliko će to da košta. Ovo se uglavnom svodi na to da razvojni tim određuje obim (vremena i novca) prilikom planiranja iteracije, a klijent u ta ograničenja uklapa korisničke celine. Važno je napomenuti da kada iteracija započne, klijent više ne menja karakteristike i detalje korisničkih celina za tu iteraciju. Međutim, za razliku od iteracija, promene su dozvoljene kada je u pitanju izdanje.

201. Objasniti metod ekstremnog programiranja "Jednostavan dizajn".

Ova praksa nalaže da dizajn softvera treba da bude što jednostavniji i što izražajniji. Ovo se ostvaruje kroz tri pravila:

- Razmotriti prvo najjednostavnije rešenje koje bi moglo da uradi posao - npr. ako nešto može da se reši bez paralelizacije, onda to treba tako i uraditi.
- Neće biti potrebno - za sve potencijalne elemente softvera koji će biti potrebni možda ili za neko vreme pretpostaviti da neće biti potrebni. Ovo se uglavnom svodi na to da se tekuća iteracija posmatra kao jedini cilj razvoja, bez obaziranja na naredne iteracije. To, međutim, može dovesti do potrebe za izmenama u kodu u narednim iteracijama, što može pokvariti dizajn, pa je potrebno primenjivati refaktorisanje.
- Jedanput i samo jedanput - ponavljanje u kodu nije dozvoljeno. Ukoliko nastane ponavljanje, treba se ukloniti refaktorisanjem (npr. izdvajanjem koda u novi metod ili klasu).

Važno je napomenuti da jednostavan dizajn ne znači loš i nepažljiv dizajn.

202. Objasniti metod ekstremnog programiranja "Refaktorisanje".

Refaktorisanje je osnovno sredstvo za borbu protiv opadanja kvaliteta koda. Ono podrazumeva preduzimanje niza malih transformacija koda kojima se unapređuje struktura koda bez vidljive promene ponašanja sistema. Posle svake primenjene transformacije potrebno je pokrenuti testove kojima se proverava da nije slučajno izmenjeno ponašanje. Ono bi trebalo da se primenjuje redovno, tj. kad god se uoči da je neka izmena narušila dizajn koda.

203. Objasniti metod ekstremnog programiranja "Metafora".

Metafora odgovara konceptu vizije u drugim metodologijama, tj. predstavlja veliku, apstraktnu, sliku softvera koji se razvija. Svi zahtevi i korisničke celine bi trebalo da potiču iz metafore. Ona se često formalizuje u vidu rečnika pojmova koji identifikuje najvažnije koncepte softvera i probleme koje bi on trebalo da reši. Neki zastupnici ekstremnog programiranja se protive ovoj praksi jer je suštinski suprotna od ideje da razvojni tim gleda najviše jednu iteraciju daleko, ali je ona ipak značajna za održavanje fokusa tokom razvoja i sprečavanje skretanja sa puta.

204. Šta je "Razvoj vođen testovima"?

Razvoj vođen testovima je praksa preuzeta iz ekstremnog programiranja. Ideja je da se pre bilo kakvog pisanja koda prvo pišu testovi koje kod ima za cilj da zadovolji. Postojanje testova omogućava lakšu proveru ispravnosti, naročito nakon refaktorisanja.

205. Navesti i objasniti vrste testova softvera.

- Testovi jedinica koda (Unit Test) - testiranje pojedinačnih delova koda, npr. metoda neke klase
- Integralni testovi (Integration Test) - testiranje povezanih celina koda, tj. kako se ponašaju delovi koda kada ih primenjujemo zajedno
- Testovi sistema (System Test) - testiranje sistema kao celine
- Testovi prihvatljivosti (Acceptance Test) - testiranje sistema iz ugla korisnika (često pomoću skriptova koji simuliraju ili emuliraju korisnički interfejs)

206. Navesti i objasniti ukratko osnovne principe razvoja vođenog testovima.

Osnovni principi su:

- Testovi prethode kodu - pre pisanja bilo kakvog koda, prvo je potrebno napisati odgovarajuće testove.
- Sistematičnost - svaka karakteristika softvera, svi granični slučajevi i svaka linija koda moraju biti pokriveni testovima. Naknadno uočavanje bagova je često rezultat nedovoljno sistematičnih testova.

207. Objasniti princip razvoja vođenog testovima "Testovi prethode kodu" i način njegove primene.

Pre pisanja bilo kakvog koda, prvo je potrebno napisati odgovarajuće testove. Ni jedna funkcija programa se ne razvija sve dok ne postoji test koji ne uspeva zbog njenog odsustva, a još preciznije, ni jedna linija koda se ne dodaje sve dok ne postoji test koji zbog nje ne uspeva. Svaka iteracija u razvoju obično počinje pisanjem testova, nakon čega sledi pisanje kostura koda koji omogućava da se testovi prevedu (oni i dalje neće prolaziti), a nakon toga sledi pisanje koda koji omogućava da testovi prođu. Pisanje testova i koda se najčešće jako brzo smenjuje, tj. nije dobro napisati ogroman broj testova i zatim sve implementirati odjednom, već je bolje napisati nekoliko testova, napisati kod tako da oni prolaze, pa onda opet dodati još testova itd.

208. Objasniti princip razvoja vođenog testovima "Sistematičnost".

Svaka karakteristika softvera, svi granični slučajevi i svaka linija koda moraju biti pokriveni testovima. Naknadno uočavanje bagova je često rezultat nedovoljno sistematičnih testova, jer da su oni bili dovoljno sistematični, pokazali bi problem.

209. Navesti osnovne uloge testova.

- Testovi su vid verifikacije
- Testovi pomažu sa refaktorisanjem
- Testovi omogućavaju programeru da posmatra kod iz drugog ugla
- Testovi su vid dokumentacije

210. Objasniti ulogu testova kao vida verifikacije softvera.

Testovi predstavljaju vid verifikacije - kolekcija testova omogućava programeru da proverava da li jedinica koda radi ispravno ili ne. Oni pomažu u ranom prepoznavanju grešaka (i na nivou koda, a i na konceptualnom nivou), a takođe i podstiču bolji dizajn projekta. Važno je naglasiti da testovi nisu formalni metod verifikacije i ne mogu stopostotno garantovati ispravnost softvera.

211. Objasniti ulogu testova u okviru refaktorisanja.

Naknadne izmene u kodu mogu, iako to nije bio cilj, da promene ponašanje programa i uvedu greške. Testovi pomažu u proveri da li je ponašanje sistema promenjeno prilikom refaktorisanja.

212. Objasniti ulogu testova u kontekstu ugla posmatranja koda.

Pravljenje testova stavlja programera na mesto korisnika koda koji se piše (na koje sve načine korisnik može da koristi program). Ovo pomaže u dobijanju lako upotrebljivog i proverljivog koda.

213. Objasniti ulogu testova kao vida dokumentacije.

Dobro napisani testovi predstavljaju oblik specifikacije zahteva, opisuju uslove funkcionisanja koda, način upotrebe interfejsa jedinice koda i karakteristične primere. Kao takvi, oni se mogu koristiti kao vid dokumentacije, a još jedna odlična njihova osobina je ta da je takav vid dokumentacije uvek ažuran.

214. Šta može biti jedinica koda koja se testira?

Jedinica koda koja se testira može biti operacija, funkcija, metod, struktura podataka, klasa, više manjih integrisanih jedinica koda (softverski paket, podsistem), spoljašnji podsistem (testiranje da li interfejs odgovara specifikaciji).

215. Šta može biti predmet testiranja jedinice koda?

Predmeti testiranja jedinice koda mogu biti funkcionalni zahtevi, zavisnost postuslova od preduslova (da li jedinica koda radi ispravno za različite očekivane kategorije ulaza), robusnost (ponašanje u slučaju neispravnih ulaznih podataka), integracija (da li se manje jedinice koda ponašaju ispravno kada se koriste zajedno), interfejs spoljašnjeg podsistema (da li interfejs odgovara specifikaciji).

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

- Catch2
- CppUnit
- CppUnitLite
- Boost.Test
- Google Test
- Unit++
- CxxTest
- CPUnit

217. Opisati ukratko osnovne mogućnosti biblioteke CppUnit.

Biblioteka CppUnit za pisanje testova jedinica omogućava relativno jednostavno pisanje testova uz pomoć bogatog skupa makroa za proveravanje uslova (pretpostavke, tj. assert-ovi), kao i grupisanje testova u grupe (Test Suite). Podržava različite načine izveštavanja o uspehu/neuspehu testova (npr. kroz proizvoljno definisanu poruku). Takođe podržava i izuzetke.

218. Koji su osnovni elementi koje programer pravi pri pravljenju testova uz primenu biblioteke CppUnit? Kako?

U okviru projekta se pravi novi cilj, tj. program za izvršavanje testova, kojem se zatim dodaje prevedena biblioteka CppUnit. Potrebno je napisati glavnu funkciju (main) tako da pokreće testove. Za svaku jedinicu produkcionog koda se piše po jedna ili više test jedinica koda (test slučajeva) u vidu klasa sa testovima. Ove klase nasleđuju klasu `CppUnit::TestCase` i njihov glavni metod je `void runTest()` u okviru koga se koriste makroi za provere (assert-ovi) koji bacaju izuzetke u slučaju neuspešne provere.

219. Šta je Test Suite?

Test Suite (svita testova) je u CppUnit-u naziv za grupu testova unutar koje se testovi izvršavaju zajedno, što omogućava lakše rukovanje sa većim skupom testova. Za označavanje početka i kraja svite unutar jedne klase se koriste makroi `CPPUNIT_TEST_SUITE` i `CPPUNIT_TEST_SUITE_END`. Ovi makroi, između ostalog, implementiraju i

statički metod `suite()` za pravljenje objekta svite testova. Izvršavaču se onda ne dodaje sam objekat, već statički napravljena svita - `runner.addTest(NekiTest::suite());` .

220. Navesti osnovne vrste pretpostavki koje podržava biblioteka CppUnit.

- `CPPUNIT_ASSERT(uslov)` - pretpostavka da je dati uslov zadovoljen
- `CPPUNIT_ASSERT_MESSAGE(poruka, uslov)` - pretpostavka da je dati uslov zadovoljen, sa proizvoljnom porukom
- `CPPUNIT_ASSERT_EQUAL(ocekivana, stvarna)` - pretpostavka da su dve vrednosti jednake
- `CPPUNIT_ASSERT_EQUAL_MESSAGE(poruka, ocekivana, stvarna)` - pretpostavka da su dve vrednosti jednake, sa proizvoljnom porukom (`_MESSAGE` sufiks se može dodati na svaki osnovni tip pretpostavke)
- `CPPUNIT_ASSERT_DOUBLES_EQUAL(ocekivana, stvarna, delta)` - pretpostavka da su dve vrednosti jednake, sa dopuštenom greškom delta
- `CPPUNIT_ASSERT_THROW(izraz, TipIzuzetka)` - pretpostavka da izračunavanje izraza baca izuzetak tipa TipIzuzetka
- `CPPUNIT_ASSERT_NO_THROW(izraz)` - pretpostavka da izračunavanje izraza ne baca izuzetak
- `CPPUNIT_ASSERT_ASSERTION_FAIL(pretpostavka)` - pretpostavka da pretpostavka ne uspeva
- `CPPUNIT_ASSERT_ASSERTION_PASS(pretpostavka)` - pretpostavka da pretpostavka uspeva

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

Catch2 je biblioteka za pisanje testova jedinica koda u programskom jeziku C++. Distribuira se u vidu jednog zaglavlja koje je potrebno uključiti u modulima sa testovima. Ima poprilično jednostavnu kolekciju makroa što pojednostavljuje upotrebu. Testovima je moguće davati imena, kao i dodatne oznake (tagove), što omogućava njihovo pokretanje, kako pojedinačno po imenu, tako i grupe testova koje imaju dati tag. Takođe, nije potrebno pisati posebne klase za testiranje, već se za sve koriste prethodno opisani testovi.

```
TEST_CASE("Dodavanje elemenata u vektor", "[vektor]")
{
    SECTION("Dodavanje u prazan vektor")
    {
        Vektor v;
        v.dodaj(1);
        REQUIRE(v.velicina() == 1);
    }
    SECTION("Dodavanje u neprazan vektor")
    {
        Vektor v = {1, 2, 3};
        v.dodaj(4);
        v.dodaj(5);
        CHECK(v.velicina() == 5);
    }
}
```

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

Prvo je potrebno definisati glavni program za testiranje. Moguće je definisati sopstvenu main funkciju, ali je isto tako moguće i koristiti main funkciju koju definiše sam Catch2 dodavanjem posebnog makroa:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
```

Osnovni elementi su:

- `TEST_CASE` - slučaj testa, obično odgovara jednoj jedinici koda koju testiramo. Potrebno je dati mu ime, a opciono i oznake (tagove).
- `SECTION` - sekcija unutar nekog slučaja testa, koriste se za definisanje više mogućih putanja unutar jednog testa (npr. za različite moguće konkretne slučajeve).
- `CHECK/REQUIRE` (i ostale pretpostavke) - makroi za pisanje pretpostavki, razlika između ova dva je što sa CHECK, provere se nastavljaju u slučaju neuspeha, dok se sa REQUIRE testiranje prekida u trenutku neuspeha.

223. Šta je Test Case? Šta je Test Case Section? (Catch)

- `TEST_CASE` - slučaj testa, obično odgovara jednoj jedinici koda koju testiramo. Potrebno je dati mu ime, a opciono i oznake (tagove).
- `SECTION` - sekcija unutar nekog slučaja testa, koriste se za definisanje više mogućih putanja unutar jednog testa (npr. za različite moguće konkretne slučajeve).

224. Navesti osnovne vrste pretpostavki koje podržava biblioteka Catch.

- `CHECK/REQUIRE` - provera da li je uslov tačan
- `CHECK_FALSE/REQUIRE_FALSE` - provera da li je uslov netačan
- `CHECK_NOTHROW/REQUIRE_NOTHROW` - provera da li neki izuzetak nije izbačen
- `CHECK_THROWS/REQUIRE_THROWS` - provera da li je neki izuzetak izbačen
- `CHECK_THROWS_AS/REQUIRE_THROWS_AS` - provera da li je neki specifični izuzetak izbačen

225. Šta su testovi prihvatljivosti?

Testovi prihvatljivosti predstavljaju primarni oblik dokumentovanja merila ispravnosti i kvaliteta implementacije korisničke celine. Testove prihvatljivosti određuje, a ako je moguće i tehnički definiše, sam klijent. Pišu se neposredno pre implementacije (pri planiranju odgovarajuće iteracije) ili tokom implementacije korisničke celine. Svakako, moraju biti spremni pre kraja iteracije. Poželjno je pisati testove na nekom skript jeziku radi automatizacije testiranja - potrebno je pokrenuti sve testove prilikom svake naredne izgradnje sistema.

226. Po čemu se testovi prihvatljivosti razlikuju od testova jedinica koda?

Testovi prihvatljivosti su konceptualno drugačiji od testova jedinica koda. Naime, oni se odnose na testiranje ponašanja sistema kao celine i obuhvataju sve aspekte njegove upotrebe (definisanje podataka, izvršavanje operacija, proveravanje rezultata izvršenih operacija). Takođe, ne implementiraju se istim alatima (a uobičajeno se definišu nekim specifičnim skript jezikom, npr. XML).

227. Ko od učesnika u razvoju softvera piše testove jedinice koda? A testove prihvatljivosti?

Testove jedinice koda pišu sami programeri iz razvojnog tima, a testove prihvatljivosti, ukoliko je to moguće, sam klijent (tj. tehnički osposobljen za to predstavnik klijenta).

228. Navesti osnovne vrste polimorfizma.

- Hijerarhijski polimorfizam
- Parametarski polimorfizam
- Implicitni polimorfizam
- Ad-hok polimorfizam

229. Šta je hijerarhijski polimorfizam?

Hijerarhijski polimorfizam je osnovni tip polimorfizma u OO jezicima. On koristi hijerarhijske odnose nadtip (generalizacija) i podtip (specijalizacija) među klasama da bi se ostvario polimorfizam.

230. Šta je parametarski polimorfizam?

Parametarski polimorfizam je tip polimorfizma koji se ostvaruje upotrebom tipskih promenljivih, tj. simboličkih tipova, koji se koriste za simboličko označavanje tipova vrednosti, promenljivih i izraza. U fazi prevođenja, simbolički tipovi vrednosti se zamenjuju konkretnim tipovima (koji su zaključeni implicitno ili eksplicitno naglašeni u kodu) i program se prevodi u nepolimorfnom obliku.

231. Šta je implicitni polimorfizam?

Implicitni polimorfizam podrazumeva da se u kodu uopšte ne navode tipovi vrednosti, već će prevodilac sam analizirati svaki konkretan segment programskog koda i zaključiti za koje sve tipove on može da se prevede. On suštinski predstavlja uopštenje parametarskog polimorfizma.

232. Šta je ad-hok polimorfizam?

Ad-hok polimorfizam podrazumeva upotrebu (u jezicima u kojima je onda podržana) osobine višeznačnosti imena (overloading) funkcija i operatora da bi se postigao efekat polimorfizma. Naime, u ovom slučaju bi se pisalo više, najverovatnije skoro pa istih, funkcija sa istim imenom, ali koje imaju različite tipove argumenata (npr. max(int a, int b), max(double a, double b)).

233. Šta su šabloni funkcija?

Šablon funkcije je funkcija koja upotrebljava parametarske (simboličke) tipove u svojoj deklaraciji (tip povratne vrednosti i argumenata) i/ili implementaciji (moguća je upotreba parametarskih tipova i u samom telu funkcije). Moguća je i apstrakcija konstanti na sličan način kao i sa tipovima. Ovo je jedan od osnovnih konstrukata (pored šablona klasa) koji omogućava parametarski polimorfizam u jeziku C++.

234. Šta su šabloni klasa?

Šabloni klasa su definicije klasa koje koriste apstrakciju nekih tipova (u vidu simboličkih, tj. parametarskih tipova) ili konstanti. One su jedan od osnovnih konstrukata (pored šablona funkcija) koji omogućavaju parametarski polimorfizam u jeziku C++.

235. Šta su funkcionalni objekti - funkcionali?

Funkcionali su klase koje imaju definisan operator izvršavanja (u programskom jeziku C++, to je operator "()"). Objekti ovakvih klasa se mogu upotrebljavati kao funkcije.

236. Kakav je odnos refaktorisanja i pisanja programskog koda?

Pisanje novog koda ne menja postojeći kod, već samo dodaje novo ponašanje, a refaktorisanje ne menja vidljivo ponašanje, već samo strukturu. Programiranje je naizmenično pisanje novog koda i refaktorisanje.

237. Šta su osnovni motivi za refaktorisanje koda?

Osnovni motiv je taj što refaktorisanje podiže nivo dizajna softvera, a sve ostalo je posledica toga - softver će biti lakše razumljiv, greške (ukoliko one postoje) će se lakše pronalaziti i brzina pisanja koda će biti veća.

238. Kada se pristupa refaktorisanju koda?

Refaktorisanje treba raditi redovno, a neki od slučajeva su kada se dodaju nove funkcije, kada je potrebno pronaći neki bag, kada se proverava ispravnost i kvalitet koda itd. U suštini, svako dodavanje novog koda je ujedno i potencijalna prilika za refaktorisanje.

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

Pošto je loš dizajn softvera često subjektivna kategorija, izdvajaju se kandidati, tj. slučajevi u kojima je potrebno razmotriti da li refaktorisanjem može da se unapredi kvalitet dizajna koda. Primer takvih kandidata je ponavljanje

koda, dugačak metod, velika klasa itd. Ovo su takozvana "zaudaranja" koda.

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

- Ponavljanje koda
- Dugačak metod
- Velika klasa
- Dugačka lista argumenata
- Divergentne promene
- Distribuirana apstrakcija
- Velika zavisnost od drugih klasa
- Grupisanje podataka
- Poplava primitivnih podataka
- Naredba switch
- Paralelne hijerarhije nasleđivanja
- Lenje klase

241. Zašto je dobro eliminisati ponavljanja iz koda? (refaktorisanje)

Ponavljanje koda otežava razumevanje i održavanje i smanjuje preglednost koda. Takođe, ukoliko je potrebno promeniti kod koji se često ponavlja (npr. usled primećene greške), potrebno je promeniti kod na svakom mestu gde se on ponavlja, što je poprilično nepraktično. Ovaj problem se uglavnom rešava izdvajanjem ponavljanog koda u funkciju/metod, pravljenjem nove klase i slično.

242. Zašto dugački metodi mogu predstavljati problem? (refaktorisanje)

Dugački metodi otežavaju razumevanje, debagovanje i menjanje koda. Oni su često uzrok loše podeljenih odgovornosti, tj. ako jedan metod radi više stvari istovremeno. Obično se rešava izdvajanjem dodatnih metoda.

243. Zašto velika klasa može da predstavlja problem? (refaktorisanje)

Iako se često dešava da su neke klase velike sa razlogom, postoje i slučajevi kada to ne bi trebalo da bude tako. Ukoliko se pravi mnogo objekata jedne klase, a oni imaju različite namene, to ukazuje na to da klasa ima više od jedne namene, pa ju je potrebno podeliti na više klasa.

244. Šta su divergentne promene? Zašto su problematične? (refaktorisanje)

Divergentne promene se odnose na situaciju kada je potrebno menjati jednu klasu na više načina iz više razloga. Primer bi bila potreba da se menja nekoliko različitih metoda usled dodavanja novog tipa artikla koji bi sistem trebalo da prepoznaje. Ponekad je rešenje za ovo izdvajanje klasa.

245. Šta je distribuirana apstrakcija? Zašto je problematična? (refaktorisanje)

Distribuirana apstrakcija se odnosi na situaciju kada je usled neke promene potrebno napraviti veliki broj sitnih izmena u većem broju klasa. Ovo često ukazuje na to da je odgovarajuća apstrakcija distribuirana u više klasa, što nije dobro (jedna klasa bi trebala da bude zadužena za jednu stvar, i ta stvar bi trebalo da bude potpuno sadržana u toj klasi). Ovo se rešava premeštanjem metoda, podataka i umetanjem klase.

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

Velika zavisnost neke klase ili metoda od drugih klasa označava visoku spregnutost između njih i nastaje ukoliko klasa/metod koristi veliki broj metoda druge klase. Ovo često znači da odgovornosti nisu dobro podeljene između klasa i dovodi do toga da zavisni element bude nestabilan (ukoliko nastanu promene u klasi od koje zavisi, onda

će se potreba za promenama verovatno propagirati i kroz njega). Ovaj problem se često rešava premeštanjem i izdvajanjem metoda.

247. Zašto naredba switch može da predstavlja problem? (refaktorisanje)

Problem nastaje ukoliko često ponavljamo neki skup provera (bilo u vidu switch naredbe, bilo u vidu pravljenja "merdevina" if-else upitima). Ukoliko je potrebno dodati još neki uslov, onda je potrebno pronaći i ažurirati sve takve naredbe sa tim dodatnim uslovom. Ovo se može rešiti upotrebom polimorfizma, izdvajanjem uslova u poseban metod i slično.

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

Spekulativno uopštavanje je praksa preteranog apstrahovanja zbog mogućnosti da će u nekom dalekom trenutku taj nivo opštosti biti potreban zbog dodatnih funkcionalnosti. Ovo može otežati preglednost i održavanje koda, pa je, ukoliko se ta opštost nigde ne koristi, poželjno ukloniti je. Ovo se u nekim slučajevima može rešiti sažimanjem klasne hijerarhije.

249. Zašto privremene promenljive mogu da predstavljaju problem? (refaktorisanje)

Ukoliko klasa ima polja (može se uopštiti i na privremene promenljive) koja se koriste samo u nekim posebnim slučajevima, a uglavnom ne, onda to smanjuje razumljivost koda (dolazimo u situaciju da se pitamo zašto to polje uopšte postoji, jer ne vidimo mesto na kojem se ono koristi).

250. Zašto lanci poruka mogu da predstavljaju problem? (refaktorisanje)

Lanci poruka označavaju veliki broj posrednika u obavljanju nekog posla, što vezuje objekte za strukturu tih međusobnih veza. Ako bi se te veze nekako promenile, onda bi se promene propagirale i do klasa koje komuniciraju na taj način. Ovo se često može rešiti sakrivanjem delegata.

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

Ukoliko neka klasa predstavlja samo posrednika, tj. klasu koja samo delegira zahteve prema drugim klasama bez ikakvog dodatnog ponašanja, onda treba razmotriti njeno uklanjanje. Ovaj problem često nastaje zbog preterivanja u enkapsulaciji i dovodi do nekompaktnog i nerazumljivog koda (komunikaciju moramo pratiti preko posrednika, a lakše bi bilo da je ona direktna).

252. Šta je nepoželjna bliskost? Zašto je problematična? (refaktorisanje)

Nepoželjna bliskost se odnosi na to kada neka klasa suviše često pristupa privatnim delovima neke druge klase. Ovo je narušavanje enkapsulacije i može dovesto do visoke spregnutosti i zavisnosti te dve klase.

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

Komentari, najčešće, ne mogu da odmognu, ali ipak je najbolje pisati kod na takav način da komentari ne budu neophodni. Ukoliko je nešto preko potrebno komentarisati, onda takav kod nije dovoljno razumljiv sam po sebi, pa je, ukoliko je to moguće, poželjno refaktorisati ga.

254. Šta bi trebalo da sadrži opis svakog od refaktorisanja u katalogu?

- Ime
- Sažet opis slučaja u kome se primenjuje
- Motivaciju
- Opis tehnike izvođenja
- Primer

255. Navesti bar 5 grupa tehnika refaktorisanja.

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

256. Navesti bar 5 tehnika refaktorisanja.

- Uklanjanje mrtvog koda
- Uklanjanje posrednika
- Izdvajanje klase
- Izdvajanje metoda
- Premeštanje metoda
- Premeštanje podataka
- Sažimanje hijerarhije

257. U kojim slučajevima refaktorisanje može biti značajno otežano?

Refaktorisanje može biti otežano u pristustvu baze podataka, u slučaju da je u sklopu refaktorisanja potrebno izvršiti promene nad javnim (spoljnim) interfejsom neke klase ili ako je dizajn softvera veoma složen.

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

Ovakvo refaktorisanje bi možda zahtevalo promenu strukture baze podataka što ima širok uticaj (možda npr. neko drugi koristi istu bazu). Migracija podataka i izmena njihove strukture može biti jako neugodna, pa važi pravilo da ne treba praviti bazu podataka prevremeno.

259. Zašto može biti otežano refaktorisanje spoljnog interfejsa neke klase?

U trenutku kada je spoljni interfejs neke klase bio objavljen, mogli smo pretpostaviti da on od tada ima neke korisnike. Velike izmene u interfejsu bi izazvale nekompatibilnosti, pa se u ovom slučaju predlaže privremeno čuvanje i starog interfejsa (što dodatno prlja kod) dok se ne izvrši potpuna migracija na novi. Naravno, najbolje je sve ovo potpuno izbeći time što ne objavljujemo javni interfejs dok on ne sazri.

260. U kojim slučajevima refaktorisanje može da ne predstavlja dobro rešenje?

Refaktorisanje nikako ne treba primenjivati ukoliko je postojeći kod neispravan (prvo je potrebno otkriti bag, srediti ga, pa tek onda razmotriti refaktorisanje). Ukoliko su problemi višestruki i međusobno zavisni, takođe ne treba primenjivati refaktorisanje, kao ni kada jednostavno nije moguće vršiti manje korake promena bez remećenja postojećeg ponašanja koda. Takođe, refaktorisanje ne treba vršiti blizu krajnjih rokova.

261. Kakav je odnos refaktorisanja i performansi softvera?

Refaktorisanje veoma često može smanjiti efikasnost koda (npr. izdvajanjem koda u zasebnu klasu dobijamo dodatnu indirekciju). Međutim, pozitivni uticaj na dizajn koji je dobijen refaktorisanjem je daleko veći nego negativni uticaj na performanse, tako da je dobar dizajn koda najčešće bitniji od najoptimalnijih mogućih performansi. Održavanje performansi koda se može vršiti sa tzv. optimizacijom unazad, gde se teži da se prvo dostigne funkcionalnost komponenti, a zatim se one dodatno optimizuju po potrebi, ili sa optimizacijom unapred, gde se tokom čitavog razvoja vodi računa o optimalnom rešenju. Najčešće se preporučuje optimizacija unazad.

262. Šta su bagovi?

Bagovi su sve ono što stvara probleme u funkcionisanju softvera kao završnog proizvoda.

263. Navesti jednu klasifikaciju bagova i objasniti je.

Padovi softvera (programa) i oštećenja podataka - pod padom softvera se podrazumevaju slučajevi neplaniranog prekida rada softvera u vidu njegovog iznenadnog zatvaranja, zamrzavanja ili slično. Ovakvi prekidi mogu prouzrokovati i padove nekih drugih programa, pa čak i operativnog sistema. Oštećenja podataka su možda čak i nezgodnija. Naime, ona mogu ostati prikrivena dugo vremena, što potencijalno ima velike posledice (kako po ispravnost podataka, tako i po dalji rad sistema). Ostale klasifikacije su nekonzistentnosti u korisničkom interfejsu, neispunjena očekivanja i slabe performanse.

264. Šta su nekonzistentnosti u korisničkom interfejsu i kakve uzroke i posledice imaju?

Nekonzistentnosti u korisničkom interfejsu su neusklađenosti između onoga što korisnik želi da uradi i što očekuje, načina na koji on svoje zahteve izdaje softveru, načina na koji softver te zahteve izvršava i načina na koji se rezultati prikazuju korisniku. Uzrok ovoga je često loša specifikacija koja nastaje zbog loše urađene analize u fazi planiranja i projektovanja korisničkog interfejsa. Izrada prototipova pomaže u prevenciji ovoga jer se na njima mogu uočiti konceptualni problemi sa korisničkim interfejsom. Posledice mogu biti raznorazne, kao na primer gubitak podataka usled zatvaranja aplikacije ukoliko ne postoji upozorenje da dokument nije sačuvan.

265. Šta su neispunjena očekivanja i kakve uzroke i posledice imaju?

Pod neispunjenim očekivanjima podrazumevamo dobijanje neočekivanog (neispravnog) rezultata usled neke operacije. Neki takvi primeri su neispravni numerički i drugi rezultati, neispravno reagovanje na akciju korisnika ili neispravan prikaz u korisničkom interfejsu, pogrešan pozitivan ili negativan odgovor i slično. Uzroci se mogu naći u raznim fazama softvera, od faze definisanja zahteva gde mogu nastati zbog propusta u komunikaciji, pa sve do implementacije samog programa zbog greške programera. Posledice opet mogu biti svakakve, od neugodnosti u radu sa softverom, do potpuno neupotrebljivih rezultata rada tog softvera.

266. Objasniti problem slabih performansi i moguće uzroke.

Slabe performanse predstavljaju sporo reagovanje programa na zahteve korisnika. Iako je najčešće u prvom planu sama ispravnost konačnog rezultata (nije nam bitna brzina dobijanja rezultata ako je on na kraju pogrešan), performanse su svakako bitne, Naročito u interaktivnim sistemima (zamislimo program za crtanje gde je potrebno par sekundi da se nacrtani pikseli pojave na ekranu). Uzroci mogu biti raznorazni i mogu se naći u svim fazama softvera. Ukoliko je do propusta došlo u fazi kodiranja, on se često može ukloniti primenom boljeg algoritma ili neke tehnike (npr. upotreba više niti). Međutim, ako je propust nastao u fazi procenjivanja i planiranja opterećenja, on se mnogo teže prevazilazi.

267. Koje okolnosti posebno pogoduju nastanku bagova? Objasniti dve.

Nastanku bagova pogoduju:

- Nedovoljna stručnost tima - tim koji nije obučen, ne razume zahteve ili ne posvećuje pažnju kvalitetu nikako ne može proizvesti kvalitetan softver. On će skoro sigurno biti pun propusta.
- Nepotrebno povećan nivo stresa u timu - stres loše utiče na produktivnost programera i povećava šansu da napravi propust. On je često izazvan kratkim (nekad i nemogućim) rokovima i prekovremenim radom, što tera programera da ne posvećuje dovoljno pažnje kodu koji piše.

268. Koje okolnosti smanjuju verovatnoću nastajanja bagova? Objasniti dve.

Manje bagova će biti napravljeno ukoliko postoji:

- Visoka stručnost tima - sistematičan tim čiji se članovi stalno usavršavaju i imaju dobru međusobnu i komunikaciju sa klijentom će praviti manje propusta.
- Posvećenost kvalitetu - tim posvećen kvalitetu će pisati robusan softver i pre svega će posvetiti veliku pažnju pisanju testova, što će dovesti do lakšeg uočavanja potencijalnih propusta i njihovog ispravljanja.

269. Koje okolnosti olakšavaju pronalaženje uzroka bagova? Objasniti dve.

U pronalaženju uzroka bagova pomažu:

- Informisanost - čuvanje istorije verzija programskog koda omogućava pronalaženje verzije u kojoj je problem prvo krenuo da se ispoljava. U otkrivanju tačnog uzroka pomaže i čuvanje komunikacije članova tima (međusobne i sa klijentom), kao i redovno pisanje dokumentacije.
- Sistematičnost i redovnost - česta i planska izgradnja koda i posvećivanje pažnje upozorenjima koja se dobijaju od prevodioca mogu biti ključni za otkrivanje bagova. Takođe, poštovanje pravila kodiranja i komentarisanja u okviru projekta omogućava lakše snalaženje u kodu, a time i lakše lociranje bagova.

270. Koji su osnovni pristupi problemu debagovanja? Objasniti ukratko njihove odnose.

Osnovne vrste pristupa su neformalni, empirijski i heuristički metod. Neformalni pristup debagovanju je površan pristup koji podrazumeva samo ponovno pokušavanje sa nekom jednostavnom popravkom sve dok se problem ne reši. Empirijski pristup je kompleksniji. Naime, lociranje problema je slično procesu istaživanja u prirodnim naukama, što uključuje i postavljanje hipoteze o uzroku problema. Heuristički metod je nadogradnja empirijskom i pomaže u izboru te hipoteze.

271. Opisati empirijski (naučni) metod debagovanja.

Proces lociranja problema u empirijskom metodu debagovanja je sličan procesu istraživanja u prirodnim naukama. On podrazumeva posmatranje uočenog problema, postavljanje hipoteze o njegovom uzroku, pravljenje predviđanja o ponašanju na osnovu te hipoteze, i eksperimentalno proveravanje ispravnosti predviđanja. Ovaj proces se ponavlja sve dok se ne potvrdi ispravnost hipoteze (ili se ostane bez mogućnosti za njeno dalje unapređenje). Međutim, najveći problem sa ovakvim pristupom je kako uopšte postaviti hipotezu, u čemu pomaže heurističko debagovanje kao nadgradnja ovog metoda.

272. Objasniti heurističko debagovanje.

Heurističko debagovanje počiva na primenjivanju određenog skupa pravila i principa za koje se pokazalo da su delotvorni u procesu debagovanja. Ova pravila su tzv. heuristike. U tom smislu, heurističko debagovanje predstavlja nadogradnju empirijskog debagovanja jer pruža odgovor na to kako treba birati hipoteze prilikom lociranja uzroka problema. Naravno, heuristike ne garantuju optimalno rešenje, pa i ovaj metod debagovanja nije bez svojih mana.

273. Navesti bar 6 osnovnih pravila za debagovanje (po D. Ejgensu).

- Razumeti sistem
- Navesti sistem na grešku
- Najpre posmatrati pa tek onda razmišljati
- Podeli pa vladaj
- Praviti samo jednu po jednu izmenu
- Praviti i čuvati tragove izvršavanja
- Proveravati i naizgled trivijalne stvari
- Zatražiti tuđe mišljenje

274. Objasniti pravilo debagovanja "Razumeti sistem".

Da bi se neki problem prevazišao, prvo se mora razumeti, a za to je potrebno razumevanje računarskog sistema u kojem je on i nastao. U ovom kontekstu, računarski sistem obuhvata i razvijani softver, i operativni sistem, ostale softverske komponente, pa i sam hardver računarskog sistema. Međutim, u praksi se najčešće ograničavamo samo na razvijani softver prilikom debagovanja, mada ne treba odmah isključiti i ostale opcije (nesistematično isključivanje opcija je loša praksa). Radi upoznavanja sa sistemom, često se čita dokumentacija. Ona vrlo često može biti preobimna za detaljno proučavanje, ali u slučaju debagovanja, najčešće je dovoljno usmeriti se na

određene delove dokumentacije za koje nam se čini da imaju veze sa problemom. Važno je naglasiti da je potencijalan problem dokumentacije taj da ona može biti neažurna, što može proizvesti dodatne probleme.

275. Objasniti pravilo debagovanja "Navesti sistem na grešku".

Da bi se neki problem rešio, prvo ga moramo posmatrati i analizirati. Ovo se najbolje postiže time što ćemo nekim nizom koraka navesti sistem da ispolji tu grešku. Osim posmatranja samog problema, ovo omogućava i posmatranje uzroka tog problema, što olakšava lokalizaciju problema. Još jedan bitan razlog za određivanje niza koraka koji dovodi do greške je mogućnost provere da li je greška u tom slučaju otklonjena. Može se čak i reći da ako ne znamo kako da navedemo sistem na grešku, onda ni ne razumemo grešku, pa je ne možemo ni popraviti. Međutim, ovaj proces nije uvek jednostavan. Često je potrebno, osim samog niza koraka, otkriti i tačne uslove u sistemu pod kojima taj niz koraka dovodi do greške, a takvi uslovi mogu biti raznovrsni (npr. datum na računarskom sistemu, vremenska zona, kodni raspored tastature, ...). Sve ovo je potrebno pažljivo i precizno dokumentovati. Ima i situacija kada se greška veoma retko ispoljava (npr. jednom u 100 pokretanja), što može biti prouzrokovano raznim faktorima (nedefinisano ponašanje usled upotrebe neinicijalizovanih podataka, loša sinhronizacija niti, uticaj spoljašnjih uređaja, ...), što često zahteva dodatnu kontrolu uslova radi otkrivanja uzroka greške. Još jedna stvar koju je važno napomenuti je da nakon nalaženju postupka za ponavljanje greške, dobro je probati sa nalaženjem još sličnih postupaka, jer lako se može desiti da dođemo u situaciju da "lečimo simptome" greške umesto samog uzroka.

276. Objasniti pravilo debagovanja "Najpre posmatrati pa tek zatim razmišljati".

Da bi uspešno otklonili problem, potrebne su nam informacije o njemu, a te informacije se dobijaju posmatranjem problema (što se najčešće postiže navođenjem sistema na grešku). Ne treba slepo krenuti sa ispravljanjem problema bez tih informacija (iako iskusniji programeri možda mogu uspešno da otklone problem sa manje informacija, one se nikada ne smeju potpuno zanemariti). Za posmatranje problema se koriste spoljašnji alati (npr. softverski debageri, postoje čak i hardverski alati koji omogućavaju izvršavanje softvera na jednom, a posmatranje na drugom uređaju) i unutrašnji alati (ugradnje u sam programski kod koji se razvija i debaguje). Međutim, upotreba alata mora biti oprezna - svako posmatranje menja sistem, jer su i sami posmatrači njegov deo (efekat posmatrača). Ovo može dovesti do toga da sa alatima promenimo uslove koji su potrebni za ispoljavanje greške.

277. Objasniti pravilo debagovanja "Podeli pa vladaj".

Sistem je često previše složen da bi se svi njegovi elementi sagledali jednako detaljno, pa se iz tog razloga teži određenoj lokalizaciji problema tako da se pažnja najviše posveti delovima koji povezuju uzrok problema i uočene posledice. Ovo se može postići primenom pravila podeli pa vladaj. Ideja je da postepenim aproksimacijama sužavamo oblast traženja greške. Prvo je potrebno odabrati dovoljno široku početnu oblast, a zatim postaviti hipotezu o lokaciji greške (ovakva hipoteza bi trebalo da podeli posmatranu oblast na dve približno jednake podoblasti). Nakon postavljanja hipoteze, potrebno je testirati i njenu tačnost (npr. isključivanjem koda) i na osnovu rezultata sužavati oblast. Postupak je potrebno ponavljati dok ne dobijemo dovoljno preciznu lokaciju greške.

278. Objasniti pravilo debagovanja "Praviti samo jednu po jednu izmenu".

U procesu debagovanja, menjanje koda je neophodno, bilo da se ono radi u fazi lokalizacije (npr. isključivanje dela koda radi provere hipoteze o lokaciji baga) ili u fazi otklanjanja greške (pronađen bag treba i ukloniti, a to se vrši nekom izmenom koda). Međutim, treba imati u vidu da svaka izmena koda dovodi do plodnijeg okruženja za nastajanje grešaka u odnosu na pisanje sasvim novog koda, pa je preporuka praviti samo jednu po jednu izmenu, a nikako više njih u paketu. Ovo može dovesti do više problema, kao što je na primer mogućnost da jedna izmena rešava problem, a da su ostale suvišne, pa čak i da ni jedna izmena nije dobra. Ukoliko se uoči da neka izmena ne rešava problem, potrebno je da se ona prvo poništi, pa da se traga za drugom potencijalnom izmenom.

279. Objasniti pravilo debagovanja "Praviti i čuvati tragove izvršavanja".

Tragovi izvršavanja predstavljaju jedno od najkorisnijih sredstava za skupljanje informacija o toku izvršavanja, što pomaže prilikom lokalizovanja problema. Ono je Naročito korisno u slučajevima gde izvršavanje treba ponoviti

veliki broj puta da bi se greška ispoljila, a i u situacijama gde je praćenje sistema otežano (npr. distribuirana okruženja). Pod tragovima izvršavanja, obično se misli na delove programskog koda koji automatski zapisuju informacije o stanju programa i toku njegovog izvršavanja u neku izlaznu datoteku (trace file). Drugi oblik su manuelni tragovi izvršavanja, tj. dnevnici aktivnosti korisnika koji testira softver (zapisivanje redosleda koraka). Glavni cilj upotrebe tragova izvršavanja je uspostavljanje relacije između problema (simptoma) i potencijalnih uzroka.

280. Objasniti pravilo debagovanja "Proveravati i naizgled trivijalne stvari".

Ponekad su trivijalne i nama naizgled nemoguće stvari uzrok problema. Naime, dešava se da, povučeni ovakvom pretpostavkom, programeri krenu u lokalizovanje problema na taj način da potpuno zanemare mesto sa pravim uzrokom. Iz ovog razloga se predlaže da se uvek na početku debagovanja u pitanje dovedu sve "podrazumevane" pretpostavke. Svako zanemarivanje ovakvih "trivijalnih" mogućnosti je zapravo prihvatanje nepotvrđenih hipoteza, što se ne sme raditi.

281. Objasniti pravilo debagovanja "Zatražiti tuđe mišljenje".

U slučaju da smo se "zaglavili" sa rešavanjem nekog problema, ne treba se ustručavati od traženja tuđe pomoći. U ovakvim slučajevima se često dešava da razvijemo neke pretpostavke o uzroku problema koje su potpuno pogrešne i koje nam smetaju da dođemo do pravog uzroka. Zbog ovoga je dobra ideja pozvati nekog, objasniti mu samo problem, i nikako mu ne saopštavati naše pretpostavke o uzroku problema i procesu kojim smo probali da ga otkrijemo (osim ako on to ne zatraži), jer u suprotnom postoji mogućnost da će se i on "zaglaviti" u istom uglu razmišljanja kao i mi. Pomoć je ponekad potrebno zatražiti i od eksperta u slučaju nepoznavanja neke od brojnih tehnika i tehnologija koje se prepliću u projektu koji razvijamo. Danas je pristup informacijama lakši nego pre, pa se i Internet može koristiti u ovu svrhu - postoje razne diskusione grupe, forumi i slično koji mogu biti od pomoći (npr. moguće je da je neko pre imao isti problem kao i mi u slučaju upotrebe neke biblioteke).

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

Suština ovog pravila je da problemi nikada ne nestaju sami od sebe. Ponekad se dešava da problem jednostavno prestane da se ispoljava (ili da se nama barem tako čini). Ovo može biti, na primer, posledica drugih promena u kodu koje su nastale u međuvremenu (od trenutka kada je greška prvobitno napravljena) koje su na neki način samo zamaskirale problem, ali ne i rešile ga. Iako je možda primamljivo potpuno zaboraviti na prethodno postojanje problema, vrlo je verovatno da će on krenuti da se ponovo ispoljava u nekom trenutku, onda kada nam to najmanje odgovara. Naravno, moguće je da su te druge promene zaista rešile problem, ali ovo je potrebno detaljno ispitati, na primer poređenjem sa prethodnom verzijom koda (onom u kojoj se problem još uvek ispoljavao).

283. Navesti najvažnije tehnike za prevenciju nastajanja bagova.

- Pisanje pretpostavki
- Ostavljanje tragova
- Komentarisanje koda
- Testiranje jedinica koda

284. Navesti osnovne unutrašnje tehnike i alate za debagovanje.

- Proveravanje pretpostavki
- Pravljenje tragova izvršavanja
- Umetanje specifičnih delova koda
- Dodavanje testova jedinica koda
- Pisanje jasnog i razumljivog koda

285. Objasniti pisanje pretpostavki kao tehniku prevencije nastajanja bagova.

Proveravanje pretpostavki (assert) je tehnika koja omogućava da se na odgovarajućem mestu u kodu provere neki zadati uslovi za koje se očekuje da na tom mestu moraju da važe. U slučaju da su ti uslovi neispunjeni, najčešće se prekida rad programa i saopštava se poruka o razlogu. Pretpostavke se proveravaju nekim oblikom funkcije ili makroa assert. Predlaže se da pretpostavke budu jednostavne, jer u slučaju da su složene i da ne prođu, onda nećemo odmah znati zbog kog dela pretpostavke se to desilo. Pretpostavke se najčešće koriste radi debagovanja u fazi razvoja, ali često samo usporavaju performanse programa u produkciji. Međutim, nije poželjno brisati pretpostavke iz koda svaki put pred puštanje u produkciju - umesto toga, može se koristiti NDEBUG makro za gašenje proveravanja pretpostavki (u C++-u). Alternativa za to su statičke pretpostavke (od C++ 11) koje provere vrše još u fazi prevođenja (tamo gde je to moguće).

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

Pitanje 279.
Pravljenje tragova se obično implementira kroz upotrebu makroa ili potprograma koji zapisuju odgovarajuće informaciju u nekoj datoteci ili izlaznom toku za greške. Ovaj zapis često sadrži i vreme i mesto (u kodu) izvršavanja radi lakše analize. Oni usporavaju rad softvera, pa je preporučljivo isključiti ih pre puštanja u produkciju. U velikim projektima česta je i upotreba posebnih biblioteka za pravljenje tragova.

287. Objasniti komentarisanje koda kao tehniku prevencije nastajanja bagova.

Komentarisanje je poseban vid dokumentacije, pa samim tim (ukoliko se koristi na ispravan način) povećava čitljivost i razumljivost koda, što dodatno dovodi do manjeg rizika od pravljenja bagova. Sugeriše se da se komentarisanje koristi ne za tehničko objašnjavanje načina rada delova koda, već za opisivanje motivacije i razloga za njihovo pisanje, kao i njihovog načina povezivanja sa drugim delovima koda. Komentare nikako ne treba koristiti za opisivanje trivijalnih (očiglednih) stvari. Interesantno je da postoje i alati za automatsko generisanje dokumentacije iz komentara, npr. Doxygen.

288. Objasniti testiranje jedinica koda kao tehniku prevencije nastajanja bagova.

Testovi su vid neformalne verifikacije - kolekcija testova omogućava proveru da li neka jedinica koda radi ispravno ili ne. Ovo je Naročito bitno kod pravljenja izmena u kodu, npr. usled refaktorisanja - pokretanjem testova možemo proveriti da li je refaktorisanje dovelo do promene rada sistema (ne bi trebalo), a ukoliko jeste, možemo videti u kom slučaju se to desilo, što olakšava lokalizaciju uzroka.

289. Navesti najvažnije spoljašnje tehnike i alate za debagovanje.

- Debageri
- Profajleri
- Alati za dinamičku analizu
- Čistači (Sanitizer)

290. Navesti osnovne tehnike upotrebe debagera.

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

291. Objasniti tehniku upotrebe debagera "Izvršavanje korak po korak".

Programer ima mogućnost praćenja izvršavanja programa od početka do njegovog kraja tako što se izvršava jedna po jedna naredba. Ukoliko se naiđe na poziv potprograma, programer može da bira da li želi da uđe u njegovo telo i prati njegovo izvršavanje korak po korak (step into), ili želi da posmatra poziv potprograma kao

jednu naredbu (step over). U većini debagera je moguć i izlazak iz potprograma (step out), kao i izvršavanje programa do neke naznačene naredbe (run to).

292. Objasniti tehniku upotrebe debagera "Postavljanje tačaka prekida".

Umesto izvršavanja programa korak po korak, što može da bude dugotrajan proces, programer može postaviti tačke prekida (breakpoint) na određene naredbe u programu. Izvršavanje se zatim može pokrenuti i ono će stati kada se naiđe na neku od naznačenih tačaka prekida, kada se kontrola prepušta programeru (izvršavanje se može nastaviti korak po korak ili do naredne tačke prekida). Često je moguće i postavljanje uslovnih tačaka prekida.

293. Objasniti tehniku upotrebe debagera "Praćenje vrednosti promenljivih".

Tačke prekida se mogu postavljati i na podatke (watchpoint). Naime, tačka prekida se vezuje za neku oblast u memoriji koja odgovara datom podatku i svaki put kada program pristupi toj zoni memorije, zaustavlja se izvršavanje programa. Ove tačke prekida mogu biti uslovne, a uslov može obuhvatati i vrstu pristupa (čitanje ili pisanje).

294. Objasniti tehniku upotrebe debagera "Praćenje lokalnih promenljivih".

Ukoliko je program privremeno zaustavljen, programer može pregledati stanje lokalnih promenljivih (koje su jedan od osnovnih elemenata stanja programa) koje su u opsegu u tom trenutku.

295. Objasniti tehniku upotrebe debagera "Praćenje stanja steka".

Programski stek služi za prenošenje argumenata i rezultata potprograma, kao i za čuvanje adrese povratka iz potprograma (adrese funkcije koja je pozvala potprogram). Analiza steka omogućava programeru da utvrdi kako je tok programa došao do trenutnog potprograma (odakle je pozvan trenutni potprogram, odakle onaj koji je pozvao njega, ...). Praćenjem stanja lokalnih promenljivih uz praćenje steka pomaže u otkrivanju odakle potiču eventualno neispravne vrednosti argumenata potprograma.

296. Objasniti tehniku upotrebe debagera "Praćenje rada na nivou instrukcija i stanja procesora".

Iako je debagovanje programa lakše uz posmatranje izvornog koda, nekada je neophodno da se program posmatra i izvršava na nivou mašinskih instrukcija. Neki slučajevi kada je ovo potrebno su ako želimo da vidimo da li je neka optimizacija izvršena na odgovarajući način, da proverimo da li je neki deo koda preveden tako da se izvršava atomično i slično. Takođe, ovakav vid upotrebe debagera je često neophodan za debagovanje umetnutog asemblerskog koda (asembler pisan u okviru teksta programa na višem programskom jeziku radi izvlačenja maksimalnih mogućih performansi iz tog dela programa), jer je to skoro nemoguće bez mogućnosti praćenja izvršavanja programa na nivou mašinskih instrukcija. Ukoliko se tok izvršavanja programa prati na nivou mašinskih instrukcija, onda se stanje programa mora pratiti posmatranjem stanja procesora (koje čine vrednosti registara). I sa ovakvim vidom debagovanja, moguće je postavljanje tačaka prekida (samo što se one u ovom slučaju vezuju za mašinske instrukcije, a ne za konstrukcije višeg programskog jezika).

297. Šta je konkurentno izvršavanje?

Konkurentno izvršavanje dva (ili više) posla znači da se oni izvršavaju u istim vremenskim intervalima, ali tako da nije unapred poznata njihova međusobna vremenska lociranost. Ovo znači da u jednom vremenskom intervalu, poslovi mogu da se izvršavaju naizmenično (isprepletano - malo jedan, malo drugi), ali ne znamo tačno kojim redom će svaki od njih dobijati procesorsko vreme.

298. Objasniti pojam paralelno izvršavanje.

Paralelno izvršavanje dva (ili više) posla znači da postoji period vremena u kojem se oni zaista istovremeno izvršavaju. Ovakav vid izvršavanja zahteva više izvršnih jedinica (procesora, tj. jezgara procesora).

299. Objasniti pojam distribuirano izvršavanje.

Distribuirano izvršavanje dva (ili više) posla znači da se oni izvršavaju u različitim adresnim prostorima. Zanimljivo je da su starije definicije, pod distribuiranim izvršavanjem, podrazumevale izvršavanje na različitim računarskim sistemima, međutim, moguće je da više računarskih sistema deli istu radnu memoriju, što dovodi do rada sličnom konkurentnom ili paralelnom na jednom računaru sa više procesora.

300. Objasniti pojam proces.

Proces je instanca programa koja se izvršava na računarskom sistemu (proces je program u izvršavanju). On obuhvata kod programa kojem odgovara (programa koji se izvršava) i tekuće stanje procesa koje obuhvata:

1. Podatke o izvršavanju:
   - Stanje izvršavanja (npr. spreman, radi, čeka, stoji, ...)
   - Brojač instrukcija (adresa naredne instrukcije koja treba da se izvrši)
   - Sačuvane vrednosti registara procesora
2. Informacije o upravljanju resursima:
   - Informacije o memoriji (tablica stranica, podaci o alokaciji memorije, ...)
   - Deskriptori datoteka
   - Ostali resursi (U/I zahtevi, ...)

301. Objasniti pojam nit.

Nit izvršavanja je komponenta procesa koja se izvršava sekvencijalno - jedan proces može imati više niti (ako računarski sistem to podržava) i svaki proces započinje sa jednom glavnom niti. Svaka nit čuva zasebne podatke za podatke o izvršavanju. Ostali podaci (kod programa koji se izvršava i informacije o upravljanju resursima) se vode na nivou procesa i zajednički su za sve njegove niti.

302. Zašto se uvodi koncept niti, ako već postoji koncept procesa?

Niti se uvode radi smanjivanja cene promene konteksta. Naime, promena konteksta je promena stanja procesora koja je neophodna u slučaju kada procesor prelazi sa izvršavanja koda jednog procesa na izvršavanje koda drugog procesa. U tom slučaju, potrebno je zapisati stanje prethodno izvršavanog procesora u memoriju (radi nastavljanja izvršavanja kasnije) i učitavanje stanja narednog procesa iz memorije. Ovo može biti veoma skupa operacija u vidu procesorskog vremena, a pored toga, i sami podaci o stanju procesa su poprilično obimni, pa zauzimaju dosta memorije (takođe, skoro uvek dolazi do promašaja u kešu prilikom promene konteksta). Za razliku od procesa, podaci koji su zasebni za svaku njegovu nit su relativno lagani (stanje izvršavanja niti, brojač instrukcija, sačuvane vrednosti registara), dok se ostali podaci vode na nivou samog procesa i dele se između svih njegovih niti. Iz tog razloga je i promena konteksta dosta efikasnija, ali važno je naglasiti da ovo važi samo ako se prelazi sa izvršavanja jedne niti na drugu nit istog procesa, dok u slučaju da se prelazi sa izvršavanja jedne niti na drugu nit nekog drugog procesa, onda se mora platiti puna cena promene konteksta.

303. Objasniti sličnosti i razlike niti i procesa.

I niti i procesi se mogu izvršavati konkurentno i/ili paralelno, ali samo se procesi mogu izvršavati distribuirano, jer sve niti jednog procesa moraju raditi u istom adresnom prostoru. Važno je naglasiti da procesi, bar ne neposredno, ne dele resurse međusobno, pa se komunikacija između njih mora odvijati pomoću drugih mehanizama (mehanizmi za međuprocesnu komunikaciju, npr. signali, soketi, cevi, deljena memorija, ...), ali zato često nije potrebno eksplicitno staranje o sukobljavanju oko resursa (ponekad je i potrebno, npr. ako su fajlovi deljeni resursi). Niti jednog procesa dele sve resurse tog procesa, pa je komunikacija između njih obično pomoću tih deljenih resursa. Međutim, u ovom slučaju je najčešće potrebno staranje o sukobljavanjima oko resursa.

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

Osnovna podrška za rad sa nitima u C++-u je pružena kroz klasu `thread` (od C++ 11). Osnovni metodi koje ona pruža su:

- `thread(fn)` - konstruktor koji prima funkciju bez argumenata koja će se izvršavati na toj niti
- `thread(fn, arg1, ...)` - konstruktor koji prima funkciju sa argumentima i odgovarajuće argumente
- `void join()` - metod za čekanje na završetak rada niti

Pravljenjem niti, ona se i pokreće.

305. Objasniti kako se programiraju niti pomoću biblioteke Qt. Klase, metodi, ...

Osnovna podrška za rad sa nitima uz pomoću Qt-a je pružena kroz klasu `QThread`. Ona se koristi tako što se pravi konkretna klasa niti koja nasleđuje `QThread` i koja mora da implementira metod `void run()`. Još neki metodi/slotovi/signali koji su sadržani u klasi `QThread`:

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()`

306. Navesti i ukratko objasniti osnovne operacije sa nitima.

- Pravljenje - Pravljenje niti na nivou operativnog sistema obično podrazumeva i započinjanje njenog izvršavanja.
- Dovršavanje - Nit se dovršava kada se završi izvršavanje "glavne" funkcije niti. Zavisno od biblioteke koja se koristi, oslobađanje resursa niti može a i ne mora biti automatsko po završetku rada.
- Suspendovanje i nastavljanje - Neki operativni sistemi (npr. Windows) omogućavaju privremeno suspendovanje niti da bi se njeno izvršavanje nastavilo kasnije. Ukoliko to nije podržano od strane operativnog sistema, odgovarajuće ponašanje se može ostvariti samo složenijim mehanizmima za sinhronizaciju (jedan od razloga je potencijalno nekontrolisano držanje zaključanih resursa od strane suspendovane niti).
- Prekidanje - Načelno, jedna nit može prekinuti izvršavanje druge niti (dovoljan jedan sistemski poziv), ali je ovo jako osetljiva operacija (npr. zbog mogućeg ugrožavanja konzistentnosti deljenih podataka usled prekida niti).
- Čekanje - Elementarni vid sinhronizacije niti je najobičnije čekanje (jedna nit čeka da druga završi sa radom). Međutim, obično se implementiraju neki složeniji mehanizmi od čekanja u "praznom hodu" (npr. slanje nekog deljenog signala od strane niti koja je završila svoj rad nitima koje čekaju).

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

Pravljenje niti na nivou operativnog sistema obično podrazumeva i započinjanje njenog izvršavanja. U skladu sa time, konstruktor klase `std::thread` će napraviti i nit na nivou operativnog sistema i odmah je pokrenuti, pa ne postoji poseban metod za pokretanje.

308. Objasniti detaljno operaciju pravljenja niti. Kako se implementira pomoću biblioteke Qt?

Pravljenje niti na nivou operativnog sistema obično podrazumeva i započinjanje njenog izvršavanja. Međutim, za razliku od klase `std::thread`, pravljenje objekta niti `QThread` ne podrazumeva i pravljenje niti na nivou operativnog sistema, već se to radi tek pozivanjem metoda `start()` (tada se nit i pokreće).

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

Nit se dovršava kada se završi izvršavanje "glavne" funkcije niti. Ovo ne znači i da će objekat niti biti automatski obrisan nakon toga.

310. Objasniti detaljno operaciju dovršavanja niti. Kako se implementira pomoću biblioteke Qt?

Po završetku izvršavanja "glavne" funkcije niti, većina resursa vezanih za nju (nit) se oslobađa. Jedino što ostaje je konačan status niti (rezultat izvršavanja).

311. Objasniti detaljno operaciju suspendovanja i nastavljanja niti. Kako se implementira pomoću standardne biblioteke C++-a?

Neki operativni sistemi (npr. Windows) omogućavaju privremeno suspendovanje niti da bi se njeno izvršavanje nastavilo kasnije. Ukoliko to nije podržano od strane operativnog sistema, odgovarajuće ponašanje se može ostvariti samo složenijim mehanizmima za sinhronizaciju (jedan od razloga je potencijalno nekontrolisano držanje zaključanih resursa od strane suspendovane niti). U slučaju standardne biblioteke C++-a, niti se ne mogu suspendovati i nastavljati od strane drugih niti, već to mogu uraditi samo same sebi. Ove operacije se implementiraju u okviru prostora imena `this_thread` (operacije nad tekućom niti) u vidu funkcija `sleep_for`, `sleep_until` i `yield`.

312. Objasniti detaljno operaciju suspendovanja i nastavljanja niti. Kako se implementira pomoću biblioteke Qt?

Neki operativni sistemi (npr. Windows) omogućavaju privremeno suspendovanje niti da bi se njeno izvršavanje nastavilo kasnije. Ukoliko to nije podržano od strane operativnog sistema, odgovarajuće ponašanje se može ostvariti samo složenijim mehanizmima za sinhronizaciju (jedan od razloga je potencijalno nekontrolisano držanje zaključanih resursa od strane suspendovane niti). Qt ne implementira ovu mogućnost.

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

Načelno, jedna nit može prekinuti izvršavanje druge niti (dovoljan jedan sistemski poziv), ali pošto je to opasno (npr. po konzistentnost deljenih podataka), standardna biblioteka ne nudi mogućnost da jedna nit prekine izvršavanje druge niti.

314. Objasniti detaljno operaciju prekidanja niti. Kako se implementira pomoću biblioteke Qt?

Jedna nit može prekinuti izvršavanje druge niti (dovoljan jedan sistemski poziv), ali je ovo jako osetljiva operacija (npr. zbog mogućeg ugrožavanja konzistentnosti deljenih podataka usled prekida niti). Qt implementira ovu operaciju kroz metod `void terminate()`. Ipak, ovaj metod ne garantuje da će nit zaista biti prekinuta, da li će se to zaista desiti zavisi od operativnog sistema. Ako je potrebno da se ona pouzdano završi, potrebno je sačekati da se nit završi nakon pozivanja ovog metoda.

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

Elementarni vid sinhronizacije niti je najobičnije čekanje (jedna nit čeka da druga završi sa radom). Međutim, obično se implementiraju neki složeniji mehanizmi od čekanja u "praznom hodu" (npr. slanje nekog deljenog signala od strane niti koja je završila svoj rad nitima koje čekaju). U standardnoj biblioteci se čekanje da nit završi sa radom izvodi pozivanjem metoda `void join()`.

316. Objasniti detaljno operaciju čekanja niti. Kako se implementira pomoću biblioteke Qt?

Elementarni vid sinhronizacije niti je najobičnije čekanje (jedna nit čeka da druga završi sa radom). Međutim, obično se implementiraju neki složeniji mehanizmi od čekanja u "praznom hodu" (npr. slanje nekog deljenog signala od strane niti koja je završila svoj rad nitima koje čekaju). Qt implementira metod `bool wait(unsigned long time = ULONG_MAX)`. Ako se argument ne navede (ili se navede `ULONG_MAX`, on je podrazumevan), metod čeka da se nit završi, koliko god dugo to trajalo. U suprotnom, čeka se najviše `time` milisekundi. Metod vraća true ako nit više nije aktivna, a false ako je još uvek aktivna.

317. Koji su najvažniji problemi pri pisanju konkurentnih programa?

Osnovni problemi pri pisanju konkurentnih programa koje treba prevazići su:

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

318. Šta je muteks? Kako se upotrebljava?

Muteksi su jedan od osnovnih mehanizama za sinhronizaciju konkurentnih programa. Oni imaju semantiku katanca, što znači da se mogu zaključati samo jednom, a ako je muteks već zaključan, novi pokušaj njegovog zaključavanja će morati prvo da sačeka da muteks bute otključan. Muteksi su obično implementirani u okviru operativnog sistema tako da su operacije sa njima atomične. Važno je napomenuti da muteksi sami po sebi ne zabranjuju pristup drugim resursima - na programerima je da usklade pristup zaštićenim resursima tako što će koristiti muteks.

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

Podrška za mutekse u okviru standardne bibloteke C++-a je implementirana u vidu klase `std::mutex`. Zaključavanje muteksa se vrši sa metodom `lock()` ili `try_lock()`, a otključavanje metodom `unlock()`. Problem nastaje ukoliko dođe do izuzetka između zaključavanja i otključavanja muteksa unutar neke niti, tj. ukoliko se muteks ne otključa. Dostupne su i klasa `std::recursive_mutex` koja dopušta rekurzivno višestruko zaključavanje od strane iste niti (takav muteks mora biti otključan isti broj puta koliko je i zaključan), kao i `klasa std::shared_mutex` koja omogućava dva nivoa zaključavanja - shared i exclusive (npr. pogodno ako više niti može da čita neke deljene podatke istovremeno, ali moraju da čekaju ukoliko neka nit krene da piše taj podatak). Alternativa muteksima je šablon funkcije `std::call_once(flag, fn, arg1, ...)` gde je zastavica flag tipa `std::once_flag` (u jednom trenutku može biti aktivan najviše jedan od poziva sa datom zastavicom).

320. Objasniti podršku za mutekse u okviru biblioteke Qt.

U okviru biblioteke Qt, muteksi su pruženi kroz klasu `QMutex`. Zaključavanje je pruženo kroz metode `lock()` i `tryLock()`, a otključavanje kroz metod `unlock()` (treba napomenuti da ako je u ovom trenutku muteks zaključan od strane neke druge niti, proizvodi se greška). Konstruktor klase je zapravo `QMutex(RecursionMode mode = NonRecursive)`, gde ako se prosledi `QMutex::Recursive` kao argument, dobijamo rekurzivni muteks koji može biti zaključan više puta od strane jedne niti (isto toliko puta mora biti i otključan). Dostupna je i klasa `QMutexLocker` koja dosta olakšava rad sa muteksima.

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

Klasa `std::lock_guard` je šablonska klasa koja olakšava rad sa muteksima i rešava problem koji nastaje ukoliko dođe do izuzetka između zaključavanja i otključavanja muteksa (muteks ostaje zaključan). Ideja je da se napravi automatski objekat tipa `std::lock_guard` kojem se u okviru konstruktora prosleđuje muteks. U trenutku konstrukcije, muteks će biti zaključan, a otključava se pri uništavanju katanca (tj. ostaje zaključan do kraja važenja opsega u kojem je definisan automatski objekat katanca). Ukoliko dođe do izuzetka, katanac se uništava i muteks se otključava.

```
std::mutex mtx;
void thread_fnc() {
    ...
    std::lock_guard<std::mutex> lck(mtx);
    ...
}
```

Od verzije C++ 17, uvodi se i `std::scoped_lock` koji omogućava zaključavanje više muteksa istovremeno (takva je bar semantika, čeka se dok se svi zaključaju). Šablonska klasa `std::unique_lock` radi slično kao lock_guard, ali omogućava i eksplicitno zaključavanje i otključavanje pre kraja opsega važenja.

322. Čemu služi klasa QMutexLocker biblioteke Qt? Objasniti detaljno.

U okviru biblioteke Qt, muteksi su pruženi kroz klasu `QMutex`, a rad sa njima je olakšan kroz klasu `QMutexLocker`. Objekat klase `QMutexLocker` se pravi za neki konkretan muteks uz pomoć konstruktora `QMutexLocker(QMutex*)`. U trenutku pravljenja ovog objekta, muteks se zaključava, a u trenutku izlaska iz bloka u kojem je `QMutexLocker` definisan, on se otključava. Ova klasa omogućava ispravan rad čak i u slučaju izuzetaka (katanac se uništava, muteks se otključava). Važno je napomenuti da iz ovih razloga, `QMutexLocker` mora biti automatski objekat (ne sme biti dinamički alociran).

```
QMutex mtx;
void thread_fnc() {
    ...
    QMutexLocker lck(&mtx);
    ...
}
```

323. Šta su katanci? Kako se upotrebljavaju?

Katanci omogućavaju pristup zaključanoj oblasti pod različitim uslovima, pa se time mogu posmatrati kao uopštenje muteksa (koji imaju samo ekskluzivan režim pristupa). Obična upotreba katanca je sledeći slučaj - ako jedna nit čita podatke, tada i ostale niti mogu čitati podatke, a ako jedna nit menja podatke, onda niko drugi ne može da koristi te podatke na bilo koj način (ni za čitanje, ni za pisanje).

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

Semantika katanca je u standardnoj biblioteci C++-a podržana kroz šablonske klase `std::lock_guard`, `std::scoped_lock` (kao lock_guard, samo omogućava zaključavanje više muteksa istovremeno) i `std::unique_lock` (kao lock_guard, ali omogućava eksplicitno otključavanje i zaključavanje pre kraja opsega važenja katanca), plus klasu `std::shared_mutex` koja ima dva nivoa zaključavanja, shared (omogućava deljeni pristup) i exclusive (omogućava ekskluzivni pristup).

325. Objasniti podršku za katance u okviru biblioteke Qt.

Katanci u Qt-ju su implementirani klasom `QReadWriteLock`. Metodi koje pruža za zaključavanje su `lockForRead()` (postavlja katanac za čitanje - ukoliko je već zaključan za pisanje, metod čeka do otključavanja), `lockForWrite` (postavlja katanac za pisanje - ukoliko je katanac zaključan bilo kako, onda se čeka do otključavanja), kao i `tryLockForRead()` i `tryLockForWrite()`. Za otključavanje se koristi metod `unlock()`. Konstruktor prima opcioni argument kojim se može naglasiti da li želimo rekurzivni katanac ili ne.

326. Čemu služi klasa QReadLocker biblioteke Qt? Objasniti detaljno.

Klasa `QReadLocker` je pomoćna klasa za olakšano rukovanje sa katancima. Ona mora biti instancirana kao automatski objekat i kao argument u konstruktoru prima pokazivač na objekat tipa `QReadWriteLock` (katanac). U trenutku konstrukcije ovog objekta, katanac se zaključava za čitanje i ostaje tako zaključan sve do prestanka

važenja automatskog opsega tog objekta. Ovaj pristup osigurava i slučajeve kada nastane izuzetak jer će se tada `QReadLocker` objekat uništiti i katanac će biti otključan.

327. Čemu služi klasa QWriteLocker biblioteke Qt? Objasniti detaljno.

Klasa `QWriteLocker` je pomoćna klasa za olakšano rukovanje sa katancima. Ona mora biti instancirana kao automatski objekat i kao argument u konstruktoru prima pokazivač na objekat tipa `QReadWriteLock` (katanac). U trenutku konstrukcije ovog objekta, katanac se zaključava za pisanje i ostaje tako zaključan sve do prestanka važenja automatskog opsega tog objekta. Ovaj pristup osigurava i slučajeve kada nastane izuzetak jer će se tada `QWriteLocker` objekat uništiti i katanac će biti otključan.

328. Šta je sinhronizacija? Šta se sve može sinhronizovati u konkurentnom programiranju?

Sinhronizacija je postupak usaglašavanja rada izvršnih jedinica (niti ili procesa) na neki način. Ono što je zapravo potrebno sinhronizovati je pristup deljenim resursima da bi se izbegli problemi poput izgubljenih promena, pristupa nepotvrđenim izmenama, neponovljivih čitanja i fantomskih podataka. Neki od mehanizama za sinhronizaciju koji omogućavaju rešavanje ovih problema su muteksi, katanci, semafori i slično.

329. Šta su semafori?

Semafori su jedan od osnovnih mehanizama sinhronizacije u konkurentnom programiranju. Oni interno vode računa o brojaču koji se može povećavati, smanjivati i proveravati. Vrednost brojača semantički odgovara broju slobodnih resursa, tj. ako je vrednost brojača jednaka nula, onda se na semaforu mora sačekati dok se brojač ne poveća, tj. dok se neki resurs ne oslobodi.

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

Standardna biblioteka C++-a pruža podršku za semafore tek od verzije C++ 20. Ona je pružena u vidu klasa `std::counting_semaphore` i `std::binary_semaphore` (semafor sa brojačem koji može biti samo 1 ili 0). Metodi koje ove klase pružaju su `release`, `acquire`, `try_acquire`, `try_acquire_for`, `try_acquire_until`.

331. Objasniti podršku za semafore u okviru biblioteke Qt.

Semafori su u Qt-u pruženi kroz klasu `QSemaphore`. Konstruktor `QSemaphore(int n = 0)` kreira semafor sa brojačem inicijalno postavljenim na n. Dodatni metodi koje klasa pruža su:

- `int available()` (vraća vrednost brojača)
- `void acquire(int n = 1)` (smanjuje vrednost brojača za n, ukoliko je n > available(), čeka se da postane n <= available())
- `void release(int n = 1)` (uvećava vrednost brojača za n)
- `bool tryAcquire(int n = 1)`
- `bool tryAcquire(int n, int timeout)`.

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

Funkcija async asinhrono poziva funkciju koja joj se prosleđuje kao argument i kao rezultat vraća objekat tipa future koji će u nekom trenutku sadržati povratnu vrednost prosleđene funkcije. Funkcija async ima nekoliko verzija, a osnovne su:

- `future async(fn, args...)`
- `future async(policy, fn, args...)`

Politika može biti `std::launch::async`, koja znači da će se napraviti nova nit koja se pokreće asinhrono, ili `std::launch::deffered`, koja znači da se posao odlaže i pokreće u istoj niti tek kada se prvi put zatraži njegov rezultat. Mogu se i navesti obe politike preko njihove binarne disjunkcije, a u tom slučaju način izvršavanja bira sama implementacija.

Šablonska klasa `std::future` omogućava čekanje na rezultat do završetka izvršavanja neke funkcije (u šablonu je parametrizovan povratni tip prosleđene funkcije). Konstruktor prima funkciju, argumente, a opciono i politiku izvršavanja (isto kao i kod async). Klasa, između ostalog, pruža metode `wait()`, `wait_for()` i `wait_until()` koji čekaju na završetak izračunavanja i metod `get()` koji implicitno poziva wait() i vraća rezultat izvršavanja funkcije (važno je napomenuti da get() metod sme da se pozove samo jedanput). Dostupan je i metod `bool valid()` koji proverava da li je stanje objekat ispravno, tj. da li na njemu sme da se pozove metod get() (što odgovara tome da get() nije bio pozvan do sad).

Suštinski, ovo je način da se apstrahuje eksplicitna upotreba niti (prepuštamo kreaciju i rukovanje nitima funkciji async i šablonu future).

333. Objasniti šablone future i promise standardne biblioteke C++-a.

Šablonska klasa `std::future` omogućava čekanje na rezultat do završetka izvršavanja neke funkcije (u šablonu je parametrizovan povratni tip prosleđene funkcije). Konstruktor prima funkciju, argumente, a opciono i politiku izvršavanja (isto kao i kod async). Klasa, između ostalog, pruža metode `wait()`, `wait_for()` i `wait_until()` koji čekaju na završetak izračunavanja i metod `get()` koji implicitno poziva wait() i vraća rezultat izvršavanja funkcije (važno je napomenuti da get() metod sme da se pozove samo jedanput). Dostupan je i metod `bool valid()` koji proverava da li je stanje objekat ispravno, tj. da li na njemu sme da se pozove metod get() (što odgovara tome da get() nije bio pozvan do sad).

Međutim, šablon future je ograničen samo na dohvatanje rezultata funkcije (iz njene povratne vrednosti). Ukoliko je na sličan način potrebno da se čeka na jedan ili više međurezultata, onda se koristi šablon `std::promise`. Objekat šablonske klase `std::promise` predstavlja objekat međurezultata čija vrednost se čeka. Ona pruža metode:

- `set_value(T)`
- `set_value_at_thread_exit(T)` (postavlja vrednost, ali dopušta njenu upotrebu tek kada se nit završi)
- `set_exception(exc)` (postavlja izuzetak - kada se pristupi objektu, ispaliće se ovaj izuzetak)
- `get_future()` (vraća odgovarajući objekat klase future i sme da se upotrebi samo jednom).

334. Kakvi mogu biti potprogrami u kontekstu konkurentnog programiranja?

- Potprogrami sa jedinstvenim pozivanjem
- Potprogrami sa ponovljivim pozivanjem
- Potprogrami bezbedni po niti

335. Šta su potprogrami sa jedinstvenim pozivanjem?

Potprogram (funkcija) je sa jedinstvenim pozivanjem ako ne sme biti istovremeno korisćen u različitim nitima. Uzrok ovoga može biti upotreba globalnih podataka u telu funkcije na način koji nije bezbedan.

336. Šta su potprogrami sa ponovljivim pozivanjem?

Potprogram (funkcija) je sa ponovljivim pozivanjem (reentrant) ako sme da se istovremeno koristi u različitim nitima, ali samo uz pretpostavku da se ne koristi nad istim podacima. Uzrok ovoga može biti upotreba podataka ili resursa na način koji nije bezbedan.

337. Šta su potprogrami bezbedni po niti?

Potprogram (funkcija) je bezbedna po niti ako sme da se istovremeno koristi u različitim nitima bez ograničenja. Ovo se postiže ako funkcija koristi sve podatke (osim lokalnih automatskih) uz puno poštovanje specifičnosti konkurentnog okruženja.

338. Kakve mogu biti klase u kontekstu konkurentnog programiranja? Objasniti.

Klasa, isto kao i potprogram, može biti:

- Sa jedinstvenim pozivanjem - bar jedan metod klase je sa jedinstvenim pozivanjem.
- Sa ponovljenim pozivanjem - nijedan metod klase nije sa jedinstvenim pozivanjem, bar jedan je sa ponovljenim pozivanjem i nijednom podatku se ne može neposredno pristupati.
- Bezbedna po niti - svi metodi su bezbedni po niti i nijednom podatku se ne može neposredno pristupati.

339. Koje su najčešće greške pri pisanju konkurentnih programa? Objasniti.

Najčešće greške su previđanje problema deljenja podataka pri pisanju funkcija i klasa (npr. nismo, ili jesmo, ali loše, sinhronizovali pristup nekim podacima u koje se može i pisati, a može se i čitati) i previđanje neprilagođenosti korišćenih klasa i funkcija kunkurentnim okruženjima (npr. koristimo klasu sa jedinstvenim pozivanjem kao da je bezbedna po niti).

340. Navesti neke načine razvoja programa koji omogućavaju bezbedno pisanje konkurentnih programa. Objasniti.

Neki načini za pisanje koda bezbednog po niti su:

- Pisanje čisto funkcionalnog koda - svi argumenti se prenose po vrednosti, ne koriste globalne podatke, nema bočnih efekata.
- Upotreba lokalnih podataka i podataka koji su lokalni za nit - ovakve podatke ni jedna druga nit ne može da koristi, pa su samim tim oni bezbedni.
- Obezbeđivanje međusobnog isključivanja - koristimo mehanizme sinhronizacije (muteksi, katanci, ...).
- Obezbeđivanje atomičnih operacija

Važno je napomenuti da bezbednost ima svoju cenu. Naime, ovakve funkcije će skoro uvek imati smanjenu efikasnost (prenošenje argumenata po vrednosti, cena zaključavanja pri sinhronizaciji, skupe atomične operacije).

341. Kada dolazi na red staranje o ponašanju koda u konkurentnom okruženju?

O ovome se treba brinuti od samog početka projektovanja neke klase ili funkcije. Ukoliko je potrebno da ona (klasa ili funkcija) radi u okruženju sa više niti, naknadno prilagođavanje za to može biti veoma skupo, sporo, i najverovatnije uz potpuno prepravljanje te klase ili funkcije.

342. Kako se bira gde se i kako postavljaju muteksi i katanci?

U ovom problemu je potrebno biti jako pažljiv. Naime, logično je da muteksi i katanci treba da zaključavaju što manju oblast, jer u suprotnom se smanjuje nivo paralelizacije (zbog nepotrebnog čekanja niti). Međutim, ovo nas može povući da napravimo mnogo katanaca koji će zaključavati male oblasti, što opet nije dobro, jer cena zaključavanja može biti poprilično velika. Odgovor na ovaj problem je da je potrebno naći određeni balans između ova dva principa, ali nikada ne smemo postavljati mutekse i katance odokativno.

343. Navesti osnovne vidove međuprocesne komunikacije.

Signali, soketi, cevi (pipes), imenovane cevi (named pipes), semafori, deljena memorija, datoteke, datoteke preslikane u memoriji (memory mapped files), redovi poruka (message queues).

344. 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 (signali, soketi, cevi, ...). Razlog za ovo je taj što procesi ne dele resurse međusobno. Sa druge strane, niti u okviru jednog procesa imaju deljene resurse (memorija, otvorene datoteke, ...), pa je moguće koristiti te resurse za komunikaciju. Međutim, u ovom slučaju je potrebno voditi računa o sinhronizaciji pristupa deljenim resursima. Često je najbolje koristiti iste mehanizme namenjene za međuprocesnu komunikaciju i za niti.

345. Šta su arhitekture zasnovane na događajima?

U arhitekturama zasnovanim na događajima, komunikacija između objekata se vrši mehanizmom emitovanja događaja i distribuiranja zainteresovanim objektima (za razliku od klasičnih arhitektura gde je komunikacija između objekata bila eksplicitna - npr. program u petlji proverava da li je korisnik pritisnuo neki taster na tastaturi i u skladu sa tim postupa na neki način). Primer ovoga bi bio sledeći - program obaveštava operativni sistem da je zainteresovan da reaguje na pritisak nekog tastera na tastaturi, operativni sistem prepoznaje takav događaj i šalje poruku programu i na kraju program reaguje na tu poruku.

346. Objasniti motivaciju za upotrebu arhitektura zasnovanih na događajima.

Glavna motivacija u upotrebi arhitekture zasnovane na događajima je u smanjivanju zavisnosti između objekata, što smanjuje ukupnu složenost sistema. U slučaju ovakvih arhitektura, uobičajeno je da nivo spregnutosti bude po parametrima ili po porukama, sprega je obično dinamička i ukupan intenzizet spregnutosti je smanjen. Iako je smanjivanje ukupne složenosti sistema dobro, u ovom slučaju ono često ima cenu povećane lokalne složenosti, tj. teže je razumeti način funkcionisanja zasebnih komponenti koje sarađuju bez posmatranja čitavog sistema (iako je složenost samog koda manja).

347. Objasniti osnovne pojmove i koncepte arhitekture zasnovane na događajima.

- Događaj - prepoznatljiv uslov koji inicira obaveštenje.
- Obaveštenje - događajem iniciran signal koji se šalje primaocu.

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

- Generator događaja - objekat koji prepoznaje da je nastupio uslov koji predstavlja neki definisan događaj.
- Mašina za obradu događaja - mesto na kojem se prepoznaje nastali događaj i pokreće odgovarajuća akcija.
- Kanal događaja - mehanizam putem kojeg se događaji prenose od generatora događaja do mašine za obradu događaja.
- Niz događajima upravljanih aktivnosti - mesto ispoljavanja događaja.

349. Objasniti osnovne koncepte primene arhitekture zasnovane na događajima u okviru biblioteke Qt.

U okruženju Qt, klase koje generišu i/ili prihvataju događaje moraju naslediti klasu `QObject`. Takođe, potrebno je navesti makro `Q_OBJECT` na početku deklaracije takve klase. Qt pruža koncept signala (metodi koji proizvode događaje) i koncept slotova (metodi koji obrađuju događaje) i njihovo povezivanje da bi se olakšala primena arhitekture zasnovane na događajima.

350. Objasniti koncept signala u kontekstu primene arhitekture zasnovane da događajima u okviru biblioteke Qt.

Signali su metodi koji proizvode događaje. Unutar klase, deklarišu se u okviru sekcije `signals`. Oni moraju biti metodi tipa void i nemaju implementaciju. Emituju se sa `emit signal(...)`. Ključna reč emit je makro i za standardni C++ ima praznu definiciju, pa se u suštini može i izostaviti, ali to se ne preporučuje (zbog kompatibilnosti koda i radi dokumentovanja).

351. Objasniti koncept slotova u kontekstu primene arhitekture zasnovane na događajima u okviru biblioteke Qt.

Slotovi su metodi koji obrađuju događaje. Unutar klase, deklarišu se u okviru sekcija `private slots` i `public slots`. Implementiraju se kao najobičnije funkcije i moraju biti istog tipa kao signali za čiju su obradu namenjeni (tipovi argumenata i povratne vrednosti (void) se poklapaju).

352. Objasniti povezivanje signala i slotova u slućaju primene arhitekture zasnovane na događajima u okviru biblioteke Qt.

Povezivanje signala i slotova se ostvaruje uz pomoć statičkog metoda `QObject::connect`. Prvi argument ovog metoda je pokazivač na objekat koji emituje signal, drugi argument je signal koji se emituje (navodi se uz pomoć makroa SIGNAL, npr. SIGNAL(izvestajKorak())), treći argument je pokazivač na objekat koji hvata signal, a četvrti argument je slot kojim se signal hvata (navodi se pomoću makroa SLOT, npr. SLOT(izvestajKorak())). Tip signala i slota se mora poklapati.

353. Objasniti odnos arhitektura zasnovanih na događajima i problema kohezije i spregnutosti.

Arhitektura zasnovana na događajima najčešće utiče veoma povoljno po spregnutost komponenti, tj. snižava intenzitet sprege. Još jedna prednost je ta što, ukoliko signali prestanu da budu značajni, možemo jednostavno da prestanemo da ih pratimo bez menjanja komponente koja ih emituje, a ukoliko je potrebno višestruko obrađivanje signala, potrebno je samo dodati nove slotove koji će ih prihvatiti, bez menjanja komponenti koje emituju signale ili drugih komponenti koje već obrađuju te signale. Dodatna osobina ovakve sprege je to što je ona dinamička - objekti koji se povezuju nisu spregnuti u kodu kojim su definisani, već se sprega uspostavlja u posebnom konfiguracionom delu programskog koda.

354. Šta je softverska metrika?

Softverska metrika je nauka koja se bavi merenjem različitih karakteristika softverskih rešenja. Za cilj ima opisivanje stanja razvojnog projekta i da pruži mogućnost poređenja pomoću numeričkih ocena različitih aspekata projekta.

355. Navesti najvažnije tipove softverskih metrika.

- Neposredne (apsolutne) mere - npr. broj linija koda
- Indirektne (izvedene) mere - npr. količnik nekih drugih mera
- Intervalne mere - mera iskazana kao opseg vrednosti
- Normalne mere - mera iskazana rečima, kao npr. "jeste", "nije", "plavo", ...
- Uporedne mere - npr. "veći od"

356. Objasniti vrste metrika u razvoju softvera.

Dve osnovne vrste softverskih metrika su:

- Metrike praćenja razvoja softvera - iskazuju u kojoj meri tok projekta prati planove, pa se time odnose na proces razvoja i upotrebljene resurse (vreme, troškove, ...).
- Metrike (kvaliteta) dizajna softvera - iskazuju neke merljive odlike softvera, pa se time odnose na sam softver.

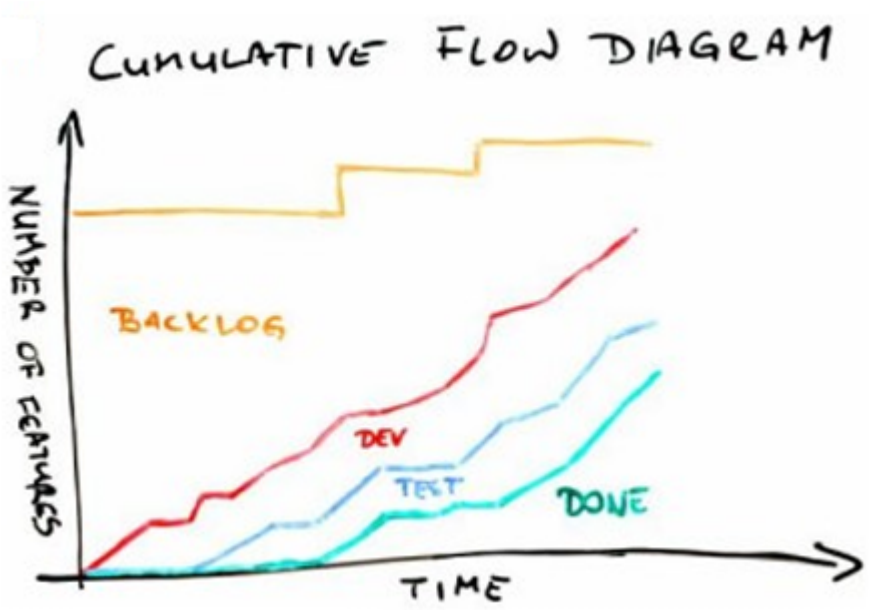
357. Navesti nekoliko metrika praćenja razvoja softvera. Šta one opisuju?

- Mera napretka - opisuje uspešnost praćenja planova (ostvareni poslovi u odnosu na vreme)
- Mera završavanja - obrnuto od mere napretka (preostali poslovi u odnosu na vreme)
- Mera napora - planirani i ostvareni broj radnih sati u odnosu na vreme
- Mera troškova - planirani i ostvareni trošak sredstava u odnosu na vreme
- Broj problema - broj otvorenih i rešenih problema u odnosu na vreme
- Stabilnost zahteva - broj zahteva tokom vremena
- Stabilnost veličine - veličina softvera (obično u broju jedinica koda) tokom vremena

358. Objasniti metriku napretka i kako se ona obično koristi.

Metrika napretka iskazuje uspešnost praćenja planova. Meri se kumulativni broj planiranih i ostvarenih poslova u odnosu na vreme. Da bi odgovarajući dijagram predstavljao realnu sliku stanja, potrebno je da veličina poslova bude relativno ujednačena. Na dijagramu se obično prikazuju prepoznati poslovi, započeti poslovi, poslovi koji se testiraju i dovršeni poslovi.

359. Nacrtati primer dijagrama praćenja napretka sa 4 krive.



360. Navesti nekoliko metrika dizajna softvera. Šta one opisuju?

- Broj jedinica koda - iskazuje broj celina koda (to mogu biti datoteke, klase, paketi, komponente, ...)
- Broj linija koda - iskazuje količinu napisanog koda
- Broj funkcionalnih elemenata koda - opisuje broj funkcionalnih elemenata (npr. broj klasa u paketu, broj metoda u klasi, ...)
- Kohezija jedinice koda
- Spregnutost jedinice koda
- Stabilnost jedinice koda - opisuje tendenciju da se jedinica koda ne menja tokom vremena
- Apstraktnost paketa - predstavlja relativnu zastupljenost apstraktnih klasa u paketu

361. Objasniti metriku stabilnost paketa.

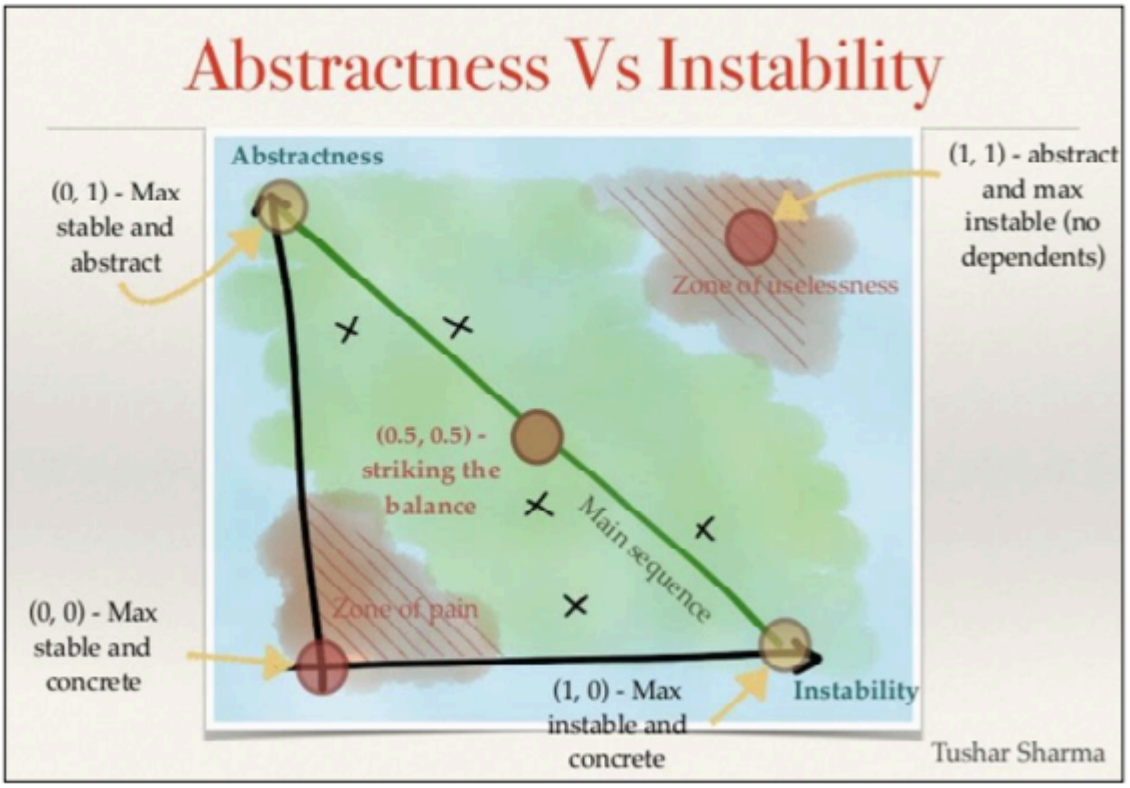
Stabilnost paketa opisuje njegovu tendenciju da se on ne menja tokom vremena. Nestabilnost nekog paketa se može izračunati na sledeći način: $I = \frac{Ce}{Ca+Ce}$, gde je Ce broj klasa u posmatranom paketu koje zavise od klasa u drugim paketima, a Ca broj klasa u drugim paketima koje zavise od klasa u posmatranom paketu. Nestabilnost ima opseg [0, 1]. Obrazloženje za ovu formulu je sledeće - ako više klasa zavisi od posmatranog paketa, onda je manja verovatnoća da ćemo menjati taj paket (jer bi potencijalno morali da menjamo i te zavisne klase), a što više klasa iz posmatranog paketa zavisi od drugih paketa, onda je veća verovatnoća da ćemo morati da ga menjamo zbog mogućih promena u tim drugim paketima. Nije moguće da svi paketi budu stabilni, jer to bi značilo da je čitav sistem nepromenljiv. Poželjno je da zavisnost paketa ide od nestabilnih prema stabilnim.

362. Objasniti metriku apstraktnost paketa.

Apstraktnost paketa je relativna zastupljenost apstraktnih klasa u posmatranom paketu. Može se izračunati na sledeći način: $A = \frac{Na}{Nc}$, gde je Na broj apstraktnih klasa u paketu, a Nc ukupan broj klasa u paketu. Apstraktnost A ima opseg [0, 1].

363. Objasniti odnos metrika stabilnosti i apstraktnosti paketa.

Princip odnosa stabilnosti i apstraktnosti kaže da bi paket trebalo da bude onoliko apstraktan koliko je stabilan, što implicira da bi manje apstraktni paketi trebalo da zavise od više apstraktnih paketa (jer manje stabilni paketi treba da zavise od stabilnijih). Ako posmatramo ravan gde x-osa predstavlja nestabilnost, a y-osa apstraktnost, dobijamo naredni grafik:



Pošto je poželjno da paket bude apstraktan koliko je stabilan, možemo reći da je najbolje da udaljenost tačke na ovom grafiku bude što manja, tj. što bliža glavnoj sekvenci. Udaljenost se računa sa $D = \frac{|A+I-1|}{\sqrt{2}}$.

364. Objasniti metriku funkcionalna kohezija paketa.

Jedan od načina izražavanja funkcionalne kohezije paketa je u narednom obliku: $H = \frac{R+1}{N}$, gde je R broj međusobnih zavisnosti među klasama paketa, N ukupan broj klasa u paketu i H relaciona kohezija. Alternativa je da se svaka zavisnost dveju klasa predstavlja brojem metoda koji tu zavisnost ostvaruju (umesto da posmatramo samo da li su klase zavisne ili ne, tj. 0 ili 1).

365. Šta je ciklomatička složenost?

Ciklomatička složenost koda je način računanja složenosti strukture koda. Ona se predstavlja na sledeći način - neka je dat graf G čiji čvorovi predstavljaju polazne i završne tačke, tačke grananja i tačke spajanja u programskom kodu. Ciklomatička složenost $V(G)$ je broj svih međusobno nezavisnih linearnih puteva kroz taj graf i računa se na sledeći način: $V(G) = E - N + 2P$, gde je E broj grana, N broj čvorova i P broj povezanih komponenti (obično odgovara broju potprograma). Ukoliko u grafu povežemo sve završne tačke sa odgovarajućom ulaznom, onda možemo definisati ciklomatičku složenost sa $V(G) = E - N + P$. Preporučuje se da važi $V(G) \leq 10$.

366. Šta je Holstedova složenost?

Holstedova složenost koda je skup mera složenosti koda koje počivaju na složenosti izraza u kodu. One se računaju na sledeći način:

- N_1 - ukupan broj operatora (pod operatorom podrazumevamo i tipove, zagrade, funkcije, ...)
- n_1 - ukupan broj različitih operatora
- N_2 - ukupan broj operanada
- n_2 - ukupan broj različitih operanada

Metrike:

- $n = n_1 + n_2 \rightarrow$ rečnik programa
- $N = N_1 + N_2 \rightarrow$ dužina programa

- $L = n_1 \log_2 n_1 + n_2 \log_2 n_2 \rightarrow$ procenjena dužina programa
- $V = L \log_2 n \rightarrow$ obim (volumen) programa
- $D = \frac{n_1 \cdot N_2}{2n_2} \rightarrow$ težina (za pisanje i razumevanje) programa
- $E = D \cdot V \rightarrow$ napor za pisanje programa
- $T = \frac{E}{18} \rightarrow$ očekivano vreme pisanja programa
- $B = \frac{E^{\frac{2}{3}}}{3000} \rightarrow$ očekivani broj isporučenih bagova (alternativno $B = \frac{V}{3000}$)

367. Šta su sistemi za kontrolu verzija? Objasniti.

Sistem za kontrolu verzija je softver za upravljanje izmenama u datotekama poput programskog koda, dokumenata i slično. Kao takav, uglavnom se koristi za upravljanje izmenama u okviru softverskog projekta, mada može se koristiti i za druge namene. Primer takvih sistema su Git i SVN.

368. Objasniti arhitekturu i navesti osnovne operacije pri radu sa sistemima za kontrolu verzija.

Arhitektura sistema za kontrolu verzija može biti centralizovana (klijent-server) i decentralizovana. U slučaju centralizovane arhitekture, klijenti su korisnici sistema za kontrolu verzija, a server sadrži spremište (repozitorijum), tj. mesto na kojem se čuvaju različite verzije projekta. U slučaju decentralizovane arhitekture, svaki učesnik sadrži svoju lokalnu verziju spremišta (repozitorijuma), mada skoro uvek postoji i jedno spremište koje je "glavno". Jedna od prednosti decentralizovane arhitekture je to što većinu vremena nije potrebno povezivanje na mrežu sa serverom za rad (osim ako je potrebno izvesti lokalnu kopiju). Osnovne operacije obuhvataju uvoz i izvoz kopija, pravljenje izmena, grananje različitih verzija i još toga.

369. Šta je spremište? Šta sadrži? Kako je organizovano?

Spremište (repozitorijum) je mesto na kome se čuvaju različite verzije projekta, što uključuje i istoriju izmena u okviru verzija i slično. Obično se implementira pomoću odgovarajuće baze podataka.

370. Šta je radna kopija, u kontekstu upotrebe sistema za kontrolu verzija?

Radna kopija je lokalna kopija verzije koja se nalazi na klijentu i koja se menja tokom rada. Moguće je ažuriranje radne kopije preuzimanjem aktuelne verzije iz spremišta, kao i podnošenje izmena u spremište nakon menjanja fajlova radne kopije.

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

Ponekad je pogodno označiti neku verziju radi lakšeg kasnijeg referisanja na nju. Ovako napravljena oznaka (tag) ima svoje ime. Verzije koje se najčešće označavaju su one koje se javno objavljuju ili verzije koje predstavljaju neki interno značajan trenutak.

372. Objasniti grananje u kontekstu upotrebe sistema za kontrolu verzija.

Verzija koja se razvija nezavisno od glavne linije razvoja naziva se grana. Pravljenje grana naziva se grananje i koristi se ukoliko je potrebno istovremeno razvijati više različitih verzija koda (npr. razvijanje novih funkcionalnosti softvera od strane više različitih timova, nova grana za svaku funkcionalnost). U slučaju kada želimo da integrišemo promene nastale u našoj odvojenoj verziji (grani) u onu iz koje smo se odvojili, vršimo spajanje grane nazad u stablo.

373. Šta su konflikti i kako se rešavaju, u kontekstu upotrebe sistema za kontrolu verzija?

Konflikti nastaju usled menjanja istog fajla od strane više klijenata istovremeno, tj. u odnosu na istu početnu verziju, ili u slučaju spajanja grana u kojima se menjao isti fajl. Oni se rešavaju spajanjem verzija, tj. biranjem iz koje verzije će biti preuzete koje izmene.

374. Šta je spajanje verzija, u kontekstu upotrebe sistema za kontrolu verzija?

Spajanje verzija je operacija spajanja dva skupa izmena fajla (ili više fajlova). Ona se primenjuje kada je potrebno rešiti konflikte. Spajanje verzija se sastoji iz skupa pojedinačnih razrešavanja problema u vidu manuelnih intervencija radi rešavanja sukobljenih izmena nad istim dokumentom.

375. Objasniti primer strategije označavanja verzija.

Jedna uobičajena strategija označavanja verzija je sledeća:
`<glavna verzija>.<podverzija>[<popravka>[.<revizija>]]`

Glavna verzija se menja kada se uvode značajne izmene u odnosu na prethodnu verziju softvera, a podverzija kada uvedene izmene nisu samo popravke nekih nedostataka, ali ipak nisu ni toliko obimne da zaslužuju novu glavnu verziju. Popravka se menja ako nisu uvedene nove izmene, već samo ispravljeni neki nedostaci. Revizija označava reviziju popravke i najčešće je potpuno interna oznaka. Glavna verzija i podverzija se najčešće koriste za javni broj verzije, a popravka i revizija se najčešće koriste samo interno u okviru razvojnog tima.

376. Šta su sistemi za praćenje zadataka i bagova? Objasniti namenu i osnovne elemente.

Sistemi za praćenje zadataka su alati za upravljanje evidencijom i komunikacijom vezanom za zadatke u okviru razvoja softvera. Sistemi za praćenje bagova su specijalni slučaj ovih sistema i oni se bave upravljanjem evidencijom i komunikacijom u vezi sa uočenim neispravnostima. Osnovni element sistema za praćenje zadataka su tzv. kartice koje odgovaraju evidentiranim zadacima (ili bagovima) i koje mogu imati različita stanja, kategorije, prioritet, opis i slično.

377. Navesti i ukratko objasniti osnovne koncepte sistema Redmine.

Redmine je jedan od najkorišćenijih sistema za praćenje zadataka i bagova. Može se koristiti direktno iz veb pregledača i ima veliki broj funkcionalnosti, kao što su na primer:

- Više vrsta kartica (bagovi, zadaci, ...)
- Definisanje različitih stanja za vrste kartica
- Grupe korisnika sa različitim pravima
- Automatsko slanje obaveštenja elektronskom poštom
- Dokumentacija
- Pregled programskog koda

378. Objasniti ulogu stanja kartica i način njihovog menjanja (na primeru sistema Redmine).

Jedna kartica odgovara nekom evidentiranom zadatku ili bagu. Stanje kartice opisuje koje se aktivnosti očekuju u odnosu na zadatak, pa je po završetku tih aktivnosti potrebno promeniti stanje u odgovarajuće novo stanje (npr. završeno ako je zadatak potpuno završen). Dodatno, u okviru Redmine-a, karticama je moguće dodeljivati kategorije, prioritet, detaljan opis, mogu se dodeljivati određenom korisniku i još toga.

379. Šta čini dokumentaciju softvera?

Dokumentaciju čini sav propratni tekst, dijagrami, ilustracije uz programski kod, a u nekim slučajevima se čak i sam programski kod može koristiti kao deo dokumentacije. Dokumentacija se ne odnosi samo na gotov program ili postavljen zadatak, već na ceo softverski projekat, što podrazumeva sve segmente posla, od prve zamisli o softveru do gotovog softvera, pa i dalje od toga u vidu uputstva za korišćenje, održavanje i slično. Važno je napomenuti da za različite učesnike u razvoju i upotrebi softvera dokumentacija ima različitu namenu i oblik.

380. Kome je i zašto potrebna dokumentacija?

Dokumentacija je potrebna svima koji će imati kontakta sa projektom na neki način, ali svakome u drugom obliku. Ovo obuhvata i naručioce posla (da bi znao šta je uopšte naručio i da bude siguran da izvođači žnaju šta je on naručio), rukovodioce projekta (da bi videli šta su zahtevi, šta je urađeno, ...), projektante (da znaju šta je potrebno da naprave, da implementatorima zapišu kako zamišljaju da to bude napravljeno, ...), implementatore (da znaju

šta je potrebno napraviti i kako to napraviti, da objasne budućim korisnicima koda šta je šta i čemu služi, a korisnicima softvera šta softver radi i kako se on koristi) i korisnike (da vide čemu softver služi i kako se koristi).

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

Neki opravdani motivi za pravljenje dokumentacije su:

- Da bi se formalizovala specifikacija
- Da bi se podržala komunikacija sa udaljenim timovima (često nisu svi članovi tima na istoj lokaciji, ovo pomaže u njihovom informisanju, outsourcing)
- Da bi se podstaklo širenje i memorisanje informacija u timu
- Da bi se nešto temeljno razmotrilo (pisanje dokumentacije podstiče kritičko razmatranje o onome za šta se piše)

Neki neopravdani motivi su:

- Pravljenje dokumentacija po inerciji (samo jer je klijent tako navikao ili jer slepo pratimo razvojni proces)
- Kao vid obezbeđenja klijentu
- Da bi se definisali ciljevi za drugu grupu, ali tako da je dokumentacija nepotrebno detaljna i preobimna

382. Objasniti ulogu dokumentacije kao vida specifikacije zahteva projekta.

Dokumentacija se može koristiti u svrhu formalizacije specifikacije projekta - formalizuju se interfejs za upotrebu komponenti i ponašanje komponenti, što je važno za sve one koji će razvijati, a i koristiti tu komponentu. Ona obuhvata i konceptualni i implementacioni model komponenti, što je važno za razvijaoce.

383. Objasniti ulogu dokumentacije kao sredstva za komunikaciju.

Dokumentacija može biti od velike koristi u komunikaciji kada je ona otežana, npr. sa udaljenim članovima tima - nisu uvek svi članovi tima na istoj lokaciji, pa dokumentacija pomaže u njihovom informisanju. Od ogromne je važnosti u slučaju projekata otvorenog koda, outsourcing-a i slično.

384. Objasniti ulogu dokumentacije u razmatranju nedoumica u projektu.

Pisanje dokumentacije podstiče kritičko razmatranje o onome za šta se ona piše i predstavlja dobar vid proveravanja pretpostavki (koje mogu biti pogrešne) koje imamo. Iz tih razloga, dokumentacija je dobra za razrešavanje nedoumica vezanih za projekat.

385. Objasniti podelu dokumentacije po nameni.

Dokumentacija se po nameni deli na:

- Korisničku dokumentaciju - koristi je krajnji korisnik softvera i bavi se njegovom primenom.
- Tehničku dokumentaciju - koriste je svi učesnici u razvoju ili održavanju softvera i namenjena je tehničkim licima koja moraju da pruže podršku krajnjim korisnicima.

386. Šta obuhvata korisnička dokumentacija softvera?

- Opis čitavog sistema - opis namene i funkcija koje sistem pruža
- Uputstvo za instalaciju - kako sistem pripremiti za rad u skladu sa specifičnim okruženjem
- Vodič za početnike - jednostavna objašnjenja o započinjanju upotrebe sistema
- Referentni priručnik - detaljan opis svih karakteristika i mogućnosti sistema i načina njihove upotrebe
- Prilog o dopunama / opis izdanja - izveštaj o izmenama i dopunama novog izdanja softvera
- Skraćen referentni pregled - pomoćni priručnik za brzo podsećanje

- Uputstvo za administraciju - tehnički priručnik sa objašnjenjima o mogućim naprednim prilagođavanjima sistema specifičnim okolnostima

387. Šta obuhvata tehnička (sistemska) dokumentacija softvera?

- Vizija - objašnjava ciljeve sistema
- Specifikacija i analiza zahteva - detaljan opis zahteva ugovorenih između zainteresovanih strana (naručioci, projektanti, izvođači, korisnici, klijenti)
- Specifikacija ili projekat - kako se sistem deli na celine, uloga komponenti sistema i sl.
- Opis implementacije - kako se pojedini elementi sistema modeliraju na konkretnim programskim jezicima, opis značajnih algoritama, specifikacija komunikacije
- Plan testiranja softvera - opis protokola testiranja softvera
- Plan testiranja prihvatljivosti - koje testove softver mora da prođe da bi bio prihvaćen
- Rečnik podataka - rečnik termina i opis osnovnih vrsta podataka koji se koriste u sistemu

388. Kakav je odnos agilnog razvoja softvera prema pisanju dokumentacije? Koji vidovi dokumentacije se podstiču a koji ne?

Agilni razvoj softvera podstiče specifičan pristup pravljenju i održavanju dokumentacije. Naime, velika količina dokumentacije često može doneti više problema nego koristi - razvojna dokumentacija je često neažurna (a neažurna informacija je dezinformacija) i cena njenog održavanja je visoka. Najvažnije je upamtiti da dokumentacija nije cilj, već sredstvo za ostvarivanje komunikacije. U agilnom razvoju važi da je najbolji vid dokumentacije dobar programski kod (štaviše, komentari u okviru koda omogućavaju automatsko pravljenje ažurne dokumentacije). Ostali vidovi dokumentacije koji se koriste su specifikacije (površno, u vidu korisničkih celina), modeli (mada često postaju neažurni tokom vremena) i dokumenti (uglavnom na osnovu modela, jako su skupi za održavanje, pa se retko prave).

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

Alati za unutrašnje dokumentovanje programskog koda omogućavaju programerima da direktno u kodu beleže informacije o funkcijama, klasama, metodama i drugim komponentama. Ovi alati automatski generišu dokumentaciju na osnovu ovih beleški, što olakšava razumevanje i održavanje koda. Potrebni su jer poboljšavaju čitljivost koda, omogućavaju lakše održavanje i efikasniju saradnju u timu. Razlike u odnosu na spoljašnju dokumentaciju:

- Unutrašnja dokumentacija se nalazi unutar samog koda, dok je spoljašnja dokumentacija smeštena u odvojenim datotekama ili priručnicima.
- Unutrašnja dokumentacija se često ažurira zajedno sa kodom, što smanjuje rizik od zastarelih informacija. Spoljašnja dokumentacija može zahtevati dodatne napore da bi ostala sinhronizovana sa promenama u kodu.
- Unutrašnja dokumentacija je prvenstveno namenjena programerima za razumevanje i održavanje koda, dok je spoljašnja dokumentacija često usmerena ka krajnjim korisnicima ili drugim zainteresovanim stranama koje možda nisu tehnički potkovane.

390. Šta je Doxygen? Šta omogućava? Navesti primere anotacije koda.

Doxygen je alat za automatsko generisanje programskog koda. On omogućava generisanje poprilično detaljne dokumentacije u raznim formatima (HTML, LaTeX, RTF, PDF, ...) na osnovu komentarima anotiranih izvornih fajlova sa kodom. Dokumentacija, između ostalog, uključuje i grafove zavisnosti, dijagrame nasleđivanja i dijagrame saradnje. Primer anotacije u C++ stilu (bar dva reda):

```
///
/// ... tekst ...
///
```

Primer anotacije u JavaDoc stilu:

```
/**
... tekst ...
*/
```

391. Šta je optimizacija softvera?

Optimizacija softvera je proces raspoređivanja opterećenja po resursima u skladu sa potrebama i mogućnostima. Razlog za ovu definiciju je sledeći - u procesu optimizacije menjamo strukturu programa radi postizanja manjeg zauzeća nekog resursa, npr. procesorskog vremena, upotrebljene radne memorije i slično, međutim, ušteda na jednom resursu najčešće povećava opterećenje nekog drugog resursa.

392. Koje su informacije neophodne za uspešnu optimizaciju?

- Razumevanje problema i rešenja (zadatak, algoritmi, implementacija)
- Razumevanje ograničenja (poslovni zahtevi, arhitektura računara i procesora)
- Poznavanje alata (programski jezik, prevodilac, asembler i mašinski jezik, alati za merenje performansi)

393. Objasniti "optimizaciju unapred". Dobre i loše strane?

Optimizacija unapred podrazumeva kontinualno staranje o performansama tokom čitavog trajanja razvoja projekta. Ovaj pristup ima smisla ako već u toku razvoja znamo koji delovi moraju biti optimizovani, ali u opštem slučaju sa sobom povlači potencijalne probleme. Na primer, moramo biti sigurni da znamo koji delovi moraju biti optimizovani, kao i kritične granice performansi koje se ne smeju probiti. Ovaj pristup je najproblematičniji ako deo koda koji smo optimizovali na kraju mora da se menja ili da se potpuno izbaci - mnogo vremena je bespovratno već utrošeno na optimizaciju. Optimizacija unapred sa sobom skoro uvek povlači i težinu u održavanju softvera.

394. Objasniti "optimizaciju unazad". Dobre i loše strane?

Optimizacija unazad podrazumeva da se postupak optimizacije vrši tek nakon završenog razvoja. Potrebno je meriti performanse i u skladu sa tim merenjima planirati optimizacije, ukoliko su one uopšte potrebne. Potencijalni problem sa ovim pristupom je da algoritam koji se koristi ne može da se optimizuje, ili, još gore, implementirana arhitektura softvera to ne omogućava, u kom slučaju bi morale da se prave velike izmene.

395. Kakva je suštinska razlika između optimizacije unapred i unazad? Kada je bolje primeniti koju od njih?

Suštinska razlika je da kod optimizacije unapred kontinualno vodimo računa o performansama komponenti za vreme razvoja, dok u optimizaciji unazad optimizujemo već implementirane komponente po potrebi. U većini slučajeva je bolje primenjivati optimizaciju unazad, ali postoje slučajevi kada je bolja optimizacija unapred, kao npr. kada znamo još od početka da će performanse biti od suštinskog značaja (npr. razvijamo biblioteku za obradu slika).

396. Koji osnovni problem proizvodi primena optimizacije u agilnom razvoju softvera? Kako se prevazilazi?

Osnovni problem nastaje usled prevremene optimizacije, tj. optimizovanja pre nego što znamo šta i koliko je potrebno optimizovati. Ovo se sukobljava sa principom agilnog razvoja "Neće biti potrebno". Da bi se problemi izbegli, vodimo se sa "Optimizuj kasnije", što je u suštini primena principa "Neće biti potrebno". Ovime će se uštedeti vreme i olakšati održavanje koda.

397. Kako se dele tehnike optimizacije? Navesti nekoliko primera.

Dele se na opšte tehnike, tj. tehnike koje se mogu primeniti nezavisno (uglavnom) od programskog jezika i na specifične tehnike, tj. tehnike koje se uglavnom odnose na konkretan (ili manji broj njih) programski jezik. Neki

primeri opštih tehnika optimizacije su odbacivanje nepotrebne preciznosti i upotreba umetnutih funkcija, a primeri specifičnih tehnika u C++-u su upotreba funkcija standardne biblioteke tamo gde je moguće i upotreba referenci.

398. Navesti bar 7 opštih tehnika optimizacije koda.

- Odbacivanje nepotrebne preciznosti
- Upotreba umetnutih funkcija i metoda
- Integracija petlji
- Izmeštanje invarijanti van petlje
- Razmotavanje petlji
- Upotreba tablica unapred izračunatih vrednosti
- Eliminacija grananja i petlji
- Zamenjivanje dinamičkog uslova statičkim
- Snižavanje složenosti operacije
- Snižavanje složenosti algoritma
- Pisanje zatvorenih funkcija
- Smanjivanje broja argumenata funkcija
- Izbegavanje globalnih promenljivih
- Redosled proveravanja uslova
- Izbor rešenja prema najčešćem slučaju

399. Objasniti tehnike optimizacije "odbacivanje nepotrebne preciznosti" i "tablice unapred izračunatih vrednosti".

Pod odbacivanjem nepotrebne preciznosti misli se na preciznost brojeva u pokretnom zarezu (npr. upotreba float umesto double) i smanjivanje opsega celih brojeva (int umesto long int i slično). Ukoliko dodatna preciznost nije potrebna, ovo je dobar kandidat za optimizaciju jer može smanjiti vreme izvršavanja i za 50%, Naročito ako je dužina podataka inicijalno veća od dužine procesorske reči ili širine magistrale podataka.

Upotreba tablica unapred izračunatih vrednosti je pogodan vid optimizacije ako se neke funkcije izračunavaju samo za ograničen broj argumenata, a inače sprovode složena izračunavanja. Umesto izračunavanja, samo konsultujemo tablicu i dohvatamo rezultat iz nje.

400. Objasniti tehnike optimizacije "integracija petlji", "izmeštanje invarijanti izvan petlje" i "razmotavanje petlji".

Integracija petlji podrazumeva spajanje više uzastopnih petlji u jednu sa složenijim korakom.Ovime će se uštedeti na dodatnoj ceni proveravanja uslova u petlji i koraka inkrementiranja, ali se čitljivost koda može dosta oštetiti (svaka petlja bi trebalo da radi tačno jedan posao). Primer bi bio računanje maksimuma, minimuma i proseka niza u jednoj petlji umesto u tri odvojene.

Izmeštanje invarijanti izvan petlje podrazumeva izmeštanje svega što se može izračunati van petlje, van nje i čuvanje rezultata u pomoćnoj promenljivoj. Primer bi bio čuvanje dužine niske u pomoćnoj promenljivoj umesto uslova u petlji `i < str.size()`.

Razmotavanje petlji je proces smanjivanja broja ponavljanja petlje uz povećanu složenost koraka petlje. Može i imati kontraefekat zbog otežanog keširanja.

401. Objasniti tehnike optimizacije "smanjiti broj argumenata funkcije" i "izbegavati globalne promenljive".

Ukoliko funkcija ima manji broj argumenata, prevodilac potencijalno može upotrebiti registre za prenos argumenata umesto steka. Najkorisnije je ako se ova funkcija često poziva. Jedan od "prljavijih" načina da se ova

optimizacija izvede je čuvanje argumenata u okviru strukture koja će se u funkciju prenositi preko adrese (mada onda imamo indirekciju za pristup argumentima).

Globalne promenljive, za razliku od lokalnih, se najčešće ne mogu optimizovati njihovim zapisivanjem u registre, jer te registre može koristiti još neko (neki potprogram npr.), a globalna promenljiva mora ostati dostupna, pa je poželjno izbegavati ih.

402. Objasniti tehnike optimizacije "upotreba umetnutih funkcija" i "eliminacija grananja i petlji".

Ukoliko je funkcije umetnuta (inline), onda se ne pravi novi stek okvir, već se sve vrši u trenutnom okviru. Naročito je pogodno za jednostavne funkcije koje se često pozivaju (npr. swap). Preterana upotreba može obesmisliti koncept.

Eliminacija grananja i petlji podrazumeva zamenu petlji i grananja sa složenijim izračunavanjem. Na primer, sumu prvih n celih brojeva umesto u petlji računamo formulom $\frac{n(n+1)}{2}$.

403. Objasniti tehnike optimizacije "zamenjivanje dinamičkog uslova statičkim" i "snižavanje složenosti operacije".

U slučaju zamene dinamičkog uslova statičim, ukoliko opseg broja ponavljanja neke operacije nije fiksan, a provera za zaustavljanje nije jeftina, može biti efikasnije uraditi posao za pun obim i onda izvršiti potrebnu proveru.

Složenost nekih operacija se može smanjiti, tj. one se mogu zameniti efikasnijim operacijama. Primer toga bi bila zamena deljenja sa stepenom dvojke sa bitovskim šiftovanjem u desno. Međutim, prevodioci su poprilično dobri u sprovođenju ovakvih operacija, pa se uglavnom ne preporučuje njihovo eksplicitno korišćenje u kodu.

404. Objasniti tehnike optimizacije "redosled proveravanja uslova" i "izbor rešenja prema najčešćem slučaju".

Redosled provera može biti važan jer u zavisnosti od njega možemo brže prihvatiti ili odbiti neko rešenje. Predlaže se da redosled bude takav da se što više smanji prosečan broj provera (ili, u drugoj varijanti, da se najskuplje provere ostave za kraj, tako da se one najređe izvršavaju).

Dosta algoritama radi efikasnije za određene instance problema, a manje efikasno za druge. Iz tog razloga ponekad je pogodno birati algoritam prema najčešćem slučaju koji se dešava. Primer ovoga je taj što bubble-sort radi efikasnije od quick-sort-a za veoma male nizove.

405. Koje su najčešće greške pri optimizaciji? Objasniti.

- Pogrešno pretpostavljanje da je jedno rešenje efikasnije od drugog - ovo je uvek potrebno i proveriti.
- Mišljenje da je manji kod uvek optimizovaniji - ovo nije uvek slučaj, iza kraćeg koda se možda krije teže izračunavanje.
- Optimizovanje tokom inicijalnog kodiranja - najčešći problem, prvo je potrebno napisati ispravan kod i testove, pa tek onda optimizovati.
- Posveiavanje više pažnje performansama nego korektnosti - koliko god da je kod optimizovan, on je beskoristan ako nije korektan.

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

- Korišćenje standardne biblioteke - ukoliko za nešto već postoji rešenje u standardnoj biblioteci, treba ga koristiti, jer veoma je teško (skoro i nemoguće) napraviti efikasnije rešenje od toga.
- Upotrebljavati reference umesto pokazivača - dobija se jednako efikasan, ali čistiji kod.
- Odložena inicijalizacija objekata - nekada nam nije neophodan komplektno inicijalizovan objekat za neku operaciju, pa je moguće da se umesto pune inicijalizacije pri konstrukciji izvede samo priprema za inicijalizaciju, a da se ona zaista uradi samo kada to bude bilo stvarno potrebno.
- Izbacivanje rukovanja izuzecima iz delova koda koji se optimizuju - obrada izuzetaka je poprilično neefikasna.

407. Šta su "optimizacije u hodu"? Navesti primere.

Optimizacije u hodu su vid optimizacija unapred koje je čak i poželjno raditi, tj. koristiti ih još za vreme kodiranja, ali, naravno, to treba raditi oprezno. Neki primeri su prenošenje objekata po referenci umesto po vrednosti (osim sasvim malih objekata), deklarisanje privremenih promenljivih što dublje u kodu (da se ne prave i ne zauzimaju prostor ako to nije potrebno), upotreba konstrukcije (inicijalizacije) objekta umesto dodeljivanja (da bi se izbeglo nepotrebno kopiranje), upotreba liste inicijalizacije članova, implementacija sopstvenih operatora alokacije (new) i dealokacije (delete), upotreba šablona funkcija i slično.

408. Šta su profajleri? Čemu služe? Šta pružaju programerima?

Profajleri su alati koji pomažu programeru u procesu optimizacije. Oni imaju mogućnost da za svaki potprogram (moguće čak i za blok ili naredbu programa) izračunačunaju određene statistike poput broja izvršavanja, ukupnog vremena izvršavanja i slično pri jednom pokretanju programa. Pored toga, mogu da mere i zauzeće memorije, upotrebu keš memorije i još puno toga. Neki primeri profajlera su GNU gprof i Valgrind.