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

* Nasledjivanje se predstavlja linijom koja se na strani parent klase zavrsava strelicom u obliku praznog trougla
* Implementacija interfejsa se prestavlja isto kao nasledjivanje samo sa isprekidanom linijom
* Agregacija se predtsavlja linijom koja na strani klase koja sadrzi delove zavrsava praznim rombom
* Kompozicija isto kao agregacija samo sa popunjenim rombom
* Asocijacija obicnom linijom bez ikakvih ismbola na krajevima
* Zavisnost obicnom isprekidanom linijom

**55. Objasniti kardinalnosti odnosa na dijagramu klasa – sta predstavljaju i kako se oznacavaju?**

Kardinalnost je broj objekata koji mogu da učestvuju u odnosu. Broj koji se navodi bliže jednoj klasi označava koliko objekata te klase može da učestvuje u datom odnosu sa jednim objektom druge klase

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

Na dijagramu klasa se retko predtsvljaju bas svi elementi da zapis nebi bio prenatrpan (posebno kod komplikovanih sistema), tako da je dijagram obicno fokusiran na jedan njihov aspekt:

* Dijagram modela domena (primarno se bavi strukturom objekata, tj na njemu nisu obelezeni metodi vec samo atributi, iako metodi mogu da postoje
* Dijagram implementacije (na njemu su prikazani samo svi metodi)
* Dijagram intefejsa klasa (ako su prikazani samo javni metodi)

**65. Objasniti odnose izmedju slucajeva upostrebe na diajgramu slucajeva upotrebe?**

Odnosi izmedju slucajeva upotrebe se oznacavaju isprekidanom linjom sa strelicom na kraju i stereotipom koji opisuje prirodu odnosa koji mogu biti:

* Include (slucaj obuhvata citav slucaj prema kom ide strelica
* Extends (slucaj od koga ide sterlica predstavlja moguce(opciono) prosirenje drugog slucaja (ali ne i njegov obavezan deo)

**66. Sta obuhvata opis jednog slucaja upotrebe?**

Moze sadrzati razne vrste informacija ali po pravilu mora da sadrzi bar:

* Naziv (sto kraci, mora dobro da ilustruje slucaj upotrebe)
* Aktere (ucesnici u slucaju upotrebe)
* Kratak opis (par recenica o sustini slucaja upotrebe
* Preduslove (moraju biti ispunjeni da bi akteri mogli da zapocnu slucaj up.)
* Postuslove (moraju da budu ispunjeni posle slucaja upotrebe)
* Opis toka slucaja upotrevbe (detaljan opis koraka koji se desvaju tokom slucaj pupotrebe, njihov redosled i sve moguce tokove)
* Opis posebnih slucajeva
* Dijagrame koji tacno opisuju slucaj upotrebe

**149. Navesti osnovne vrste polimorfizma**

Postoje 4 osnovne vrste polimorfizma:

* Hijerarhijski
* Parametarski
* Implicitni
* Ad-hoc polimorfizam

**150. Sta je hijerarhijski polimorfizam**

Hijerarhijski polimorfizam koristi upostavljene hijerarhijske relacije podtip i nadtip medju klasama kao sredstvo ostvarivanja polimorfizma. Koristi se mogucnost objektno orijentisanih jezika da programski kod koji moze da izvrsi objekat jedne kalse, mogu da izvrse i objekti svih njenih podklasa

**151. Sta je parametrarski polimorfizam**

Parametarski polimorfizam pociva na upotrebi tipskih promenljivih. One se koriste za simbolicno oznacavanje tipova vrednosti/promenljivih/izraza. Simbolicni tipovi se zamenjuju u konkretne tipove i program se prevodi u nepolimorfnom obliku (pri cemu je potrebno da na svim mestima u kodu ovi simboli mogu validno da se zamene). U C++u ovaj tipo polimorfizma se ostvaruje kroz sablone funkcija i sablone klasa.

**152. Sta je implicitni polimorfizam**

Implicitni polimorfizam predstavlja uopstenje parametarskog polimorfizma. Tipovi se uopste ne navode i ostavlja se prevodiocu da zakljuci o kojim tipovima je rec

**153. Sta je ad-hoc polimorfizam**

Overloading prestavlja tip ad-hoc polimorfizma. Dopusteno je da napisemo funkcije koje se zovu isto, ali da imaju razlicite povratne tipove/broj argumenata/tipove argumenata

**156. Sta su funkcionalni objekti - funkcionali**

Funkcionali su klase cije instance mogu da se pozivaju kao funkcije. U C++u ovo npr radimo tako sto za klasu predefinisemo operator() (zove se call operator)

**191. Koji su osnovni pristupi problemu debagovanja? Objasniti ukratko njihov odnos**

Razlikujemo tri pristupa:

* Neformalni metod
* Empirijski (naucni) metod
* Heuristicki metod

**192. Opisati empirijski (naucni) metod debagovanja**

Postupak lociranja priblema slican je uobicajnom naucnom metodu. Posmatramo problem, postavljamo hipotezu i predvidjamo ponasanje, eksperimentalno potvrdimo/oborimo nasu hipotezu i to ponavljamo dok ne otklonimo bag.

Uopsteno posmatrano, ovo je najbolji pristup debagovanju, ali nije uvek lako (npr postavljanje hipoteze je prepusteno iskustvu i intuiciji)

**193. Opisati heuristicki metod debagovanja**

Heristike su strategije za resavanje problema koje pocivaju na nekom prethodnom iskustvu. One ne moraju predstavljati optimalno resenje, ali cesto predstavljaju dovoljno dobro resenje. Heuristicko debagovanje pociva na primeni neke izabrane strategije koja za cilj ima da nas usmerava ka uzroku problema. Heuristike mogu biti razne, od kategorizacije problema do pitanja nekog drugog za pomoc

**205. Navesti osnovne unutrasnje tehnike i alate za debagovanje**

Unutrasnje tehnike i alati su sve ono sto ugradjujemo u sam kod radi lakse detekcije i otklanjanja bagova. To su npr: assertovi, trace-ovi, komentari, unit testovi, interni opisi stanja ...

**210. Navesti osnovne spoljasnje tehnike i alate za debagovanje**

Spoljasnji alati su razliciti alati koji se koriste tokom razvijanja i debagovanja programa . To su npr: debager, alati za pracenje verzije koda, alati za automatizovanje pravljenja dokumentacije itd...

**222. Objasniti odnos apstrakcije i dekompozicije u projektovanju softvera**

Apstrakcija i dekompozicija idu razlicim smerom. Apstrakcijom uopstavamo problem, tj njegovo resenje, dok dekompozicijom tokom vremena preciziramo resenje njegovim razlaganjem na manje celine

**224. Navesti kljucne principe OO dizajna**

Ima ih 5 (cesto se zovu akronimom SOLID):

* Princip jedinstvene odgovornosti (SRP - single responsibility principle)
* Princip otvorenosti i zatvorenosti (OCP – open-closed principle
* Princip zamenljivosti (LSP – Liskov subsitution principle)
* Princip razdvajanja interfejsa (ISP – interface segregation principle)
* Princip invernze zavisnosti (DIP – dependency inversion principle)

**225. Objasniti princip jedinstvene odgovornosti**

Kaze nam da jedna klasa/modul/funkcija treba da ima jednu namenu i nista vise od toga

**226. Objasniti princip otvorenosti i zatvorenosti**

Kaze nam da svaki modul/softverska celina mora da bude u isto vreme otvorena za prosirivanje (npr da moze da se nasledi), i zatvorena za svaku vrstu unutrasnjeg menjanja (misli se na menjanje od strane nekog korisnika, ne od strane programera koji su ga pisali)

**227. Objasniti princip zamenljivosti**

Kaze nam da podtipovi morajui da mogu da se upotrebe umesto njihovih osnovnih tipova, tj jako je lose uvoditi nova ponasanja duboko u hijerarhiji, tj trudimo se da podtipovi implementiraju interfejs koji nudi bazna klasa, a ne da uvode neki skroz novi interfejs

**229. Objasniti princip razdvajanja interfejsa**

Interfejs mora potpuno da bude odvoje od implementacije. Zelimo da menjamo implementaciju u pozadini, ali ne zelimo da menjamo interfejs

**229. Objasniti princip inverzne zavisnosti**

Tokom projektovanja, nase razmisljanje ide tokom od nekih specificnosti do opsteg problema, ali u nasem softveru taj tok treba da bude obrnut, tj da specificne implementacije zavise od opsteg/apstraktnog resenja

**230. Navesti principe oblikovanja celina**

Dele se na principe grupisanja i principe razdvajanja.

Principi grupisanja su:

* Princip ekvivalentnosti izdanja i upotrebe (REP – Relese-reuse equivalence principle)
* Princip zajednicke zatvorenosti (CCP – common closure principle)
* Princip zajednicke upotrebe (CRP – common reuse principle)

Principi razdvajanja su:

* Princip stabilne zavisnosti (SDP – stable dependency principle)
* Princip stabilne apstrakcije (SAP – stable abstractions principle)
* Princip aciklicne zavisnoti (ADP – acyclic dependencies principle)

**231. Objasniti princip ekvivalentnosti izdanja i uptrebe**

Ideja je da celinu treba grupisati tako da ako je ona jedinica upotrebe onda je ona i jedinica proizvodnje, tj da pisemo kod tako da neko ko tu celinu treba da koristi ne mora da iz nje uzima samo neke delove nego da moze celu da je koristi

**232. Objasniti princip zajednicke zatvorenosti**

Delovi neke celine treba da budu zajedno i podjedanko zatvoreni, tj ako neko sme da menja nesto, onda mora da sme da menja i ostalo, i obrnuto, ako ne zelimo da dozvolimo menjanje neke klase u celini, onda zabranjujemo i menjanje ostalih.

**233. Objasniti princip zajednicke upotrebe**

Ako se koristi jedan deo celine, onda se koristi citava celina

**234. Objasniti princip stabilne zavisnosti**

Zavisnost komponenti jedna od druge treba da ide u istom smeru kao i porast stabilnosti, tj nije dobro da neki modul zavisi od nestabilnih modula

**235. Objasniti princip stabilne apstrakcije**

Slicno kao princip stabilne zavinosti. Zelimo da modul bude onoliko apstraktan koliko je stabilan. Drugi moduli bi trebalo da zavise od apstraktnih i stabilnih modula

**236. Objasniti princip aciklicnih zavisnosti**

Treba da izbegavamo situacije tipa: A zavisi od B, B zavisi od C, C zavisi od A, zato sto promene u ovakvim ciklicno zavisnim modulima mogu da budu veoma problematicne

**237. Objasniti princip razdvajanja odgovornosti**

Razlicite komponente treba da imaju razlicite odgovornosti. Znamo vec da jedan modul treba da ima jedinstvenu odgovornost, a ovaj princip dodaje da ta odgovornost ne treba da bude podeljena na vise mesta

**238. Objasniti princip minimalnog znanja**

Interfejse pravimo tako da budu potpuno odvojeni od enkapsulacije. Jedna komponenta ne mora da zna kako druga funkcionise vec samo da zna kako da je koristi

**239. Objasniti princip informisane odgovornosti**

Ovaj princip nam kaze da podaci za obavljanje nekog posla treba da budu tamo gde se taj posao obavlja, tj generalno nije dobro da jedna klasa mora stalno da komunicira sa drugom da bi dobila potrebne podatke

**240. Objasniti princip izbegavanja ponavljanja**

Ne zelimo da ponavljamo kod vise puta (ovo dolazi iz agilnog razvoja)

**241. Objasniti princip izbegavanja suvisnog posla**

Ne zelimo da pisemo kod koji je nepotreban (ovo dolazi iz agilnog razvoja)