1. Sta je klasa i sta je objekat? - *definicije*

Osnovni pojmovi oop su klasa i objekat i direktno su vezani za misao. Misao o bitnim karak. nekog predmeta jeste pojma ili koncept. Podela na klasne i individualne pojmove.

Postupak izbora konacnog broja relevantnih oznaka pojma u odnosu na dati domen problema se naziva softversko modelovanje, a taj dobijeni konacni skup se zove softverski model. Isti pojam se moze modelovati na vise razlicitih nacina. Odnos klasa-objekat je analogan odnosu tip-promenljiva. Objekat je instanca klase.

Klasa objekata je softverski model klasnog pojma. Objekat je softverski model individualnog pojma.

2. Kako se definise klasa i kako se kreira objekat?

Definicija sadrzi rec class, nakon cega se navodi ime klase, a onda unutar viticastih zagrada se navode clanovi klase i na kraju je znak ;. Ime klase pise se velikim slovom, dok su ostala slova mala. Polja i metode se pisu malim pocetnim slovom. Clanovi klase mogu biti podaci-clanovi, objekti-clanovi i fje-clanice(metode). Podaci-clanovi i objekti-clanovi se jednim imenom mogu zvati polja.

KLASA U C++:

```
class Naziv{
        private: polja
        public: metode
};
```

KLASA U JAVI:

```
public class Naziv{
        private polje
        public metoda
}
```

KLASA U C#:

```
public class Naziv{
        private polje
        public metoda }
```

KREIRANJE OBJEKTA C++:

```
Krug k1;
```

KREIRANJE OBJEKTA U JAVI:

```
Krug k1 = new Krug();
```

KREIRANJE OBJEKTA U C#:

```
Krug k1 = new Krug();
```

3. Prava pristupa?

C++: private, protected i public

JAVA: private, protected, public, friendly(default)

C#: private, protected, public, internal

Pravo pristupa private podrazumeva da se tom clanu klase moze pristupati samo iz unutrasnjosti klase. Pravo pristupa public podrazumeva da se tom clanu klase moze pristupati kako iz unutrasnjosti, tako i iz spoljasnjosti. Pravo pristupa protected - dostupan samo unutar sadržanog tipa ili potklasa. Pravo pristupa internal - dostupan samo unutar sadržanog sklopa ili prijateljskih sklopova. Podrazumevana dostupnost za neugnežđene tipove.

4. Sta je konstruktor?

Konstruktor je metoda koja kreira objekat i postavlja ga u pocetno stanje. Konstruktor ima identicno ime kao i klasa, moze i ne mora da ima ulazne parametre i nema povratnu vrednost.

Za postavljanje nove vrednosti polja, a time i promenu stanja objekta koristi se set metoda, koja ima parametar tipa koji odgovara tipu polja kojeg menja. Ocitavanje vrednosti polja se vrsi pomocu get metode koja je uvek tipa koji odgovara tipu tog polja. Metoda menja stanje objekta ukoliko menja vrednost bar jednog polja (set metoda) i tada je zovemo modifikatorom. Postoje metode koje ne menjaju stanje objekta i imaju oznaku const.

KONSTRUKTOR U C++:

```
Krug(){}
```

KONSTRUKTOR U JAVI:

```
public Krug(){}
```

KONSTRUKTOR U C#:

```
public Krug(){}
```

5. Default konstruktor, konstruktor bez parametara, konstruktor sa parametrima i konstruktor kopije?

Default konstruktor(ugradjeni konstruktor) je metoda koja vrsi kreiranje objekta, ali ne i njegovu inicijalizaciju. Postoji u svakoj klasi, pa ukoliko projektant nije napisao svoj konstruktor tada kreiranje klase vrsi ugradjeni konstruktor koji kreira objekat, ali ga ne postavlja u pocetno stanje. Konstruktor bez parametara kreira objekat i postavlja ga uvek u isto pocetno stanje koje je uvek definisano od strane projektanta klase. Konstruktor sa parametrima kreira objekat i postavlja ga u inicijalno stanje koje odredjuje korisnik klase. Postavlja objekat u pocetno stanje u kojem je vr. polja n jednaka vr ulaznog parametra nn koji odredi korisnik klase. Konstruktor kopije je metoda koja vrsi kreiranje objekta i njegovu inicijalizaciju kopiranjem sadrzaja drugog objekta iste klase. Ima parametar koji je lvalue referenca na objekat iste klase. Postavlja objekat u inic. stanje u kojem je vr. polja n jednaka vrednosti polja n unutar objekta ref.

6. Šta je destruktor ? - *definicija i primer C++, Java i C#*

Destruktor je metoda koja unistava objekat i poziva se automatski u trenutku kada je unistenje neophodno(npr. na kraju slobodne fje je potrebno unistiti lokalne objekte). Nema ulazne parametre, ima identicno ime kao i klasa i ispred obaveza znak ~.

DESTRUKTOR U C++:

```
~Krug(){}
```

DESTRUKTOR U JAVI:

```
NE POSTOJI
```

DESTRUKTOR U C#:

```
~Krug(){} ne koristi se
```

7. Šta je statičko polje ? - definicija i primer C++, Java i C#

Unutar klase se mogu deklarisati i tzv. zajednicki(staticki) clanovi tako sto se ispred njihove deklaracije navede rezervisana rec static.

Staticko polje se inicijalizuje na pocetku programa i ono je zajednicko za sve objekte date klase. To znaci da u svakom trenutku vrednost tog polja je jednaka za sve objekte date klase.

Staticka metoda je metoda na nivou klase i za njeno pozivanje nije neophodno da postoji objekat. Staticka metoda moze da koristi iskljucivo staticka polja **.

Preklapanje operatora ? - radili samo iz C++ i Jave **valjda ne dolazi** ima i u priručniku za C# strana 140 (napisala sam samo c++) . Preklapanje operatora je mehanizam koji omogucava da se za vecinu standardnih tipova definise njihovo ponasanje za slucaj da su operandi klasnih tipova.

8. Klijentske veze - vrste polimorfizma (*asocijacija, agregacija, kompozicija, veze korišćenja, nasleđivanje i veze zavisnosti***) ?** - *definicije*

Veze izmedju klasa modeluju odnose medju pojmovima, a mogu se klasifikovati u 3 vrste:

1. Klijentske veze u kojima klase-klijenti koriste usluge klasa-servera, i tu spadaju:

a) ASOCIJACIJA - koja obuhvata sirok spektar logickih relacija izmedju semanticki nezavisnih klasa.

b) AGREGACIJA - koja modeluje odnos celina-deo.

c) KOMPOZICIJA - takodje modeluje odnos celina-deo, ali uz uslov zavisnosti dela od celine.

d) VEZE KORISCENJA

2. Nasledjivanje - kreiranje potklase na osnovu preuzimanja sadrzaja natklase, uz mogucnost modifikacije preuzetih i dodavanja novih sadrzaja.

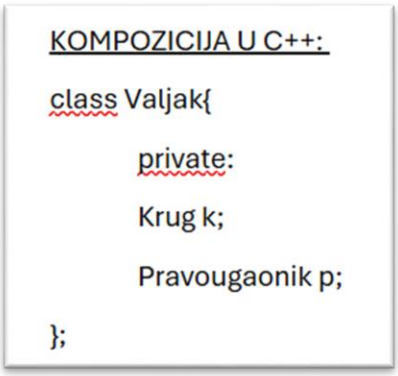
3. Veze zavisnosti

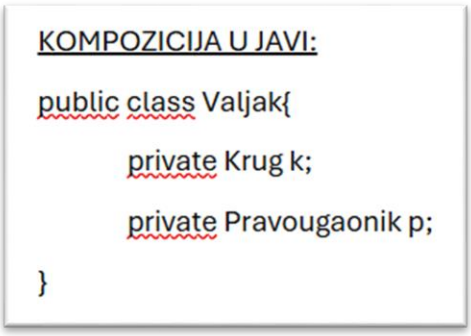
9. Šta je kompozicija ? - *definicija i primer C++, Java i C#*

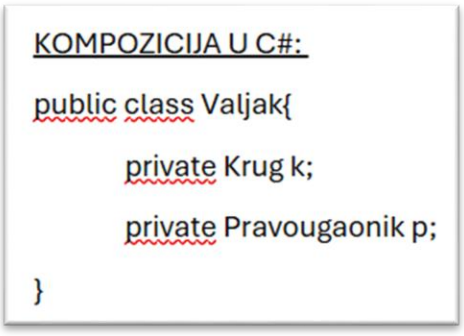
Kompozicija je klijentska veza izmedju klasa, za koju vazi to da vlasnik poseduje komponentu, pri cemu je zivotni vek komponente sadrzan u zivotnom veku vlasnika. Komponenta ne moze postaojati pre kreiranja i posle unistenja vlasnika.

Klasa koja je celina se zove vlasnik, a klasa koje je deo je komponenta.

Definicija: AKO SVAKI OBJEKAT KLASE A POSEDUJE BAR JEDAN OBJEKAT KLASE B, PRI CEMU STVARANJE I UNISTAVANJE DATOG OBJEKTA KLASE B ZAVISI OD STVARANJA I UNISTAVANJA KLASE A, ONDA SE KAZE DA IZMEDJU KLASA A I B POSTOJI VEZA KOMPOZICIJA.







10. Šta je nasleđivanje ? - definicija i primer C++, Java i C#

Nasledjivanje je veza izmedju klasa koja podrazumeva preuzimanje sadrzaja osnovnih klasa, tzv. roditelja i na taj nacin, uz mogucnost modifikacije preuzetog sadrzaja i dodavanja novovg dobija se izvedena klasa, tzv. potomak.

NASLEDJIVANJE U C++:

```
class Naziv: public Naziv2{}
```

NASLEDJIVANJE U JAVI:

```
public class Naziv extends Naziv2{}
```

NASLEDJIVANJE U C#:

```
public class Naziv: Naziv2{}
```

11. Vrste nasleđivanja (private, protected I public) ? *–U kom jeziku koji je podrazumevan I koji postoje :*

PRAVA PRISTUPA C++:

private,protected, public

PRAVA PRISTUPA JAVA:

private,protected,public,friendly(default)

PRAVA PRISTUPA C#:

private,protected,public,internal

Private clanovima natklase moze se direktno pristupati u potklasi samo preko metoda koje su ili protected ili public u natklasi. Clanu klase koji je **private** moze se direktno pristupati samo iz metoda te klase i njenih prijateljskih fja. Clanu klase koji je **protected** moze se direktno pristupati iz metoda te klase, njenih prijateljskih fja i metoda njenih potklasa.

12. Statičke i virtualne metode ? - definicije, razlike i primer C++, Java i C#

Vitruelna metoda je sopstvena metoda osnovne klase, koja moze da se zameni istoimenom metodom odg izvedene klase kada se pozove na objekat koji pripada izvedenoj klasi.

Staticka metoda je metoda koja je povezana sa klasom i nema sposobnost da radi na instanci klase.

Virtualne metode razlikuju se od statickih po tome sto se na mestu poziva u prevedenom kodu ne nalazi direktan skok na njihov pocetak.

C++:

```
virtual int metoda(){}
```

JAVA:

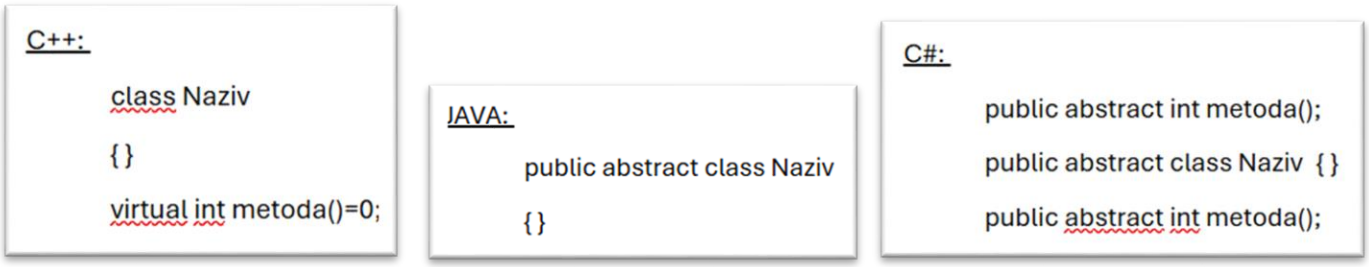
svaka metoda koja nije static je virtualna

C#:

```
public virtual int metoda(){}
```

13. Apstraktna klasa ? - definicija i primer C++, Java i C#

Klasa koja ima bar jednu apstraktnu metodu zove se apstraktna klasa. Ne moze se napraviti objekat aps. klase. Apstraktna metoda je virtuelna metoda koja nema telo.



14. Private i višestruko nasleđivanje ? - *objasniti i kod kojih jezika postoji i kod kojih ne*

Private:

C++ omogucava da se prilikom nasledjivanja navede nacin izvodjenja koji moze biti private, protectec ili public. Ukoliko je nacin izvodjenja private tada nasledjeni sadrzaj postaje private. Preuzeta polja i metode ce u potklasi imati pravo pristupa private, odnosno moci cemo da ih koristimo samo unutar potklase.

U Javi je to izbaceno,tj podrazumeva se da je nacin izvodjenja uvek public. Omogucava da se neka klasa moze izvesti iz bilo koje druge klase.

Visestruko nasledjivanje: Situacija u kojoj se neka klasa izvodi iz dve ili vise natklasa. Postoji u C++, u Javi je izbaceno.

15. Generičke klase ? - objasniti i primer C++, Java i C#

U C++ postoji mehanizam pomocu kog mozemo napisati sablon kojim opisujemo opsti slucaj (bez upotrebe konkretnih tipova). Klasa koja je napisana pomocu sablona zove se genericka klasa. Kada se sablonu navedu konkretni tipovi dobijamo konkretne klase.

16. Rukovanje izuzecima ? - primer C++, Java i C#

Izuzetak je namerno izazvan dogadjaj cija je svrha predupredjivanje otkaza i koji ostavlja program u zatecenom stanju. Prijavljivanje izuzetka se vrsi naredbom throw. Prihvatanje u obrada izuzetaka se vrse u try-catch bloku.

17. Unistavanje objekata u Javi ? - *Garbage Collection*

U Javi ne postoje destruktori. Svu potrebnu dealokaciju memorije u Javi obavlja GC proces. Ovaj proces radi nezavisno od pokrenutog programa, u smislu da sam odlučuje u kom trenutku će iz memorije osloboditi koji nedostupni objekat.U trenutku dealokacije podrazumeva se da je Java objekat oslobodio ostale resurse koje je koristio (otvorene datoteke, mrežne konekcije, itd).Ukoliko je neophodno obaviti neku operaciju neposredno pre nego što GC uništi objekat, ta operacija se može implementirati u okviru specijalne metode finalize().

18. Apstraktna klasa i Interface ? - razlike i u kojim jezicima koji postoje

Apstraktna klasa i interfejs se koriste za postizanje apstrakcije gdje možemo deklarisati instancirati. apstraktne metode. Apstraktna klasa i interfejs se ne mogu Interfejs: JAVA, C# ABSTRACT CLASS: C++, JAVA, C# Apstraktna klasa je klasa koja sadrzi aps. metode - metode bez implementacije Interfejs u javi je mehanizam za postizanje potpune apstrakcije.

- Aps. klasa moze imati aps. i neaps. metode, a interfejs moze imati samo aps.
- Aps. klasa ne podrzava visestruko nasledjivanje, interfejs podrzava.
- Apstraktna klasa može imati static metode, main metod i konstruktor, interfejs ne moze.
- Apstraktna klasa može obezbijediti implementaciju interfejsa, obrnuto ne moze.

Kod aps. klase kljucna rec je abstract, kod interfejsa je interface :

C++:

ne postoji

JAVA:

public interface naziv

{

neka metoda bez tela

}

C#:

interface naziv

{

metoda bez tela(definicija)

}

19. Imenski prostori u C# i paketi u Javi ? - *objasniti i navesti razlike*

- C#

Imenski prostor predstavlja skup povezanih klasa određen imenom. Imenski prostor eliminiše potrebu za posebnim imenima koja po pravilu znače uvođenje raznih prefiksa po kojima će se imena razlikovati od bilo kog drugog. Preporučuje se da se svaka klasa definiše u određenom imenskom prostoru. Za izradu imenskog prostora koristi se rezervisana reč namespace iza koje sledi ime.

- JAVA

U Javi se definišu PROSTORI IMENA u kojima se smeštaju METODE i PODACI koji se koriste u odgovarajućoj klasi.Ovo omogućava da više metode mogu IMATI ISTA IMENA ako pripadaju RAZLIČITIM imenskim prostorima. Prostori imena u Javi je HIJERARHIJSKI organizovan slično strukturi folderaodvojenih tačkom. Zbog toga, da bi se definisalo KOJA METODA se tačno poziva, MORA se navesti PROSTOR IMENA. U Javi se PROSTOR IMENA naziva PAKET.

20. Šta je this i za šta se koristi ? - Java i C#

- JAVA

Specijalna referenca na objekat kojem se upravo pristupa. Moze se koristiti unutar nestatickih metoda. Nacin da se prosledi referenca na tekuci objekat.

- C#

Referenca this upućuje na samu instancu. Zahvaljujući referenci this razlikuju se lokalna promenljiva ili parametar od polja. Referenca this je upotrebljiva samo unutar nestatičkih članova klase ili

strukture. Klasa ili struktura mogu da preklapaju konstruktore. Jedno preklapanje može da poziva drugo, pomoću rezervisane reči this.

21. Šta su Svojstva (properties) ? - C#

Svojstva spolja izgledaju kao polja, ali sadrze unutrasnju logiku, kao metode. Deklarise se kao polje, ali mu se dodaje blok get-set, a to su metode za pristupanje svojstvima. Svojstvo može samo da se čita ukoliko je zadata samo metoda get, a samo da se upisuje ako je zadata samo metoda set.

22. Klasa Object ? - *objasniti šta je Java i C#*

- JAVA

Klasa Object je roditeljska klasa svih klasa u javi po defaultu. Drugim riječima, ona je najviša klasa u javi. Klasa Object je pogodna ako želimo da uputimo na bilo koji objekt čiji tip ne znamo.

- C#

object (System.Object) predstavlja vrhovnu osnovnu klasu za sve tipove. Svaki tip se može implicitno konvertovati naviše u tip object. object je referentni tip, zahvaljujući tome što je klasa. Uprkos tome, vrednosni tipovi, kao što je int, takođe mogu da se konvertuju u tip object ili iz njega.

Da bi to omogućio, CLR mora da obavi specijalan posao kako bi se premostile razlike između vrednosnih i referentnih tipova. Taj proces se zove pakovanje (engl. boxing) i raspakivanje.

23. Modelovanje softvera ?

U softverskom dizajnu i razvoju baziranom na modelu, modelovanje se koristi kao osnovni deo procesa razvoja softvera. Modeli se kreiraju i analiziraju pre implementacije samog sistema, i služe da usmere implementaciju koja će uslediti. Sa sve većim brojem notacija i metoda za objektno-orjentisanu analizu i dizajn softverskih aplikacija, stvorila se potreba za jedinstvenim jezikom za modelovanje. Kao rezultat, nastao je UML (Unified Modeling Language) kako bi ponudio standardizovani grafički jezik i notaciju za opis objektno-orjentisanih modela. UML model može biti nezavistan od platforme PIM (Platform-Independent Model), ili model namenjen specifičnoj platformi PSM (Platform Specific Model). Modelovanje softvera predstavlja opisivanje softverskog dizajna grafički, tekstualno, ili i grafički i tekstualno.

U razvoju aplikacija se koriste sledeći dijagrami:

Dijagram slučajeva korišćenja - predstavlja pristup za opisivanje funkcionalnih zahteva sistema. Funkcionalni zahtevi se opisuju pomoću učesnika, koji predstavljaju korisnike sistema, i slučajeva korišćenja. Slučaj korišćenja definiše sekvencu interakcija između jednog ili više učesnika i sistema (predstavljen kao crna kutija).

- Klasni dijagram
- Dijagram objekata
- Komunikacioni dijagram
- Sekvencijalni dijagram
- Dijagram stanja

Dijagram aktivnosti - predstavljaju UML dijagrame koji prikazuju tokove kontrole i sekvence koje se dešavaju tokom softverske aktivnosti. Dijagram aktivnosti prikazuje sekvence aktivnosti, čvorove odluke, skokove, pa čak i konkurentne aktivnosti. Ovi dijagrami se dosta primenjuju u modelovanju tokova aplikacije, npr. kod servisno-orjentisanih aplikacija.

Dijagram razmeštanja

Dijagram Klasa - se odnosi na statički strukturni pogled na problem, koji je nepromenljiv u odnosu na vreme. Preciznije, statički model definiše klase u sistemu, atribute klasa, relacije između klasa i operacije koje poseduje svaka klasa. Asocijacija definiše relaciju između dve ili više klasa i označava statičku, strukturalnu relaciju. Npr. Zaposleni Radi u Odeljenju, gde su Zaposleni i Odeljenje klase a Radi u predstavlja asocijaciju. Višestrukost asocijacije može biti sledeća Jedan-prema-jedan asocijacija, Jedan-prema-više asocijacija, Numerički navedena asocijacija, Opciona asocijacija i Više-ka-više asocijacija. Kompozicija i agregacija se odnose na klase koje su sačinjene od drugih klasa. Kompozicija i agregacija predstavljaju specijalne forme relacija u kojima su klase povezane pomoću celina/deo veze.

24. Arhitektura sistema ?

Difinisanje arhitekture softverske aplikacije predstavlja proces kreiranja softvera koji zadovoljava sve tehničke i funkcionalne zahteve, dok u isto vreme optimizuje kvalitativne osobine kao što su performanse, bezbednost i mogućnost jednostavnog održavanja. Softverska arhitektura uključuje niz odluka zasnovanih na mnogobrojnim faktorima, a svaka od ovih odluka može imati značajan uticaj na kvalitet, performanse, održavanje i opšti uspeh cele aplikacije. Arhitektura sistema predstavlja skelet datog sistema. Za arhitekturu sistema su odgovorne arhitekte.

Ciljevi arhitekture - Softverska arhitektura ima za cilj da kreira vezu između poslovnih zahteva i tehničkih zahteva razumevanjem slučajeva korišćenja, kao i pronalaženjem načina da se ti slučajevi korišćenja implementiraju u softver. Cilj arhitekture je da identifikuje zahteve koji utiču na strukturu aplikacije.

- Softverska arhitektura bi trebala da:
- Prikaže strukturu sistema ali da sakrije detalje implementacije
- Realizuje sve slučajeve korišćenja i scenarije
- Odgovori i na funkcionalne i na kvalitativne zahteve

Principi softverske arhitekture - Prilikom kreiranja arhitekture moramo pretpostaviti da će dizajn evoluirati tokom vremena i da ne možemo znati sve što je potrebno unapred kako bi u potpunosti kreirali arhitekturu sistema.

Treba razmotriti sledeće ključne principe kada se kreira arhitektura aplikacije:

Kreiraj da menjaš umesto da kreiraš da traje - traba razmotriti kako će se aplikacija vremenom menjati kako bi se što bezbolnije moglo odgovoriti na zahtevane promene.

Modeluj kako bi analizirao i smanjio rizik - treba koristiti alate za modelovanje kao što je UML (Unified Modeling Language).

Koristi modele i vizuelizaciju kao alate za komunikaciju i saradnju - efikasna komunikacija, kreiranje odluka i predviđanje izmena predstavljaju ključne faktore u cilju kreiranja dobre arhitekture. Treba koristiti modele, poglede i druga sredstva vizuelizacije kako bi se efikasno komuniciralo sa svim činiocima koji učestvuju u kreiranju softvera.

Identifikovanje ključnih inženjerskih odluka - treba koristiti sve dostupne informacije i znanja kako bi se razumele najčešće inženjerske odluke i uočile oblasti gde se greške najčešće prave. Treba investirati kako bi se na samom početku donele ispravne odluke i kako bi dizajn bio više fleksibilan i otporan na izmene.

Modeli u razvoju softvera :

Tradicionalna metodologija - Najpoznatija i najstarija metodologija je model vodopada. To je model u kom razvoj softvera produžava iz jedne faze u drugu u čisto sekvencijalnom smislu. U korak N+1 se premeštamo tek kada je korak N 100% uspešno završen.

Agilna metodologija - Iterativni razvoj je ciklični proces koji je razvijen kao odgovor na metod vodopada, a koji naglašava inkrementalno kreiranje softvera. Nakon početnih aktivnosti, projekat prolazi kroz niz iteracija koje uključuju analizu, kodiranje i testiranje. Svaka iteracija daje isporučivu, ali ipak nekompletnu verziju sistema. U svakoj iteraciji, tim unosi promene u dizajnu i dodaje nove funkcionalnosti sve dok se ne ispune svi zahtevi.

25. Dizajn principi i parametri ?

Jedna stvar je napisati kod koji radi. Nešto sasvim drugo je napisati dobar kod koji radi. Usvajanje stanovišta "pisanje dobrog koda koji radi" proističe iz sposobnosti da se sistem posmatra iz široke perspektive.

Dizajn šabloni (design patterns) predstavljaju apstraktne primere rešenja na visokom nivou. Njih treba posmatrati kao plan u rešavanju problema, a ne kao samo rešenje. Gotovo je nemoguće pronaći okvir (framework) koji će biti primenjen kako bi se kreirala cela aplikacija. Umesto toga, inženjeri vrše generalizaciju (uopštavanje) problema kako bi prepoznali patterne koje treba da primene. Dizajn paterni imaju za cilj ponovnu upotrebu postojećih rešenja.

3 grupe paterna :

Creational Patterns: odnose se na kreiranje objekata i referenciranje

Structural Patterns: odnose se na relacije između objekata i na to kako objekti međusobno deluju jedan na drugi u cilju kreiranja većih i kompleksnijih objekata

Behavioral Patterns: odnose se na komunikaciju između objekata, naročito u smislu odgovornosti i algoritama

Uobičajeni dizajn principi - Postoji veliki broj uobičajenih dizajn principa koji, kao i dizajn paterni, predstavljaju najbolju praksu i omogućavaju osnovu na kojoj se mogu kreirati profesionalni softveri koji su laki za održavanje. Neki od najpoznatijih principa su:

Keep It Simple Stupid (KISS) – veoma često softverska rešenja budu suviše komplikovana. Cilj KISS principa je da kod bude jednostavan i da se izbegne nepotrebna kompleksnost

Don't Repeat Yourselt (DRY) – ovaj princip nalaže da se izbegne bilo kakvo ponavljanje delova sistema. Ovo se postiže apstrakcijom onih delova koji su zajednički i njihovim smeštanjem na jednu lokaciju. Ovaj princip se ne bavi samo kodom, već bilo kojom logikom koja je duplirana u sistemu.

Tell, Don't Ask – ovaj princip je blisko povezan sa enkapsulacijom i dodelom odgovornosti odgovarajućim klasama. On govori da je potrebno reći objektima koja akcija treba da se izvrši, umesto da se postavlja pitanje o stanju objekta i da se onda pravi odluka koja akcija treba da se izvrši. Ovo pomaže da se uredi odgovornost i da se izbegne jaka veza između klasa.

You Ain't Gonna Need It (YAGNI) – ovaj princip govori da je potrebno uključiti samo funkcionalnosti koje su neophodne aplikaciji, a izbaciti sve ostaje funkcionalnosti za koje se misli da bi mogle zatrebati.

Separation of Concerns (SoC) – predstavlja proces razdvajanja dela softvera na zasebne karakteristike koje sadrže jedinstveno ponašanje, kao i na podatke koje mogu koristiti i druge klase. Proces razdvajanja programa po diskretnim odgovornostima značajno pospešuje ponovnu upotrebu koda, održavanje i testiranje.

S.o.l.i.d. dizajn principi - S.O.L.I.D. dizajn principi predstavljaju kolekciju najbolje prakse za objektno-orjentisan dizajn. Svi dizajn paterni koje su zapisali Gang of Four odgovaraju u određenoj formi ovim principima. Naziv S.O.L.I.D. dolazi iz prvih slova svakog principa:

Single Responsibility Principle (SRP)

Open-Close Principle (OCP)

Liskov Substitution Principle (LSP)

Interface Segregation Principle (ISP)

Dependency Inversion Principle (DIP)

26. Višeslojna arhitektura ?

Antipatern – SMART UI - ASP.NET web forme i Visual Studio čine kreiranje aplikacija veoma lakim i jednostavnim prevlačenjem kontrola na HTML dizajner. Ovi fajlovi u sebi sadrže sve obrade događaja, pristup podacima i poslovnu logiku. Problem kod ovog pristupa je to što su svi koncepti pomešani. Smart UI aplikacije se, međutim, ne moraju izbegavati po svaku cenu. One su dobre za izradu prototipova i za aplikacije koje će biti kratko korišćene.

Slojevita aplikacija - Rešenje za Smart UI antipatern je razdvajanje aplikacije u slojeve. Razdvajanje se može postići pomoću imenskih prostora, foldera ili odvojenih projekata.

Sa slike se može videti da korisnici pristupaju prezentacionom sloju. Taj sloj predstavlja izgled aplikacije i obično se implementira pomoću MVC paterna. Preko prezentacionog sloja se pristupa sloju servisa koji dalje obrađuje zahteve i komunicira sa ostatkom aplikacije.

Ukoliko neki spoljni sistem treba da koristi aplikaciju, on direktno komunicira sa slojem servisa koristeći sopstveni prezentacioni sloj.

Zašto koristiti slojeve - Sloj se obično shvata kao neka vrsta crne kutije sa interfejsom koji definiše ulaz i izlaz i malo ili nimalo zavisnosti ka drugim slojevima i fizičkim serverima. Glavne prednosti koje se dobijaju upotrebom slojeva su organizacione i obuhvataju ponovnu upotrebu i razdvajanje nadležnosti. Ponovna upotreba sloja omogućava da se jednom napisani sloj upotrebljava u okviru više poslovnih aplikacija. Razdvajanje nadležnosti predstavlja ideju da slojevi budu zatvoreni, ili drugim rečima, sa visokim stepenom kohezije i niskim stepenom sprege.

Kreiranje slojeva u praksi :

Oblast #1: FORMATIRANJE PODATAKA

Oblast #2: CRUD OPERACIJE

Oblast #3: UGRAĐENE PROCEDURE

27. MVC (Model-View-Controller) patern ?

Nastao je kao deo Smalltalk projekta u okviru kompanije Xerox PARC kao način organizacije u okviru ranih GUI aplikacija.

MVC zahteva razdvajanje odgovornosti jer su model i logika kontrolera razdvojeni od

korisničkog interfejsa. U okviru Web aplikacija, ovo znači da je HTML kod izdvojen od ostatkaaplikacije, što čini održavanje i testiranje jednostavnijim i lakšim.

Posmatrajući sa visokog nivoa apstrakcije, MVC patern razdvaja aplikaciju u tri glavnekomponente:

Model - predstavlja poslovne podatke koje će pogled da prikaže ili da modifikuje

Controller - vrši interakciju sa modelom na osnovu zahteva koje dobiaj preko browser-a , izvršava operacije nad modelom i bira odgovarajući view koji će biti prikazan

View - je pasivan i ne poseduje znanje o kontroleru. On jednostavno prikazuje podatke iz modela koje je dobio od controller-a

Asp.net implementacija mvc-a - U okviru MVC-a, kontroleri predstavljaju C# klase koje su izvedene iz klase Controllera. Svaka public metoda u klasi koja je izvedena iz klase Controller se naziva akciona metoda i može joj se pristupiti preko URL-a na način kako se odredi u okviru ASP.NET sistema za rutiranje.

Korisnik upućuje zahtev koji prvi prima i obrađuje controller. Controller vrši interakciju sa modelom na osnovu dobijenog zahteva i dobavlja podatke. On prikazuje odgovarajući pogled (view) i obezbeđuje ga potrebnim podacima koji treba da budu prikazani.

Ne postoje bilo kakva ograničenja kada je u pitanju implementacija domen modela. Možemokreirati model pomoću standardnih C# objekata i implementirati skladištenje pomoću bilo koje baze podataka, objektno-relacionog mapiranja, ili drugih alata koji su podržani od strane .NET-a. Visual

Studio sam kreia folder Models kao deo šablona za MVC projekat. Ovo je odgovarajuće za jednostavneprojekte, dok složenije aplikacija teže kreiranju domen modela u posebnom Visual Studio projektu.Pametni gradovi :

28. Šta je grad ?

Grad je relativno veliko i stalno urbano naselje u kojem većina stanovništva živi od industrije, trgovine i servisnih delatnosti za razliku od sela gde je većina ekonomskih aktivnosti zasnovana na poljoprivredi.

Gradovi su rasli i razvijali se noseći sa sobom svoje prednosti i mane. Prednosti su uključivale: smanjene troškove transporta, deljenje prirodnih resursa, neograničen pristup tržištu, razvoja i pogodnosti poput vodovoda i kanalizacije. Mane su uključivale: smanjenje stambenog prostora, porast kriminala svih vrsta, povećanu smrtnost, povećane troškove života i zagađenje.

29. Šta je urbanizacija ?

Urbanizacija je naziv kojim se označava prirodni ili mehanički prirast stanovništva u gradskim područjima, širenje gradskih područja, odnosno transformacija pretežno seoskih karakteristika nekog područja u gradsko.

Proces urbanizacije je započeo sa samim počecima civilizacije i stvaranja gradova, ali se intenzivirao tek nakon industrijske revolucije, odnosno korištenja novih tehnologija u poljoprivredi koje su smanjile potrebu za ljudskom radnom snagom, te ekspanzije uslužnog sektora u ekonomiji.

30. Šta je pametan grad ?

Pametni grad je urbano područje koje koristi različite vrste elektroničkih senzora za prikupljanje podataka kako bi se osigurale informacije potrebne za upravljanje imovinom i resursima.

To uključuje podatke prikupljene od građana, uređaja i imovine koja se obrađuje i analizira za praćenje i upravljanje prometnim i transportnim sustavima, elektranama, vodoopskrbnim mrežama, policijom, informacijskim sustavima, školama, knjižnicama, bolnicama i drugim zajednicama.

Primer pametnog grada : Dubaija, Southamptona, Amsterdama, Barcelone, Madrida, Stockholma.

31. IKT (Informacijske i Komunikacione Tehnologije) ?

Informacijska i komunikacijska tehnologija koristi se za poboljšanje kvalitete života u gradovima. Pametne aplikacije razvijene su za upravljanje gradskim tokovima i omogućavaju reakcije u realnom vremenu.

Metodologije na procenu visoke upotrebe IKT kroz implementaciju e-rešenja u gradskim upravama, a sve na osnovu kvantitativnog prikupljanja podataka iz postojećih baza podataka i uvođenje novih indikatora, danas predstavlja osnovu pristupa koncepta pametnog grada.

IKT se primenjuje : transport (upotreba GPS za što brži prenos), zdravstvo (elektronsko zakazivanje i dojava rezultata), edukaciju (online organizovana nastava, sastanci), upravljanje otpadom (GPS, automatizacija vozila) itd.

32. IoT (Internet of Things) ?

Pametni gradovi koriste takozvane Internet of things (IoT) uređaje, poput senzora, svetla i brojila za prikupljanje i analizu podataka. Gradovi zatim koriste te podatke kako bi poboljšali infrastrukturu, komunalne usluge, itd. IoT doprinosi smanjenju troškova i poboljšanju nekih radnih mesta.

Primeri :

Kopenhagen je počeo da koristi senzore za praćenje gradskog biciklističkog saobraćaja, koji pružaju značajne informacije za praćenje poboljšanja biciklističke rute u gradu.

San Diego je počeo da koristi kamere koje su ugrađenje u semafore, koje će da prate pešački saobraćaj i preusmere automobile tokom špica, kako bi se izbegle pešačke nesreće i smanjile gužve.

33. Big Data ?

Big data se odnosi na podatke koje generišu digitalne tehnologije kao što su mobilni telefoni, veb sajtovi, sateliti ili senzori.

Big data je često sekundarni proizvod našeg digitalnog ponašanja, zasnovan na digitalnim “mrvicama” koje ostavljamo kao trag kada interagujemo sa sistemima ili mašinama u svojim svakodnevnim životima.

Izvori podataka u osnovi big data mogu biti grupisani u podatke koje generišu ljudi, podataka zasnovanih na procesima, mašinski generisane podatke, kao i podatke sa medija.

Da li su podaci validni i tačni? Vrednost big data leži u njihovoj mogućnosti da generišu pouzdana i validna merenja.U svojstvu pasivnih podataka čija namena može biti promenjena, big data treba da bude validiran putem drugih izvora podataka, kao što su tradicionalniji izvori.

34. Cloud Computing ?

Cloud computing je pojam kojim se označava isporuka svih hostovanih usluga putem Interneta. Uz cloud computing, kompanije imaju mogućnost da po potrebi koriste resurse poput virtualnih mašina, skladištenog prostora na kojima se nalaze njihove aplikacije, platforme itd.

Vrste Cloud Computinga : Public Cloud (svima dostupan), Private Cloud (koriste samo neke kompanije) i Hybrid Cloud (kombinacija predhodna dva - razmena podataka između Public i Private Clouda).

Uticaj Cloud Computinga na pametne gradove je takođe značajan. Kompanije mogu da koriste velika skladišta podataka (Big Data) u koja će smestiti prikupljene podatke o gradu (policiji, medicini, edukaciji itd) i te podatke koristiti za unapređenje, razvoj itd.

35. AI (Artificial Intelligence) & ML (Machine Learning) ?

AI (Artificial Intelligence) - Veštačka inteligencija (AI), omogućava računarima da koriste ogromnu količinu podataka kako bi sami doneli odluku ili izvršili neku aktivnost. Primeri softveri za prepoznavanje lica, automobili kojima nije potreban vozač itd.

ML (Machine Learning) - Mašinsko učenje je oblast veštačke inteligencije koja se bavi izgradnjom računarskih sistema koji uče iz iskustva.

Podela mašinskog učenja :

Nadgledano učenje (Supervised Learning) - imamo skup podataka [Atribut, Vrednost] na osnovu tih podataka kreira se model koji će za nove ulazne podatke (atribute) kreirati korektne vrednosti. Neki algoritmi : Linear Regression, Logistic Regression, Support Vector Machines (SVMs), Neural networks itd.

Nenadgledano učenje (Unsupervised Learning) - imamo skup neoznačenih podataka i pokušavamo da nađemo neku zevu između njih, interesantne stvari o njima itd. Neki algoritmi : K Means Clustering, Isolation Forest, Apriori itd.

Polunadgledano učenje (Semisupervised Learning) - imamo skup podataka koji je delimično označen a delimično nije na osnovu njih treba kreirati model koji će nove podatke dodati u neku od grupa.

Učenje sa podsticajem (Reinforcement Learning) - Primer : Pas Robot - ako uradi nešto dobro dobija nagradu ako loše kaznu i na osnovu kazni i nagrada on uči.

AI i ML se očevidlo dobro uklapaju u razvoj pametnih gradova gde na primer imamo auto koji prevozi putnika od kuće do posla i nazada bez vozača i kada je dovoljno blizu kuće šalje signal kapiji ili garaži da se vrata otvore sve ovo se dešava komunikacijom preko interneta upotrebom AI za prevoz i IoT.