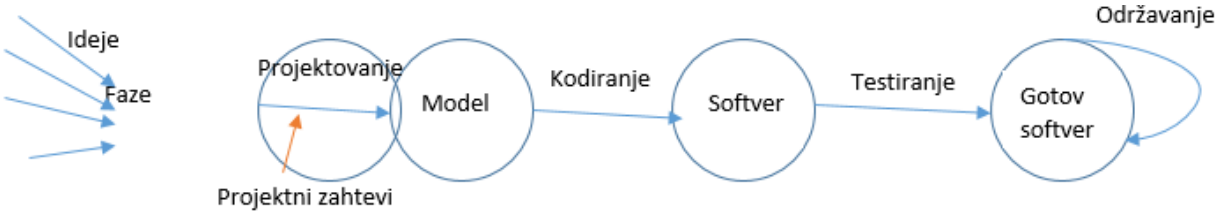
1. Nabrojati i objasniti elemente kvalitetnog softvera.

- Prilagodjlivost (razlicitim OS)
- Optimalan utrosak resursa
- Prosirljivost
- Skalabilnost
- Standardizovanost
- Bezbednost
- Pouzdanost
- Intuitivan interfejs
- Sto krace vreme odziva
- Dokumentovanost
- Konfigurabilnost

2. Životni ciklus softvera i faze razvoja softvera.



To je period koji softver prođe od trenutka identifikacije potreba za softverskim proizvodom do trenutka prestanka korišćenja softverskog proizvoda. Na početku imamo neke ideje, koje na osnovu projektnog zahteva projektujemo u neki model, gde taj model kodiranjem pretvaramo u softver koji nakon toga testiramo i eksploatišemo kao gotov softver, pri čemu radimo na njegovom održavanju i idejama za neki novi softver (slika sa slajda). Postoji više faza u životnom ciklusu softvera:

1. Iniciranje i definisanje projekta (definicija problema) – ovo je najkraća faza. Potrebno je da se pri definiciji izbegnu kontradiktorni i nepotpunu zahtevi. U ovoj fazi se javljaju podfaze kao što su:
    - Iniciranje projekta - investitor opisuje problem i zadatke koje novi sistem treba da ispuni, a kao rezultat toga dobija se projektni zahtev koji sadrži opis problema, ciljeve uvođenja novog sistema, kratak sadržaj funkcija koje novi sistem treba da realizuje, očekivani efekat i dobit, moguće korisnike i ograničenja
    - Preliminarna analiza - ova podfaza se izvršava samo ako projektni zahtev ima manjkavosti ili nedovoljne podatke za donošenje zaključaka, pri čemu se projektni zahtev dopunjuje, precizira i izvode se intervjui sa kompetentnim osobama o validaciji projekta, a nakon dopuna se ponovo daje ocena o izvodljivosti, gde se u ocenu uključuju funkcionalna, tehnička, ekonomska i zakonska sredstva na osnovu kojih je proračunata izvodljivost
    - Priprema projektnog zadatka - u ovoj podfazi se opisuje problem koji se rešava, ciljevi, prednosti u odnosu na postojeće sisteme, ograničenja, mogući korisnici i očekivani rezultati
    - Usvajanje projektnog zadatka - usvaja ga nadležni organ (korisnik ili investitor) i donosi pisanu odluku o tome

2. Analiza i specifikacija zahteva (specifikacija problema) - u ovoj fazi se vrši usaglašavanje raznih specifičnosti eliminišući kontradiktornosti zahteva; usaglašavaju se funkcionalne karakteristike (funkcije koje se dodeljuju na izvršenje, vrednosti parametara koji karakterišu izvršenje funkcija, pouzdanost karakteristika programskog proizvoda) koje, ako su nejasne, mogu da dovedu do nesporazuma između naručioca i realizatora softvera; zahtev sastavlja korisnik u saradnji sa projektantom pri čemu nastaje specifikacija koja predstavlja formalna opis bitnih osobina objašnjenih na jasan i nedvosmislen način tekstualnim jezikom.
3. Idejno projektovanje – predstavlja prvu fazu kreiranja softvera; zahteva da prve dve faze budu korektno završene. U ovoj fazi se programski proizvod razbija na niz nezavisnih jedinica gde se ove jedinice ne razrađuju (ne sadrži detalje) već se posmatraju kao crne kutije sa ulaznim i izlaznim priključcima (idejni projekat). Ova faza ima za cilj projektovanje modela proizvoda pri čemu ovaj projektovani model mora da ima hijerarhijsku strukturu veza između problema, da poštuje principe modularnosti i da je svaki modul funkcionalno nezavistan.
4. Detaljno projektovanje – počinje nakon neformalne provere idejnog projektovanja i sa ovom fazom počinje stvaranje konkretnog softverskog proizvoda. Ukoliko je ova faza uspešna, detaljni projekat će se lako pretvoriti u niz programskih celina. Nakon ove faze se obično pristupa formalnoj reviziji, tj. odbrani uspešnog projekta i odbacivanju projekta koji sadrži krupne nedostatke, dok se sitni nedostaci otklanjanju i ponovo se vrši formalna revizija od strane naručioca i izvršioca.
5. Programiranje – otpočinje tek nakon uspešne formalne revizije. Koriste se različite tehnike kodiranja kojima se obezbeđuje poštovanje principa nezavisnosti programskog sistema.
6. Testiranje – predstavlja direktnu proveru pomoću računara čime se ne dokazuje, već se opovrgava valjanost softvera. Ostvaruje se dobro smišljenim testovima kojima se može dokazati valjanost softvera.
7. Eksploatacija – prodaja, upotreba, održavanje.
8. Dokumentacija – dokumentacija za prodaju, instalaciju i za proceduru rada i korišćenja

3. Nabrojati modele razvoja softvera.

- Model vodopada – da
- Modifikovani model vodopada
- Inkrementalni model – da
- Model prototipa – da
- Visestruko koriscenje softverskih komponenti – vodjen, sa
- Automatska sinteza softvera
- Spiralni model – da
- Model softverske oluje

4. Model *vodopada*.

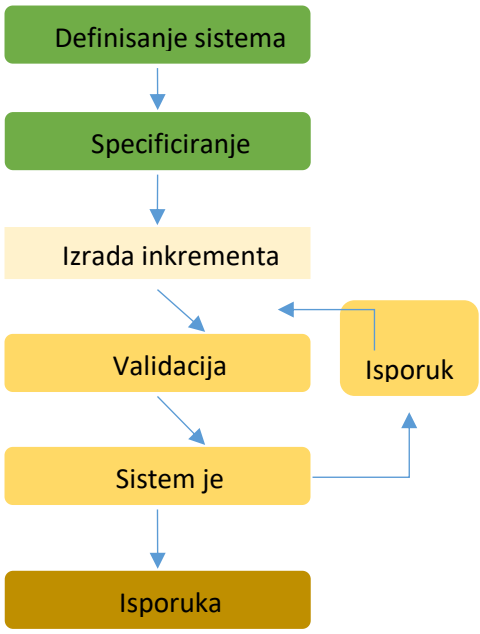
Slika: model vodopada. Zeleno predstavlja iniciranje projekta, plavo predstavlja razvoj softvera, pri čemu arhitekturno i detaljno projektovanje spada u fazu projektovanja. Sve se izvršava sekvencijalno.

Ovaj model je najprostiji model koji prati faze životnog razvoja softvera. Faze ovog modela su *identifikacija problema, definicija problema i ocena izvodljivosti, analiza i specifikacija zahteva, arhitekturno projektovanje, detaljno projektovanja, kodiranje i prečišćavanje, testiranje, instaliranje, eksploatacija i održavanje i ukidanje sistema* kao poslednja faza. Sve ove faze se izvršavaju jedna za drugom, pa otud i sam naziv *model vodopada*. Ovaj model se uglavnom koristi kada su svi zahtevi jasno definisani i nisu podložni promenama u toku razvoja.

Prednosti ovog modela su to što podstiče precizno definisanje zahteva pre samog projektovanja sistema. Takođe podstiče definisanje interakcije među komponenta pre samog kodiranja i omogućava lako vođenje i nadgledanje projekta i tačnije planiranje resursa što vodi do smanjenja troškova razvoja i održavanja. Dobra strana je i to što zahteva formiranje dokumentacije u svakoj fazi.

Glavni nedostaci modela vodopada su ti što specifikacija zamrzava prepoznavanje novih zahteva u ranim fazama razvoja, što se troši dosta vremena na izradu specifikacije i zbog toga se kasni sa kodiranjem. Još jedan nedostatak je taj što je faza kodiranja blizu faze eksploatacije, tj. krajnjem korisniku, pa ukoliko softver ne odgovara zahtevima korisnika, unošenje izmena je vrlo skupo. Takođe ovaj model ima vrlo slabe veze između softvera koji se razvija i prethodno razvijenih softverskih proizvoda. Još jedna mana mu je i to što nije pogodan za male poslovne aplikacije, interaktivne aplikacije, ekspertske sisteme i sisteme bazirane na znanju.

5. *Inkrementalni* model razvoja softvera.



Glavna odlika ovog modela je ta što se softver proizvodi u manjim celinama koji će kasnije činiti ceo softver. Te celine se nazivaju inkrementima. U toku samog razvoja proizvodi se inkrement po inkrement, gde se u svakom inkrementu dodaju nove funkcionalnosti ili ispravljaju greške po zahtevu korisnika. Ovaj model zapravo primenjuje model vodopada u inkrementima.

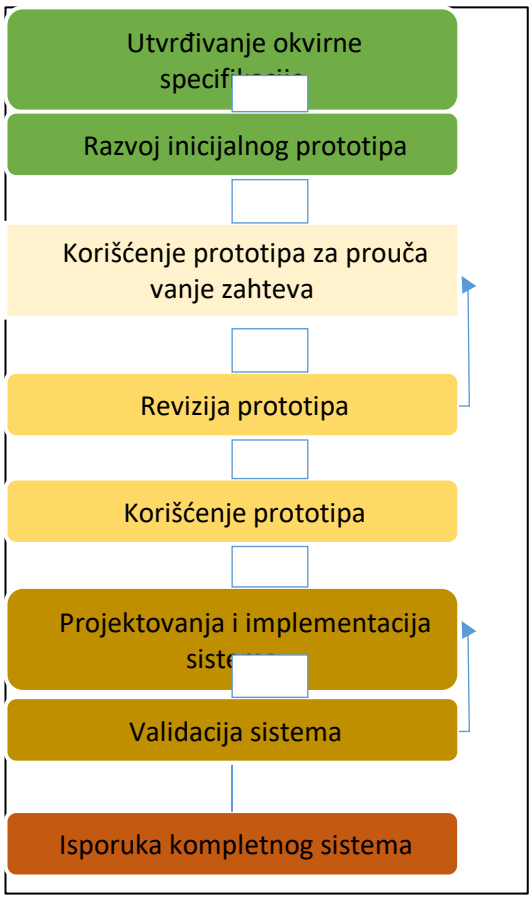
Prednosti ovog modela su skraćenje vremena i redukovanje troškova do pojave incijalne verzije sistema, lakše testiranje jer su inkrementi prostiji od sistema u celini, ranije uključenje korisnika u proces razvoja čime se obezbeđuje njihov veći uticaj na konačan izgled softvera što smanjuje kasnije potrebe za izmenom softvera.

Glavni nedostatak je taj što svaki novi inkrement koji se doda mora biti od nekog značaja za korisnika. Takođe, kako vreme odmiče i novi inkrementi se dodaju, moguća su povećanja troškova razvoja.

6. Poređenje modela *vodopada* i *inkrementalnog* modela razvoja softvera.

Glavni problem modela vodopada je taj što se on sekvencijalno izvršava, pa iz tog razloga imamo veoma kasno uključivanje korisnika u sam proces i dobijanje povratne informacije o ispunjenju zahteva korisnika. Inkrementalni model to rešava na taj način što softver isporučuje u inkrementima, a prilikom same izrade softverskog proizvoda oslanja se na faze modela vodopada. Na ovaj način korisnik dobija veoma ranu sliku o softverskom proizvodu i može lakše da utiče na sam dalji razvoj softvera, a proizvođaču softvera se omogućava lakše upravljanje i nadgledanje projektom.

7. *Prototipni* model razvoja softvera.

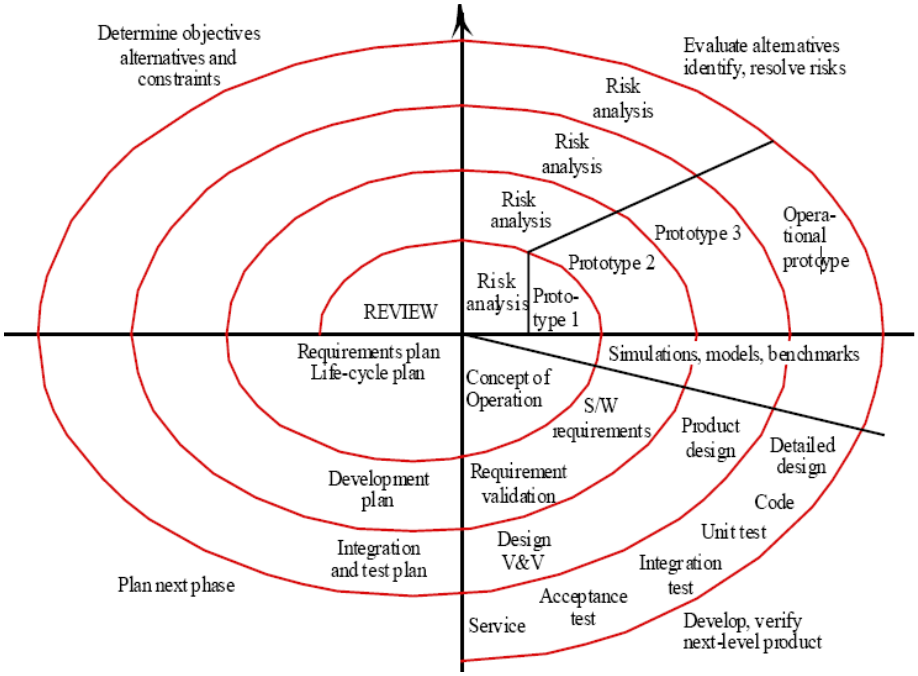


Glavna odlika ovog modela je to što se softver razvija kao prototip koji se dalje dorađuje sve dok ne zadovolji krajnji zahtev potražioca. Pri razvoju prototipa se ignorišu nefunkcionalni zahtevi (brzina, prostor), ignoriše se obrada grešaka (što dovodi do redukovanja pouzdanosti i kvaliteta softvera), ali se ne sme redukovati ni jedan zahtev vezan za korisnički interfejs. Prototip ne sadrži sve funkcije sistema, on nije sastavni deo ugovora, kroz njega se ne mogu na adekvatan način izraziti nefunkcionalni zahtevi i on se nikako ne sme koristiti kao specifikacija. Takođe, prototip ne predstavlja radnu verziju gotovog sistema.

Prednosti ovog modela su sledeće:

- Pri demonstraciji prototipa se mogu otkriti nesporazumi između klijenta i projektanta
- Mogu se lako detektovati izostavljeni zahtevi klijenta
- Lako se identifikuju i otklanjanju nejasnoće u funkcionalnosti sistema
- Lako se otkrivaju nekompletni i nekonzistentni zahtevi
- Brzo se dolazi da skromne verzije sistema, tzv. demonstracioni prototip
- Prototip može poslužiti kao osnova za pisanje specifikacije sistema koji se razvija

8. *Spiralni* model razvoja softvera.

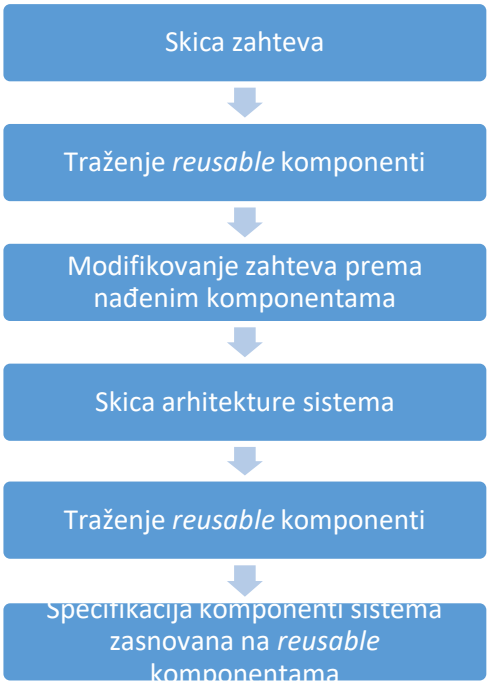


Ova odvratna slika bi trebalo da objasni da je ključna stvar razvijanje prototipa i da na kraju svake faze postoji analiza rizika.

Primenom spiralnog modela se omogućava da se u svakom ciklusu ove spirale primeni različit pristup razvoja softvera i sve to u zavisnosti od analize rizika.

9. Razvoj softvera vođen *višestrukim korišćenjem komponenata*.

Glavni akcenat se stavlja na korišćenje već gotovih softverskih komponenti i njihovog uklapanja u zahteve sistema. Ovaj model koriguje zahteve sistema u odnosu na pronađene postojeće komponente.



Ovo dovodi do sledećih uslova:

- Mora postojati mogućnost da se nađu komponente pogodne za višestruko korišćenje
- Mora postojati njihova dobra specifikacija
- Mora postojati opis njihovog korišćenja
- Mora postojati dovoljno bogata biblioteka ovih komponenti
- Mora postojati visok stepen generalizacije ovih komponenti

Prednosti ovog sistema su što se na ovaj način maksimalno koristi već gotov softver, redukuje se ukupni troškovi razvoja, potrebno je manje softverskih komponenti specificirati, projektovati, implementirati i proveriti, povećava se stabilnost sistema i redukuje se ukupan rizik, bolje se iskorišćavaju specijaliste, lakše se ugrađuje standard u višestruko korišćene komponente i smanjuje se vreme razvoja softvera.

10. Nabrojati gradivne elemente UML-a. Nabrojati i opisati vrste relacija koje se sreću u UML modelima.

Gradivni elementi UML-a su:

1) Stvari
   a) Strukturne
      i) Klasa – opis skupa objekata koji imaju iste atribute, operacije, relacije i semantiku.
      ii) Interfejs – skup operacija koje definišu usluge klasa ili komponenata.
      iii) Kolaboracija – skup uloga i drugih elemenata koji sarađuju da bi ispunili kooperativno ponašanje složenije od proste sume ponašanja elemenata.
      iv) Slučaj upotrebe – opis skupa sekvenci akcija koje sistem izvršava da bi proizveo spolja uočljivo ponašanje bitno za nekog aktera.
      v) Aktivna klasa – klasa čiji objekti poseduju jedan ili više procesa ili niti kontrole.
      vi) Komponenta – fizički i zamenljiv deo sistema koji zadovoljava i realizuje skup interfejsa
      vii) Čvor – fizički element koji postoji u vremenu izvršenja i predstavlja računarski resurs koji u principu poseduje memoriju i najčešće mogućnost obrade
   b) Stvari ponašanja
      i) Interakcija – skup poruka koje se razmenjuju između objekata u određenom kontekstu da bi se ostvarila određena svrha
      ii) Mašina stanja – sekvenca stanja kroz koje objekat ili interakcija prolazi tokom svog života
   c) Stvari grupisanja – najčešće su to paketi koji služe za grupisanje svih ostalih elemenata jezika.
   d) Stvari označavanja – komentari.
2) Relacije ***(O RELACIJAMA VIŠE U PITANJU ISPOD!)***
   a) Zavisnost
   b) Asocijacija
   c) Agregacija
   d) Generalizacija
   e) Realizacija
3) Dijagrami – prezentacija skupa elemenata koji predstavljaju samo jedan pogled na deo modela. Oni ne nose nikakvo značenje.
   a) Statički aspekti sistema: dijagram klasa, objekata, komponenata, raspoređivanja.
   b) Dinamički aspekti sistema: dijagram slučajeva upotrebe, interakcije, stanja, aktivnosti.

11. Nabrojati UML dijagrame za opis dinamickih aspekata sistema.

- Use-case dijagrami
- Dijagrami interakcije (sekvence i kolaboracije)
- Dijagrami stanja
- Dijagrami aktivnosti
- Staticki – dijagrami klasa, objekata, komponenti, rasporedjivanja

12. Relacije u UML-u.

One su jedan od gradivnih elemenata UML-a. Opisuju u kakvom su odnosu stvari na datom dijagramu. Najčešće korišćene relacije su:

1. Zavisnost – promena jedne stvari utiče na ponašanje druge stvari. Predstavlja se isprekidanom strelicom. Ova relacija predstavlja zavisnost između dva elementa i usmerenost te zavisnosti. Zavisna stvar koristi nezavisnu stvar.
    a. Relacija uključivanja predstavlja stereotip ove relacije i predstavlja se sa dodatnim natpisom <<include>> iznad strelice. Ona pokazuje da će jedan element dijagrama, ili jedno ponašanje, uključiti i drugo (ono od koga ide uključuje i ono ka kome ide).
    b. Relacija proširivanja je takođe stereotip relacije zavisnosti i predstavlja se sa dodatnim natpisom <<extend>> iznad strelice. Označava da se ponašanje ka kom ide strelica može proširiti i na ponašanje od kog ide strelica. Kada se osnovno ponašanje proširi na opciono, to se dešava u posebno definisanim tačkama koje se nazivaju tačkama ekstenzije.
2. Asocijacija – strukturna relacija, predstavlja vezu između stvari. Prikazuje se punom linijom gde sa obe strane imamo specificirane uloge i multiplikativnosti. Pored ova dva ukrasa, može imati i ukrase naziv i pravo pristupa (+ svi objekti mogu preko objekta sa druge strane, - samo objekti sa druge strane mogu, # mogu i objekti izvedenih klasa i ~ isti paket) Može biti usmerena i neusmerena (bidirekciona) što označava da objekti međusobno imaju reference jedan na drugog. Asocijacija ne mora da bude između dve stvari, već može da bude i n-arna. Još jedan ukras asocijacije je kvalifikator, tj. ključ, koji se koristi za odabir objekata iz neke strukture. Sama veza asocijacije može imati svoje atribute koji pripadaju klasi koja opisuje tu vezu. Ta klasa se, logično, naziva klasom asocijacije.
    a. Agregacija – asocijacija u varijanti celina-delovi. Celina sadrži delove (celina je sa strane praznog romba). Deo u agregaciji može biti zajednički deo više celina.
    b. Kompozicija – celina je odgovorna za životni vek dela. Postoji strogo vlasništvo celine u odnosu na deo (deo je deo samo jedne celine). Obeležava se punim rombom sa strane celine.
3. Generalizacija – nasleđivanje; objekat generalizovanog elementa (roditelj) se može zameniti objektom specijalizovanog elementa (dete). Označava se punom linijom i trougaonom strelicom. Ova relacija kaže da je stvar od koje ide strelica zapravo specijalizacija stvari ka kojoj ide strelica (dete je specijalizacija svog roditelja). Kod klasa, operacije deteta imaju isti potpis kao operacije roditelja, ali mogu da ih redefinišu što dovodi do polimorfizma.
4. Realizacija – semantička veza u kojoj je jedan element ugovor koji drugi element ispunjava (klasifikator opisuje strukturu i ponašanje, a neki drugi element ispunjava tu strukturu). Postoji u dva konteksta: kontekst interfejsa (realizuje ga klasa ili komponenta) i kontekst slučaja korišćenja (realizuje ga kolaboracija).
5. Agregacija – označava se punom linijom koja sa jedne strane ima prazan romboid, znači da stvar na strani gde je romboid predstavlja celinu a stvar sa druge strane deo pri čemu postojanje dela ne zavisi od postojanja celine
6. Kompozicija – označava se punom linijom koja sa jedne strane ima ispunjen romboid, na isti način označava smisao celine i dela kao i agregacija ali je uslovljeno da celina mora da postoji da bi postojao deo

Sve ove relacije mogu da imaju dodate neke stereotipe, kao što su use, bind, friend, instanceOf kod relacije zavisnosti ili utility, stereotype itd. kod klasifikatora.

13. Ukrasi relacije *asocijacije*.

Ukrasi relacije asocijacije su: ime, uloge, multiplikativnost, pravo pristupa, agregacija/kompozicija.

Ime je naziv te konkretne relacije asocijacije, određuje kako se ona čita.

Ukras uloge označava koju ulogu izvršava klasa sa jedne odnosno sa druge strane asocijacije.

Multiplikativnost označava broj učesnika objekta jedne klase u asocijaciji sa tačno jednim objektom sa druge strane relacije. Može biti 0 (nijedan), 1 (tačno jedan), * (proizvoljno mnogo), neki opseg (2..* npr.) ili neki izraz (0..1, 3..6, itd.).

Pravo pristupa određuje vidljivost objekata u asocijaciji prema spoljašnjem svetu. Kada imamo + znači da objektima sa te strane mogu pristupi svi objekti preko objekta sa druge strane. Znak - označava da objektima klase sa te strane mogu pristupiti samo objekti klase sa druge strane. Znak # označava da i objekti izvedenih klasa drugog kraja imaju pristup preko asocijacije, a ~ znači da i objekti klasa iz istog paketa kao klasa sa drugog kraja imaju pristup preko asocijacije. Pravo pristupa obično stoji uz ulogu.

Ukras agregacija i kompozicije označava da li je asocijacija zapravo agregacija ili kompozicija. Relacija asocijacije ima dva specifična podtipa, oba označavaju da objekat sa jedne strane predstavlja celinu, a onaj sa druge strane deo, s tim što kompozicija nalaže da delovi ne mogu da postoje bez celine, a kod agregacije to ograničenje ne postoji.

14. Ukrasi vidljivosti kod veze (link).

Označavaju vidljivost objekta koji učestvuje u vezi za spoljašnji svet. Označavaju se sa +,-,#,~ na odgovarajućoj strani veze i postavljaju se uz ime odnosno ulogu te strane.
+ znači da je vidljivost javna i da svi objekti mogu da pristupe ovom objektu preko objekta sa druge strane veze.
- Znači da objektima klase sa te strane veze mogu da pristupe samo objekti klase sa druge strane veze.
znači da objektima sa te strane veze mogu da pristupe objekti klase sadruge strane veze i objekti klasa izvedenih iz te klase.
~ znači da pristup objektima te klase imaju pored objekata klase sa druge strane veze i objekti svih klasa koje pripadaju istom paketu kao klasa sa druge strane veze.

15. Objasniti relaciju *agregacije* i relaciju *kompozicije*.

Relacija agregacije je relacija koja stavlja dve klase u odnos deo-celina, gde se celina sastoji od delova koji ne zavise od te celine. Ovi delovi mogu da budu i delovi nekih drugi celina i da postoje nezavisno. Za razliku od agregacije, kod kompozicije i dalje postoji taj odnos deo-celina, ali je sada deo zavisan od celina i ne može da postoji bez nje. Takođe, deo postoji samo za jednu celinu i ne može da bude deo neko druge celine.

16. Opisati relaciju *uključivanja* i relaciju *proširivanja*. Navesti odgovarajuće primere.

Obe ove relacije predstavljaju stereotip relacije zavisnosti.

Relacija uključivanja predstavlja se sa dodatnim natpisom <<include>> iznad strelice. Ona pokazuje da će jedan element dijagrama, ili jedno ponašanje, uključiti i drugo (ono od koga ide uključuje i ono ka kome ide).

Relacija proširivanja predstavlja se sa dodatnim natpisom <<extend>> iznad strelice. Označava da se ponašanje ka kom ide strelica može proširiti i na ponašanje od kog ide strelica. Kada se osnovno ponašanje proširi na opciono, to se dešava u posebno definisanim tačkama koje se nazivaju tačkama ekstenzije.

Primer!!!!!

17. *Use-case* dijagrami.

Dijagrami slučajeva korišćenja prikazuju skup slučajeva korišćenja, aktere i relacije između njih. Oni modeluju kontekst sistema i zahteve sistema i služe da specificiraju neku funkcionalnost i ponašanje aplikativnog sistema. Ukratko, oni nam govore šta sistem radi. Preko njih možemo da prikažemo ponašanje sistema, podsistema, klasa ili interfejsa. Služe i kao provera nakon realizacije sistema da li su svi zahtevi u vidu funkcionalnosti zadovoljeni.

Elementi ovih dijagrama su:

1. Slučajevi korišćenja – opisuju se tokom događaja i to se opisuje kada slučaj korišćenja počinje i kada se završava, kada slučaj korišćenja interaguje sa akterima i kada se razmenjuju objekti. Tok događaja može biti primarni (osnovni) i alternativni i opisuje se na neki od sledećih načina: neformalan struktuiran tekst, formalan struktuiran tekst (sa pred i post uslovima), pseudokod ili dijagramima interakcije (jedan dijagram za primarni i dodatni za alternativne tokove). Obično se predstavlja kao tabela koja sadrži ime, aktere, trigere, preduslove, postuslove, ispravan scenario i alterantivni tok.
2. Akteri – skup uloga koji može biti korisnik ili neki sistem sa kojim se interaguje. Sistem može da interaguje sa jednim ili više spoljašnjih aktera. Akter predstavlja standardni stereotip klase koji se predstavlja posebnim grafičkim simbolom u vidu čiča gliše.
3. Relacije – predstavljaju povezanost aktera i slučajeva korišćenja.
    a. Asocijacije – relacija komunikacije; puna linija od aktera do slučaja korišćenja; komunikaciju iniciraju i jedna i druga strana.
    b. Zavisnosti – konkretno se koriste dva stereotipa ove relacije: uključivanje (<<include>>) i proširivanje (<<extend>>). Predstavlja se isprekidanom linijom sa strelicom. Uključivanje od jednog ka drugom slučaju korišćenja (od A ka B) govori o tome da će ponašanje A da uključi i ponašanje B (ne važi obrnuto). Uključivanjem se opisuje zajedničko ponašanje između dva ili više slučaja korišćenja. Proširivanje od A ka B znači da B može da obuhvati i ponašanje iz A (može da se proširi). Proširivanje se događa u tačno određenim tačkama koje se nazivaju tačke ekstenzije.
    c. Generalizacije – nasleđivanje; puna linija sa trougaonom strelicom. Generalizacija od A ka B kaže da je A specifičan slučaj korišćenja u odnosu na B.
4. Paketi – moguće je grupisanje aktera, slučajeva korišćenja i njihovih veza u pakete i kasnije pozivanje na te pakete u nekom dijagramu; mogućuju gurpisanje delova use-case dijagrama i njhovo ponovno referenciranje u tom formatu

18. Dijagrami *klasa*

Ovim dijagramom se prikazuje skup klasa, interfejsa, kolaboracija i njihove relacije. Elementi dijagrama klasa su stvari (klase, interfejsi, kolaboracije, paketi, objekti) i relacije (zavisnost, generalizacija, asocijacija i realizacija). Ovaj dijagram specificira logičke i statičke aspekte modela.

Klasa predstavlja opis skupa objekata sa istim atributima, operacijama, relacijama i semantikom. Ona može da implementira jedan ili više interfejsa. Njome se opisuje apstrakcija iz domena problema, ali i apstrakcija iz domena rešenja. Opisuje se nazivom klase (jednostavan i jedinstven za sistem, može da sadrži i putanje paketa gde se nalazi klasa), atributima (imenovana svojstva klase koja opisuje opseg vrednosti koje instance tog svojstva mogu da koriste; može da se pišu i sa tipom i podrazumevanom vrednošću), operacijama (implementacije servisa koji se mogu zahtevati od objekta klase; mogu da sadrže potpis sa tipovima i podrazumevanim vrednostima, kao i povratni tip) i odgovornostima (opisuju čemu klasa služi; dobra klasa ima bar jednu i ne više od nekoliko odgovornosti; piše se u zasebnom odeljku gde svaka odgovornost počinje sa --). Na dijagramu klasa moguće je da neka klasa ima prazne odeljke ili da se odeljci izostave, što ne znači da ne postoje ti odeljci, već da je dijagram uprošćen jer ti odeljci nisu relevantni za njega. Moguće je da imaju opisni prefiks koji se navodi kao stereotip. Apstraktne klase i operacije se pišu iskošenim slovima, a zajednički članovi klase se pišu podvučeno. Prava pristupa atributima i operacijama su javni (+), zaštićeni (#) i privatni (-). Na dijagramu klase se još mogu i označiti da li klasa ima roditelje ili potomke i da li postoji ograničenje po broju njenih instanci (0 je uslužna klasa, 1 je singlton, a podrazumevano je da je proizvoljan broj).

Na klasnim dijagramima se pojavljuje sve relacije UML-a. Najčešće se sreću zavisnost, asocijacija, generalizacija i realizacija. ***O relacijama u nekom od pitanja iznad!***

19. Dijagrami *aktivnosti*.

Ovim dijagramima se modeluje dinamički aspekti sistema. Podsećaju na dijagrame toka jer prikazuju tok kontrole od jedne do druge aktivnosti koje se izvršavaju nad objektima. Mogu se koristiti za opis dinamike skupa objekata ili za opis toka neke operacije. Aktivnost predstavlja obradu, koja nije atomična, u okviru automata stanja, dok je akcija atomska obrada koja menja stanje. Aktivnost je rezultat akcije. Osnovni elementi su: stanje aktivnosti, stanje akcije, tranzicije između stanja, sekvencijalno grananje u toku kontrole i konkurentno grananje u toke kontrole. Stanje aktivnosti se može dalje dekomponovati, što dovodi do toga da se svako stanje može predstaviti posebnim dijagramom stanja. Grafički se ne razlikuje prikaz stanja aktivnosti i stanja akcije, sem u tom da stanje aktivnosti može da sadrži ulazno-izlazne akcije i specifikaciju podautomata. Tranzicija je legalna putanja od jednog do drugog stanja i obeležava se strelicom. Moguća je pojava grananja i iteracija. Konkurentna grananja govore o tome da se nit kontrole račva u nekoj tački na više konkurentnih niti što se obeležava sinhronizacionom tačkom (zadebljana linija).

Moguće je deliti aktivnost na plivačke staze koje specificiraju odgovornosti za delove celokupne aktivnosti. One nemaju nikakvo dublje značenje. Svaka staza predstavlja neki entitet iz realnog sveta i svaka staza poseduje neka stanja, ali je tranzicija moguća iz staze u stazu.

Na ovim dijagramima je takođe moguće prikazati tok objekta preko relacije zavisnosti.

Podsećaju na flow charts i opisuju akcije koje se izvršavaju nad objektima. Osnovni elementi su:

- stanje aktivnosti – grupa atomičnih akcija
- stanje akcije – neka akcija koja se ne može dalje dekomponovati
- tranzicije između stanja – legalni prelazi it jednog u drugo stanje

- sekvencijalno grananje u toku kontrole – sekvencijalno grananje na različite tokove, samo jedan od tokova će biti izvršen
- konkurentno grananje u toku kontrole – označava podelu izvršenja na više niti ili procesa, delovi u razdvojenim granama se istovremeno izvršavaju

20. Dijagrami *interakcije*.

Interakcije je ponašanje koje obuhvata skup poruka koje se razmenjuje između skupa objekata u nekom kontekstu sa nekom namenom. Od ovih poruka se očekuje da će uslediti neka aktivnost. Ovim dijagramima se modeluju dinamički aspekti modela.

Postoje dva prikaza interakcija, tako da imamo dva dijagrama interakcije koja na različiti način prikazuju istu informaciju:

1. Dijagram sekvence – naglašava vremensko uređenje interakcije. Poruke su uređene u neku vremensku sekvencu i dodatno se obeležavaju brojem. Moguća je pojava fokus kontrola koje definišu period u toku kog objekat obavlja jednu akciju.
2. Dijagram kolaboracije – naglašava strukturu veza između učesnika interakcije; tretiraju objekte kao prototipske stvari sa specifičnim ulogama.

U zavisnosti od toga da li nam treba da vremenski modelujemo sistem ili da prikažemo samo njegovu organizaciju i tok poruka kroz njega, koristimo ili dijagram sekvence ili dijagram kolaboracije.

Kontekst interakcije može biti sistem (interakcije u kolaboraciji objekata koji postoje u sistemu), operacija (interakcije su među objektima koji implementiraju operaciju), klasa (atributi kolaboriraju međusobno i sa globalnim objektima i parametrima operacije kako bi opisali semantiku klase) ili slučaj korišćenja (scenario slučaja korišćenja).

Glavni akteri dijagrama interakcije su objekti koji imaju svoje uloge. Oni mogu biti konkretne stvari (konkretna instanca neke klase) ili prototipske stvari (proizvoljna instanca). Kod ovih dijagrama se objekti mogu pojaviti kao instance apstraktnih klasa i interfejsa, ali ove instance ne označavaju konkretne stvari već su to prototipske stvari. Dijagrami interakcije, za razliku od dijagrama klasa, predstavljaju dinamički aspekt specificirajući sekvencu poruka koju razmenjuju objekti. Ove poruke između objekata su zapravo pozivi adekvatnih operacija. Objekti mogu da imaju promenljiv životni vek, zato možemo da uvedemo neka ograničenja koja se obeležavaju na sledeći način: *{new}* – objekat se kreira za vreme izvršenja interakcije, *{destroyed}* – uništava se pre završetka interakcije, *{transient}* – kreira se i uništava za vreme interakcije. (Sve ovo važi i za veze.)

Ograničenjima možemo da specificiramo životni vek objekata i veza:
- {new} – objekat ili veza se kreira za vreme izvršenja interakcije
- {destroyed} – objekat ili veza se uništava pre završetka interakcije
- {transient} – obejkat ili veza se kreira i uništava za vreme trajanja interakcije

Veza je instanca relacije asocijacije između objekata odgovarajućih klasa. Preko veze se šalju određene poruke. Veza ima slične ukrase poput relacije asocijacije (ime, uloga, navigabilnost, agregacija), ali nema multiplikativnost jer u vezi uvek učestvuje po jedan objekat sa svake strane. Takođe postoji i ukras vidljivosti koji specificira način na koji jedan objekat vidi objekat sa druge strane veze. Navodi se kao stereotip veze i može biti: *association* (vidljiv jer postoji asocijacija između klasa), *self* (vidljiv jer sam sebi šalje poruku), *global* (vidljiv jer je u globalnom opsegu), *local* (vidljiv jer je u lokalnom opsegu), *parametar* (vidljiv jer je argument operacije).

Preko veza se šalju poruke. Prijem poruke se može smatrati instancom događaja, a slanje poruke predstavlja izvršenje neke narede (metod/operacija). UML u sebi sadrži sledeće poruke: *call* (pokreće se operacija primaoca), *return* (vraća se vrednost pozivaocu), *send* (šalje se asinhroni signal primaocu), *create* (kreira se objekat), *destroy* (uništava se objekat) i *become* (objekat je isti sa obe strane veze i menja prirodu). U okviru poruke se može prikazati i njen argument i vraćena vrednost.

21. Konteksti *interakcije*.

Kontekst može biti:

- Sistem ili podsistem – opisuju se interakcije u kolaboraciji objekata koji postoje u sistemu ili podsistemu npr. Web commerce – sarađuju objekti na strani servera sa objektima na strani klijenta
- Operacija – opisuju se interakcije među objektima koji implementiraju operaciju. Parametri operacije, lokalni i globalni objekti interaguju da izvrše algoritam operacije
- Klasa – Interakcija opisuje semantiku klase. Atributi klase, globalni objekti i parametri operacija međusobno interaguju kako bi opisali semantiku klase.
- Use-case – interakcija je scenario za slučaj korišćenja

22. Poruke u dijagramima *interakcije*.

Poruke se jedan od elemenata dijagrama interakcije i one predstavljaju specifikaciju komunikacije između objekata koja prenosi informacije. Poruke se šalju preko veze. Prijem poruke se može smatrati instancom događaja, a slanje poruke predstavlja izvršenje neke narede (metod/operacija). UML u sebi sadrži sledeće poruke: *call* (pokreće se operacija primaoca), *return* (vraća se vrednost pozivaocu), *send* (šalje se asinhroni signal primaocu), *create* (kreira se objekat), *destroy* (uništava se objekat) i *become* (objekat je isti sa obe strane veze i menja prirodu). U okviru poruke se može prikazati i njen argument i vraćena vrednost.

Ovi dijagrami opisuju interakciju među objektima. Opisuju dinamički aspekt sistema i to kada i na koji način objekti komuniciraju razmenom poruka. Postoje dva tipa dijagrama interakcije:

- Dijagrami sekvence – opisuju vremenski tok interakcija među objektima
- Dijagrami kolaboracije – ne vodi se računa o vremenskim trenucima interakcije već samo o tome ko sa kim i na koji način interaguje

Ova dva tipa dijagrama međusobno su potpuno ekvivalentna tj. Predstavljaju iste informacije i moguće je iz jednog dijagrama izvući drugi dijagram. Objekti mogu da komuniciraju u različitim kontekstima. Konteksti mogu biti ... . Čitav dijagram sastoji se od objekata, veza i poruka. Veze su instance veza asocijacije iz dijagrama klasa i preko tih veza se šalju poruke. Ove veze mogu imati sve ukrase kao i veza asocijacije izuzev multiplikativnosti jer je ovde uvek odnos 1 na 1. Ukas vidljivosti specificira se na malo drugačiji način, može da ima sledeće vrednosti:

- Association – objekat je vidljiv jer postoji veza asocijacije između pošiljaoca i primaoca poruke
- Self - objekat je vidljiv jer sam sebi šalje poruku
- Global – objekat je vidljiv jer pripada neko opsegu koji je globalan za trenutni opseg
- Local – objekat je vidljiv jer je u lokalnom opsegu
- Parametar – objekat je vidljiv jer je on parametar operacije

Preko veza se šalju poruke, to su zapravo pozivi funkcija objekata. Postoji nekoliko različitih vrsta poruka:

- Call – poziva neku operaciju objekta primaoca
- Return – vraća vrednost pozivaocu

- Send – neki signal se asinhrono šalje primaocu
- Create – objekat se kreira
- Destroy – objekat se uništava
- Become – objekat menja prirodu (sa obe strane veze je isti objekat samo prelazi iz jednog oblika u drugi)

23. Dijagrami *sekvence.*

Dijagram sekvence predstavlja podvrstu dijagrama interakcije koji služi za naglašavanje vremenskog uređenja interakcije. Poruke su uređene u neku vremensku sekvencu i dodatno se obeležavaju brojem. Moguća je pojava fokus kontrola koje definišu period u toku kog objekat obavlja jednu akciju. Na dijagramu sekvence svaki objekat ima svoju vremensku osu (predstavlja se isprekidanom linijom) i poruke se šalju u nekom uređenom redosledu od jedne ose do druge.

24. Dijagrami *kolaboracije*.

Dijagram kolaboracije je podvrsta dijagrama interakcije koji naglašava strukturu veza između učesnika interakcije. U njemu se objekti tretiraju kao prototipske stvari sa specifičnim ulogama. On služi da se samo prikaže tok razmene poruka između objekata.

25. Standardni UML stereotipovi elemenata u dijagramima komponenata.

Dijagrami komponenata prikazuju fizičku organizacjiu softverskog sistema. Sadrže komponente, interfejse i pakete i relacije među njima. Komponenta može da bude – izvorna datoteka, izvršna datoteka, biblioteka, tabela i dokument. Komponenta je fizička reprezentacija klase, i ona može da implementira razne interfejse. Interfejs je skup operacija koji se koristi da bi specificirao neki servis klase ili komponente.

Standardni UML stereotipovi elemenata:

- Executable – komponenta koja može da se izvršava na čvoru
- Library – statička ili dinamička objektna biblioteka
- Table – tabela baze podataka
- File – Izvorni kod ili podaci
- Document - dokument

26. Dijagrami *stanja*.

Oni su jedni od dijagrama koji služe za opisivanje dinamičkih aspekata sistema. Ovim dijagramima se modeluju različita stanja jednog objekta u toku njegovog životnog ciklusa. Promena ovih stanja je izazvana od strane nekih događaja. Ovaj dijagram nam omogućava da jedan objekat posmatramo kao konačni automat čija se stanja menjaju u zavisnosti od događaja koji se dešavaju unutar sistema (ponašanje vođeno događajima). Takođe nam pomažu da odredimo celokupno ponašanje sistema u toku njegovog izvršavanja u zavisnosti na određene događaje koji se javljaju. Ovaj dijagram prikazuje tok poruka kroz objekat. Sam objekat može biti instanca klase, slučaj korišćenja ili ceo sistem u celini. Koriste se i za modelovanje reaktivnih sistema (sistema koji odgovara na signale aktera iz spošaljenjeg sveta).

Elementi dijagrama stanja su:

- Događaj – zbivanje koje nema trajanje i može prouzrokovati prelaz
- Zaštitni uslov – Bulov izraz koji čini prelaz mogućim kada je uslov ispunjen
- Akcija – atomska radnja koja je pridružena prelazu. Može biti:
    - Poziv operacije objekta vlasnika automata stanja ili drugog objekta koji je vidljiv datom objektu
    - Kreiranje ili uništavanje drugog objekta
    - Slanje signala nekom objektu

Elementi dijagrama stanja su:

1) Stanja – predstavlja uslov ili situaciju u kojoj objekat može da postoji pri čemu objekat u tom stanju zadovoljava neki uslov (korisnik zadovoljava uslov registracije), obavlja neku aktivnost (program je u stanju izvršenja) ili samo čeka događaj (procesor koji čeka neki signal).
    a) Grafička notacija: Stanje se predstavlja zaobljenim pravougaonikom. Početno i završno stanju su specijalna stanja i ona se predstavljaju punim krugom i punim krugom sa prstenom.
    b) Elementi stanja: ime stanja (nije obavezno), ulazna akcija (atomična radnja koja se obavlja pri ulazu), izlazna akcija (atomična radnja pri izlazu iz stanja), aktivnost (radnja koju objekat vrši dok je u datom stanju; nije atomična), podstanja (stanja unutar datog stanja), odloženi događaji (događaji koji se ne obrađuju u datom stanju već se smeštaju u red i čekaju na stanje koje će da ih izvrši) i unutrašnje tranzicije (tranzicije koje obrađuju događaje ali zadržavaju objekat u istom stanju; ne izazviaju izlazne i ulazne akcije).
    c) Stanja mogu biti jednostvna (bez unutrašnje strukture) i kompozitna stanja (stanje koje je samo po sebi automat stanja). Kompozitna (ugnježdena) stanja se koriste da bi se smanjila grafička kompleksnost. Kod ovih stanja razlikujemo nadstanje (ono koje obuhvata više unutrašnjih stanja) i podstanje (unutrašnje). Objekat u podstanju je istovremeno i u nadstanju. Razlikujemo i 2 tipa podstanja: sekvencijalna (objekat je u samo jednom podstanju) i konkurentna podstanja (izvršavaju se u paraleli; objekat je u svakom podstanju). Kod sekvencijalnog podstanja je moguće korišćenje stanja sa istorijom, tj. stanja koje pamti u kom podstanju je nadstanje bilo pre izlaska.
2) Događaji koji uzrokuju promenu stanju
    a) Događaj se odvija u toku *prelaza* iz jednog u drugo stanje. Prelaz predstavlja relaciju između dva stanja i on ukazuje na to da objekat napušta jedno stanje i ulazi u drugo (pri tom obavljajući izlazne i ulazne akcije ova dva stanja) kada se dogodi specificirani događaj. Prelazi se na dijagramima obeležavaju strelicom iznad koje se piše ime događaja, zaštitni uslov koji čini prelaz mogućim kada je on ispunjen i akcija koja se odvija pri prelasku iz jednog u drugo stanje (atomična radnja koja može biti poziv operacije objekta vlasnika stanja ili nekog drugog objekta, kreiranje ili uništavanje nekog drugog objekta ili neko slanje signala).
3) Akcije koje nastaju kao rezultat promene stanja

Murov automat je onaj kod koga su akcije vezane za stanja, a Milijev je onaj kod kog su akcije vezane za tranzicije. Kada šaljemo nekom objektu signal, možemo da ga prikažemo na dijagramu stereotipnom vezom *send*.

27. Elementi *stanja* i elementi *prelaza*.

Elementi stanja:

- Ime – neki tekst kojim se jedno stanje razlikuje od drugih stanja, međutim stanje može biti i anonimno
- Ulazna akcija – atomska radnja koje se obavi pri ulasku u stanje
- Izlazna akcija – atomska radnja koja se obavi pri izlasku iz stanja
- Aktivnost – neatomska radnja koja se izvršava dok je objekat u datom stanju
- Podstanja – stanja koja postoje unutar datog stanja, sekvencijalno ili konkurentno aktivna
- Odloženi događaji – lista događaja koji se ne obrađuju u datom stanju već se smeštaju u red
- Unutrašnje tranzicije – tranzicije koje obrađuju događaj ali zadržavaju objekat u istom stanju (nisu samo-tranzicije, ne izazivaju izlaznu akciju pa odmah zatim ulaznu akciju)

Elementi prelaza:

- Događaj – zbivanje koje nema trajanje i može prouzrokovati prelaz
- Zaštitni uslov – Bulov izraz koji čini prelaz mogućim kada je uslov ispunjen
- Akcija – atomska radnja koja je pridružena prelazu. Može biti:
    - Poziv operacije objekta vlasnika automata stanja ili drugog objekta koji je vidljiv datom objektu
    - Kreiranje ili uništavanje drugog objekta
    - Slanje signala nekom objektu

28. Dijagrami stanja.

Prikazuju automate stanja fokusirajući se na ponašanje vođeno događajima. Koristi se da modelira ponašanje reaktivnih sistema, onih koji odgovaraju na signale koje daju akteri iz spoljašnjeg sveta. Elementi: stanja, događaji koji prouzrokuju promenu stanja i akcije koje rezultuju iz promene stanja. Stanje je situacija u kojoj objekat može da se nađe pod nekim uslovom i u kome onda izvršava neku aktivnost. Početno i završno stanje su pseudostanja. Milijevi i Murovi automati, elementi stanja, elementi prelaza i stanja sa istorijom. Stanja mogu da budu jednostavna, kompozitna (nadstanja), ugnježdena (podstanja).

29. Stanja sa (dubokom) istorijom.

Kada se uđe u stanje koje obuhvata više unutrašnjih stanja obično se kreće od inicijalnog podstanja, ali je nekad potrebno i da se krene od podstanja iz kojeg je poslednji put napušteno ovo nadstanje. U tom slučaju je potrebno da nadstanje pamti istoriju. Stanje sa istorijom obeležava se kao kružić sa slovom H ili H* gde H označava da se pamti plitka istorija odnosno istorija samo za neposredno ugnježden automat stanja dok H* označava da se pamti duboka istorija odnosno istorija za sve automate stanja koji su ugnježdeni, do proizvoljne dubine.

30. Definicija projektnih obrazaca. Katalozi projektnih obrazaca. GoF katalog projektnih obrazaca.

Projektni obrasci kod softvera predstavljaju opise objekata i klasa koje komuniciraju i prilagođeni su rešavanju opšteg problema projektovanja u određenom kontekstu. Elementi projektnih obrazaca:

- ime obrasca – u nekoliko reči opisuje na šta se odnosi taj obrazac
- problem – opis situacije u kojoj se obrazac koristi
- rešenje – rešenje problema u formi dijagrama klasa
- posledice – benefiti korišćenja obrasca za rešenje navedenog problema

Postoji više definicija. Recimo GoF definicija glasi da je projektni obrazac kod softvera opis objekata i klasa koje komuniciraju i prilagođene su rešavanju opšteg problema projektovanja u određenom kontekstu.

Katalog projektnih obrazaca je skup povezanih obrazaca nekog domena. Katalog obrazaca može biti katalog za distribuirane sisteme, za veštačku inteligenciju, za telekomunikacione sisteme, itd. Preko kataloga grupišemo projektne obrasce i lako možemo da nađemo međusobno slične obrasce.

		Namena		
		Kreiranje	**Struktura**	**Ponašanje**
Domen	**Klasa**	Factory method	Adapter	Interpreter Template method
	Objekat	Abstract factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of responsibility Command Iterator Mediator Memenot Flyweight Observer State Strategy Visitor

Tabela iznad prikazuje GoF katalog projektnih obrazaca koji deli projektne obrasce na osnovu dva kriterijuma: kriterijum namene koji može biti kreiranje (obrazac se odnosi na proces kreiranja objekta), struktura (obrazac se odnosi na kompoziciju klasa ili objekata) ili ponašanje (obrazac opisuje način interakcije između objekata i klasa), i kriterijum domena koji govori o tome da li se obrazac primenjuje prvenstveno na klase (odnos između klasa koji se ostvaruje nasleđivanjem) ili na objekte (odnos između objekata koji se uglavnom uspostavlja dinamički, u toku vremena izvršenja).

31. Dati definiciju i objasniti osnovne elemente projektnih obrazaca.

Postoji više definicija. Recimo GoF definicija glasi da je projektni obrazac kod softvera opis objekata i klasa koje komuniciraju i prilagođene su rešavanju opšteg problema projektovanja u određenom kontekstu.

Projektni obrazac uglavnom ima četiri glavna elementa:

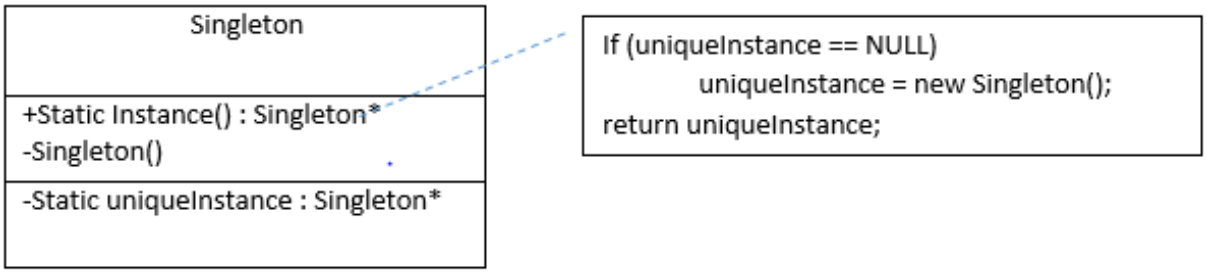
1. Ime obrasca – nekoliko reči koje najbolje opisuje problem, rešenje i posledice primene obrasca.
2. Problem – opis situacije u kojoj se obrazac koristi.
3. Rešenje – opis elemenata koji čine projekat, njihove veze, odgovornosti i saradnje.

4. Posledice – rezultati i ocene primene obrasca.

Dodatno pitanje: opis projektnih obrazaca!
Grafički prikaz projektnih obrazaca nije uvek dovoljan. Pored toga se svakom obrascu dodaje opis koji ima svoju definisanu struktura koja se sastoji od imena obrasca i klasifikacije (dobro ime je od suštinskog značaja kako bi najbolje opisalo obrazac; klasifikacija je na osnovu domena ili namene), namena (odgovor na pitanje šta obrazac radi, koja mu je namena i na koji problem projektovanja se odnosi), takođe poznat kao (AKA; drugo ime za obrazac), motivacija (scenario koji ilustruje problem koji se najčešće rešava), primenjivost (kada treba primeniti obrazac), struktura (grafički prikaz klasa), učesnici (klase i njihova zaduženja u obrascu), saradnje (način saradnje učesnika u sprovođenju sopstvenih odgovornosti), posledice (koji su rezultati primene obrasca), implementacija (greške, pomoći ili tehnike koje treba znati pri implementaciji obrasca), primer koda, poznate primene (obrasci u realni sistema), povezani obrasci (usko povezani obrasci i njihove razlike).

32. Objasniti projektni obrazac *singleton* i dati primer korišćenja.

Singleton obrazac obezbeđuje da klasa ima samo jednu instancu i obezbeđuje globalni pristup toj instanci. Najčešće se koristi u sistemima gde je potrebno da neka klasa ima samo jednu instancu. Briga o jedinoj instanci klase je poverena samoj klasi. Prema GoF, on spada u obrasce čija je namena kreiranje u domenu objekta.



Najčešće se realizuje tako što se kao parametar klase postavi statički pointer na objekat klase, i samo pri prvom pozivu se taj pointer i instancira dok se pri svakom sledećem vraća već instancirani pointer, dok je konstruktor klase privatan.

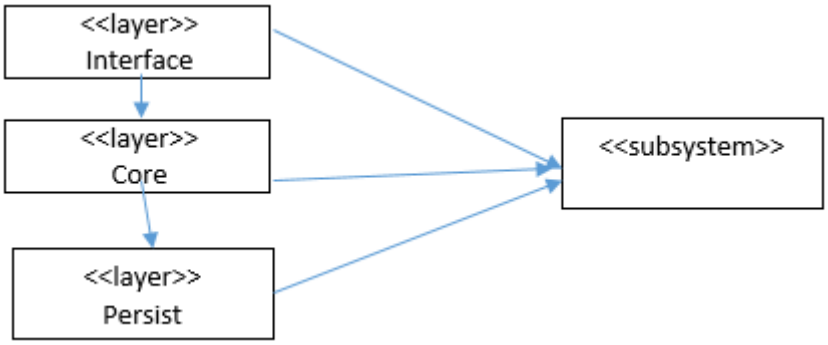
Veze koje smo imali sa klasom sada postaju veze sa Singleton instancom te klase. Primenjuje se svaki put kada u nekom sistemu imamo neku klasu koja je povezana sa velikim brojem drugih klasa a njena kardinalnost u svim tim vezama je 1.
Singleton možemo upotrebiti u softveru gde je potrebno da neke klase tog softvera beleže nešto u neki fajl (logging), npr. server. Na ovaj način nećemo imati 20, 30, 40 klasa od kojih je svaka zadužena za beleženje akcije isto tolikog broja klasa koje čine naš sistem (u ovom slučaju server) gde bismo morali da odradimo neku zamršenu sinhronizaciju u pozadini, već će jedna singleton klasa obavljati ceo taj posao.

33. Obrazac Layer Architecture Pattern.

Nije projektni obrazac, već arhitekturni obrazac, podrazumeva podelu aplikacije na slojeve. Najpoznatiji je pristup 3 sloja:

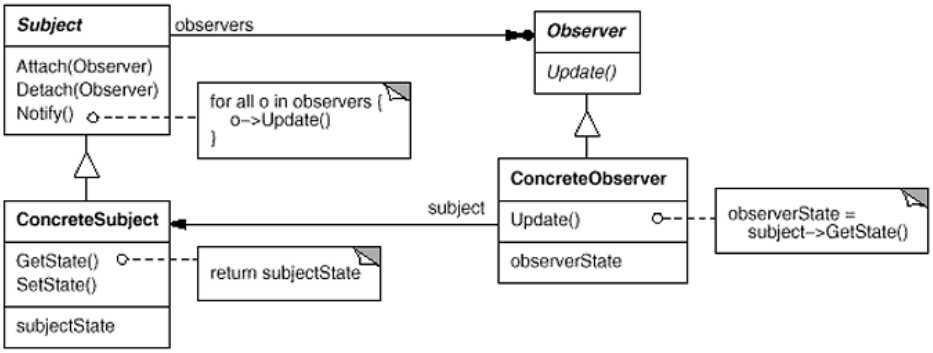
- Sloj korisničkog interfejsa – sloj najveće apstrakcije
- Bussiness tj. Core sloj – čitava funkcionalna logika
- Sloj za trajno čuvanje podataka

Ako rešimo da menjamo neki sloj te promene utiču samo na njega, ne i na ostale slojeve. Takođe svaki sloj je u vezi samo sa slojevima neposredno iznad i ispod njega, i po pravilu nivo apstrakcije opada pri kretanju po slojevima odozgo na dole.
Primena: Java virtuelna mašina, API-ji, informacioni sistemi, operativni sistemi.



34. Objasniti *observer* projektni obrazac i dati primer korišćenja.

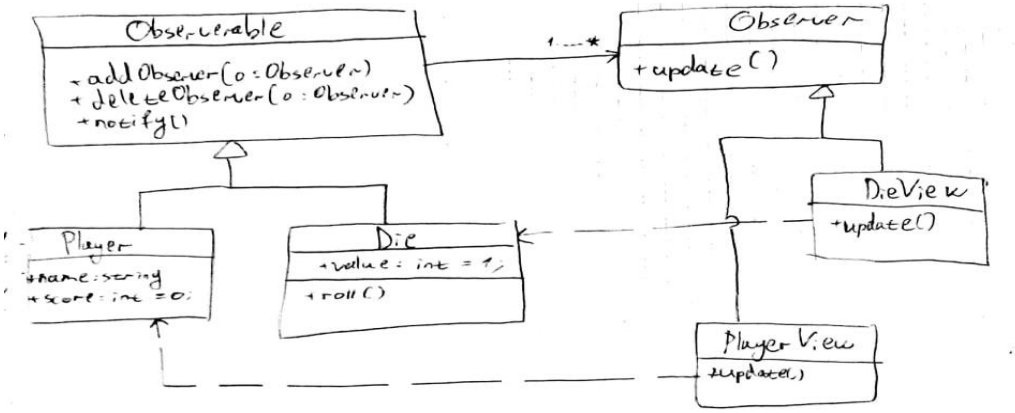
Ovaj obrazac definiše zavisnosti tipa jedan na više između različitih objekata i obezbeđuje da se promena stanja u jednom objektu automatski reflektuje u svim zavisnim objektima. Zbog toga se najčešće sreće u sistemima gde se promena dešava na jednom mesto, ali je potrebno da se ona reflektuje na više drugih, nezavisnih mesta. Kada se promena desi, objekat koji je zadužen za njenu kontrolu šalje svojim observer-ima notifikaciju o tome da se desila promena. Ovaj obrazac se sastoji od jednog subjekta i više zavisnih posmatrača, observera, tog subjekta. Prema GoF, ovaj obrazac ima namenu ponašanja u domenu objekta.



U ovom obrascu možemo da identifikujemo više učesnika:

- Subjekat – onaj nad kojim se dešavaju promene. Zna sve svoje observere i pruža interfejs za dodavanja i uklanjanje observerskih objekata. U klasnom dijagramu iznad je predstavljen kao apstraktna klasa; apstraktna klasa koja opisuje element na kome se dešavaju promene, može da ima više Observer-a koji su zainteresovani za njega
- Observer – interfejs sa metodom za obaveštavanje i promenu stanja iz koga su izvedeni konkretni observereri; apstraktna klasa, predstavlja element koji je zainteresovan za promene na nekom subjektu
- Konkretan subjekat(Concrete Subject) – čuva trenutno stanje u kome se nalazi i prosleđuje obaveštenja svojim observerirma kada se to stanje promeni; ne aprstraktna klasa izvedena iz klase Subject, predstavlja konkretni subjekat koji ima neko stanje
- Konkretan observer(Concrete Observer) – implementira konkretan mehanizam koji se desi prilikom promene stanja; ne apstraktna klasa izvetena iz klase Observer koja predstavlja nekog konkretnog observer-a koji je vezan za neki konkretni subjekat i kada se nad njime pozove

update (kada mu subjekat javi da se desila neka promena) on iz svog konkretnog subjekta čita njegovo stanje i pamti ga kao aktuelno

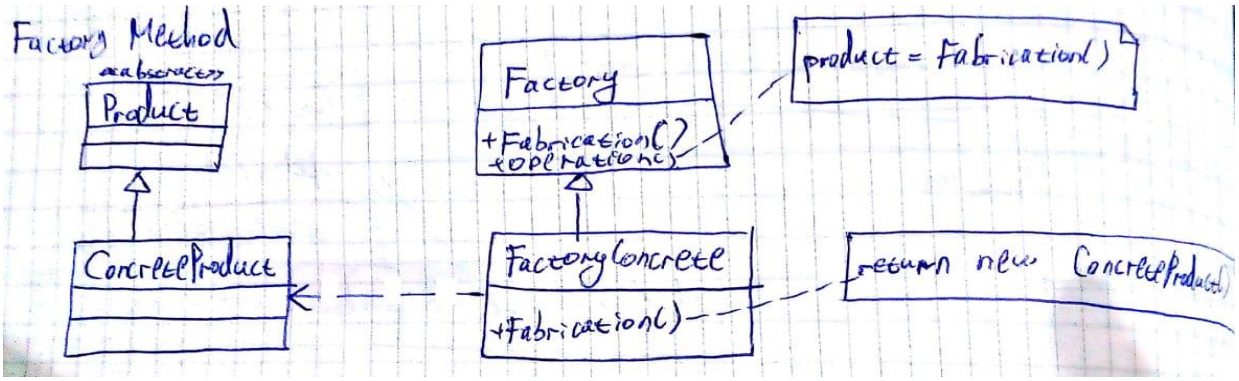


Observer možemo iskoristiti u softveru za statistiku, gde imamo jedan konkretan subjekat predstavljen tabelama i mi unosimo neke podatke u te tabele. Zamislimo da je tabela prazna i da smo pri tom otvorili prikaz grafikona. Kada unesemo neke podatke u tabelu, prikaz grafikona, koji je u našem slučaju konkretan observer, biva obavešten od strane našeg konkretnog subjekta o promeni vrednosti, izvršava predefinisane metode za osvežavanje prikaza i prikazuje trenutno stanje na osnovu toga.

35. Objasniti Factory Method projektni obrazac i dati primer koriscenja.

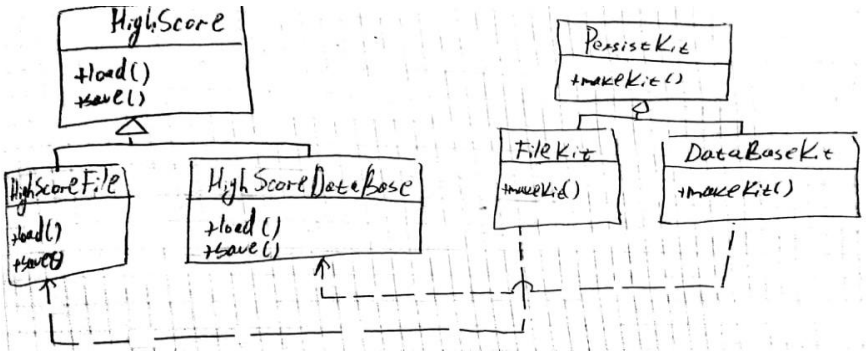
Factory method pomaže kod situacije kada treba da ostavimo prostora da neka funkcionalnost može da se realizuje na više različitih načina. Npr. Ako je u pitanju trajno smeštanje podataka, da ne bismo odmah fiksno napravili tu funkcionalnost da radi samo sa bazom podataka možemo da koristimo ovaj obrazac da ostavimo prostor da ta funkcionalnost može da se realizuje i smeštanjem u fajl i smeštanjem u bazu podataka i možda na još neki dodatan način.

Definiše interfejs za kreiranje objekta ali samo kreiranje instance objekta prepušta potklasama na čijem nivou će biti doneta odluka koji od konkretnih načina implementacije neke funkcionalnosti će biti korišćen.



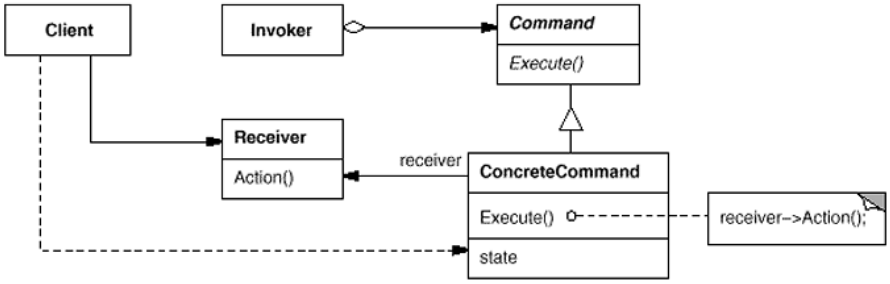
Klasa Factory daje okvir da će objekat tipa Product biti kreiran ali je na instanci klase ConcreteFactory da definiše koji konkretan objekat proizvoda će biti kreiran. Evo konkretnog primera primene, redimo za slučaj kada treba pamtiti HighScore trajno, pri čemu to može biti u bazu podataka i u fajl:

Kreiranje konkretnog kit-a dovodi do kreiranja konkretne realizacije highScore klase i samim tim do donošenja odluke koji će tip trajnog pamćenja da bude primenjen.

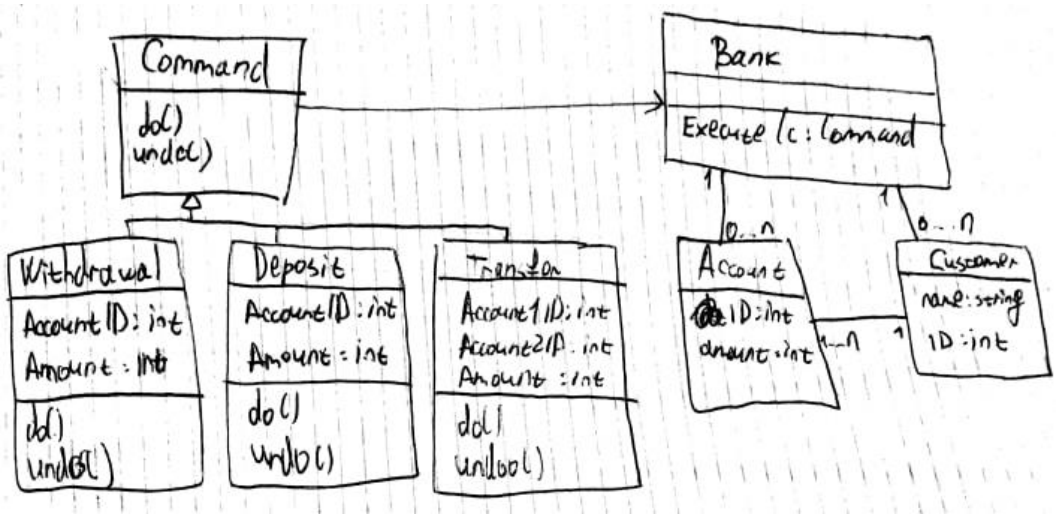


36. Objasniti *command* projektni obrazac.

Enkapsulira zahteve u objekte i omogućava njihovo smeštanje u red, pamćenje zahteva i realizaciju raznih funkcija nad zahtevima. Takođe razdvaja objekte koji pozivaju operaciju od objekata koji znaju kako da izvrše tu operaciju. Invoker klasa na dijagramu je ona koja inicira pokretanje akcije a komanda je ta koja zna kako da je izvrši.



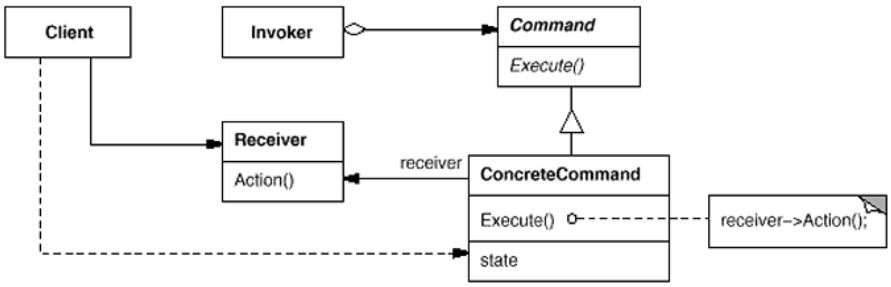
Klasa Command je apstranktna klasa i ona grupno opisuje sve komande koje se mogu javiti u sistemu, ima funkciju execute() koja opisuje izvršenje komande. Na ovaj način obezbeđeno je izuzetno lako dodavanje novih komandi u sistem (samo dodamo neku konkretnu komandu i mi smo već proširili sistem). Klasa ConcreteCommand izvedena je iz klase Command i nije apstraktna, ona opisuje neku konkretnu komadu u sistemu, koja sada zna na koji se način treba izvršiti pa predefiniše apstraktnu funkciju Excecute() iz svoje nadklase, ona ima referencu na objekat tipa Reciever koji trpi izvršenu komandu, odnosno izvršenje funkcije Execute podrazumeva zapravo izvršenje funlcije Action() nad objektom klase Reciever.



Može se primeniti u sistemu banke, samoj banci je oduzeta odgovornost da brine o tome kako tačno treba svaka od komandi da se izvrši već samo vrši funkciju execute() nad objektom klase Command iz koje su onda izvedene sve konkretne klase komandi koje se mogu izvršiti i koje onda brinu o konkretnom načinu svog izvršenja. Pored oslobođenja banke ove odgovornosti postigli smo i izuzetno lako proširenje funkcionalnosti. Ako samo kreiramo neku novu komandu i definišemo kako ona radi već smo i banci obezbedili njeno korišćenje bez potrebe menjanja samog objekta klase Banka.

Ovaj obrazac enkapsulira zahteve u objekte i na taj način omogućava parametrizovanje klijenata sa različitim zahtevima, organizovanje reda zahteva, logovanje zahteva, omogućavanje *undo* operacije i slično. Obično se koristi kada nam je potrebno da odradimo neki zahtev ka objektu, ali bez ikakvog znanja o operaciji koja se izvrši ili o znanju o tome koji se objekat poziva da se ona izvrši. Njime razdvajamo objekat koji pozivamo da izvrši neku komandu i objekat koji zapravo zna kako da izvrši tu komandu. Prema GoF, ovaj obrazac spada u domen objekata sa namenom ponašanja.

Učesnici:

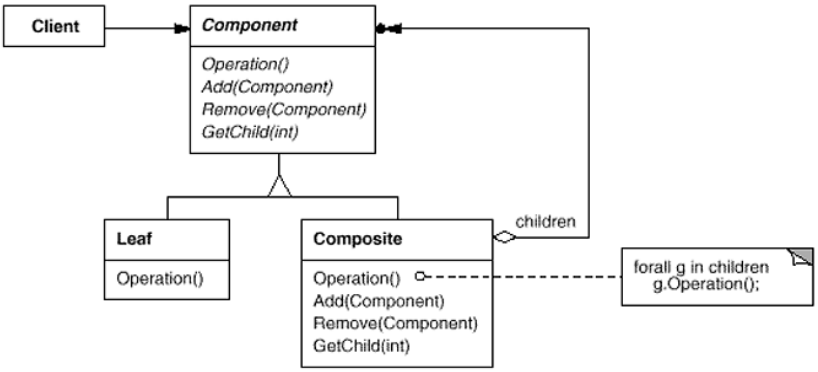


- Komanda – interfejs za izvršenje neke operacije
- Konkretna komanda – implementira određenu operaciju koja se izvršava nad primaocem
- Klijent – kreira konkretnu komandu i postavlja njenog primaoca
- Pozivač – poziva komandu na izvršenje
- Primalac – zna kako da izvrši zadatu komandu

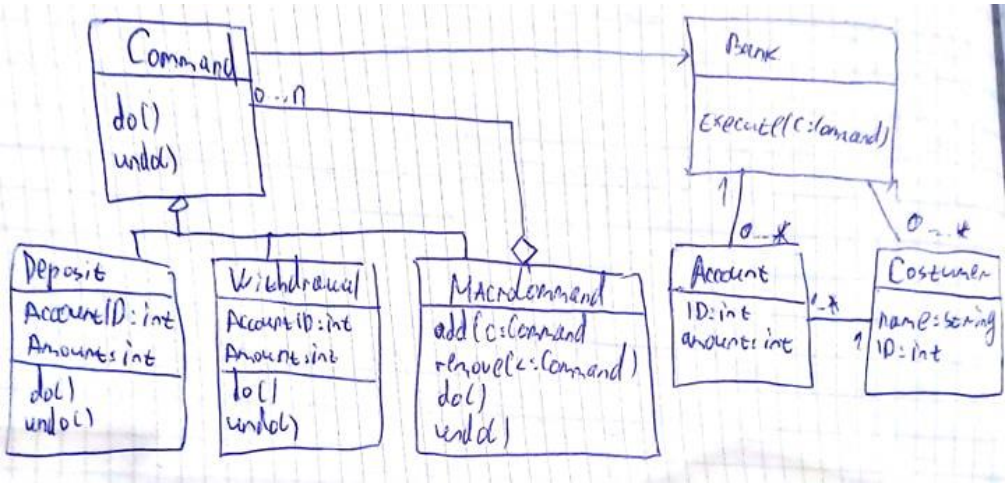
Konkretan primer primene ovog projektnog obrasca bi bio u pravljenju nekog okvira za korisnički interfejs gde imamo neku apstraktnu klasu *command* i iz nje izvodimo sve klase koje služe izvršenju konkretnih komandi, a samu tu komandu pozivamo kroz neko dugme koje se nalazi na meniju ili preko neke prečice (oba ova načina zapravo čuvaju instancu na isti objekat).

37. Objasniti *composite* projektni obrazac.

Ogranizuje objekte u strukturu stable i omogućava predstavljanje hijerarhije tipa celina-deo. Omogućava istovetno tretiranje individualnih i kompozitnih objekata.



Iz klase Component izvode se klase Leaf – koja predstavlja individualne objekte i Composite koja predstavlja kompozitne objekte koji kao komponente mogu da sadrže i elemente tipa Leaf i tipa Composite, to je obezbeđeno vezom agregacije između Composite i Component.



Može da se iskoristi za rešenje problema komande transfera novca u banci jer već kao komande postoje povlačenje novca sa nekog računa i uplata novca na neki račun, zašto bismo onda duplirali kod pišući iste te funkcionalnosti u okviru transfera novca. Postojanjem klase MacroCommannd omogućeno je da se jedna komanda sastoji od drugih komandi i time se rešava ovaj problem. I generalno se olakšava dodavanje novih komandi i omogućava kreiranje raznih komandi koje se sastoje od drugih komandi.

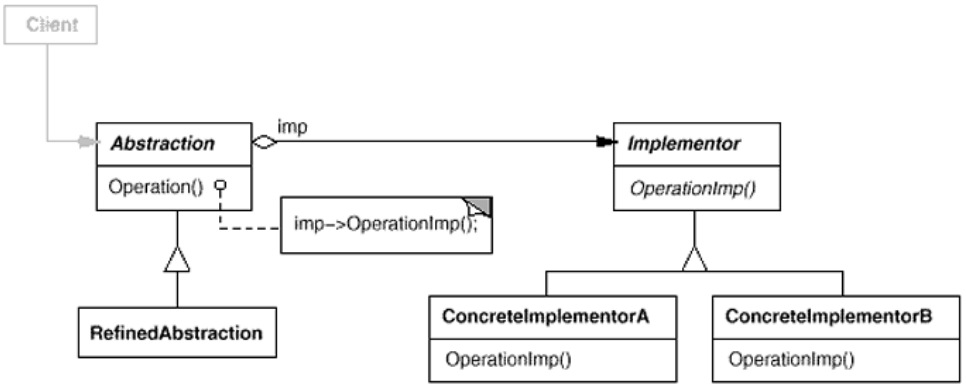
Učesnici u ovom obrascu su:

- Komponenta – interfejs objekata koji čine kompoziciju koji sadrži osnovno ponašanje zajedničko svim komponentama sistema (uključujući i metode za pristup deci komponentama)
- List – definiše ponašanje primitivnog objekta kompozicije i nema decu
- Kompozit – definiše ponašanje komponente koja ima decu, metode za rad nad decom komponentama i čuva reference na tu decu
- Klijent – manipuliše objektima kompozicije kroz interfejs komponente

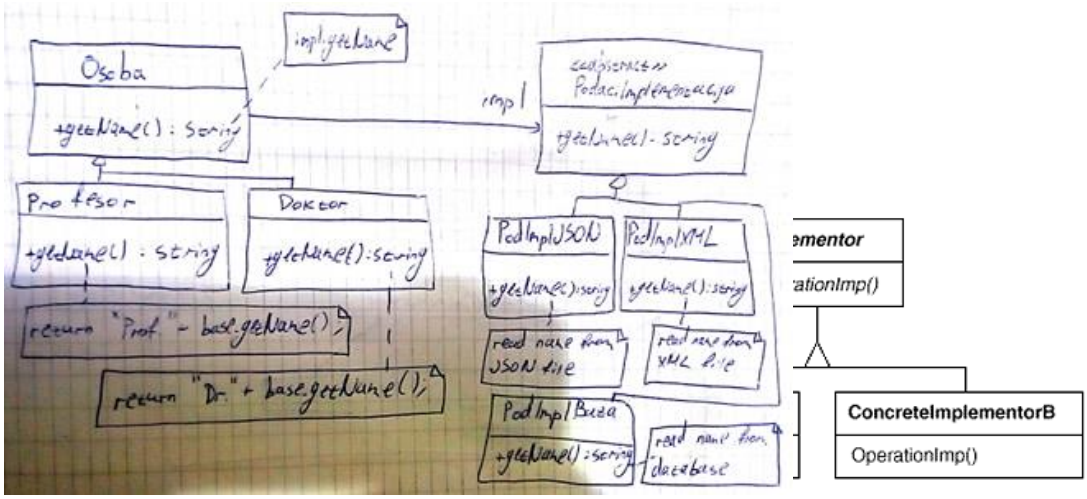
Najčešća primena ovog obrasca je u programima za grafičku obradu, gde je list zapravo neka primitiva poput linije, tačke i slično, dok je kompozit zapravo neka slika koja se sastoji od tih primitiva.

38. Objasniti *bridge* projektni obrazac i dati primer korišćenja.

Cilj ovog projektnog obrasca je da omogući što optimalniju realizaciju slučaja kada postoji hijerarhijska struktura klasa i hijerarhijska struktura implementacija. Umesto naivnog rešenja u kome se ove dve hijerarhijske strukture prepliću i kreiraju jedan ogroman, trom hijerarhijski sistem koristi se rešenje u kome se ove dve hijerarhijske strukture posmatraju odvojeno ali se obezbeđuje veza između njih. To dovodi do toga da možemo zasebno da menjamo implementacioni deo i deo u kome smo modelovali klase bez da to utiče jedno na drugo. Takođe nam omogućava da imamo veliki broj različitih implementacija čije se funkcionalnosti pozivaju preko zajedničke nadklase, u ovakvom sistemu je dodavanje nove implementacije izuzetno lako.



Primer primene je npr. Pri čuvanju podataka o osobama, možemo zasebno da kreiramo hijerarhiju osoba (student, profesor, dete, roditelj, ...) i da zasebno vodimo računa o različitim implementacijama čuvanja podataka (XML format, JSON format, zapis u bazu podataka, ...) i da možemo efikasno da čuvamo i čitamo podatke na bilo koji način o svim različitim vrstama osoba.



Ovaj obrazac služi za odvajanje apstraktne klase i njene konkretne implementacije kako bi omogućio njihovo zasebno menjanje. Na ovaj način promena u kodu na strani implementacije neće zahtevati promenu koda na strani apstrakcije. Takođe će jedna apstrakcija moći da ima gomilu implementacija i da im svima pristupi po potrebi. Prema GoF, ovaj obrazac ima strukturnu namenu u domenu objekta.

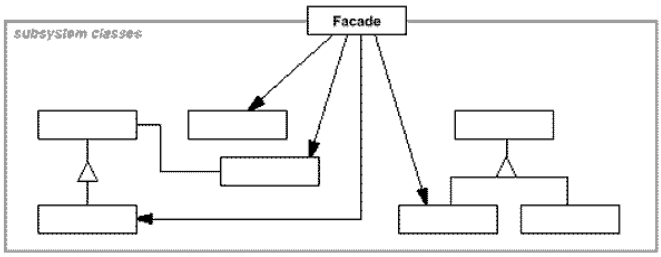
Učesnici u ovom obrascu su:

- Apstrakcija – definiše interfejs apstrakcije i održava referencu na implementacije
- Rafinisana apstrakcija – apstrakcija koja dalje proširuje osnovnu apstrakciju
- Implementacija – definiše interfejs za implementacijske klase koji ne mora da bude isti kao i interfejs apstrakcije; apstrakcija treba da definiše interfejs višeg nivoa, dok implementacija definiše osnovne operacije nižeg nivoa na koje se oslanja apstrakcija
- Konkretna implementacija – definiše konkretne operacije

Ovaj obrazac se može iskoristiti kod iscrtavanja prozora, gde u zavisnosti od operativnog sistema bismo koristili odgovarajuće klase. Apstrakcija bi imala neke generalne metode koje su recimo ispiši naziv prozora ili promeni boju prozora, dok bi u implementaciji bile odgovarajuće metode za odgovarajući operativni sistem i njegove klase koje rade ove stvari.

39. Objasniti *facade* projektni obrazac i dati primer korišćenja.

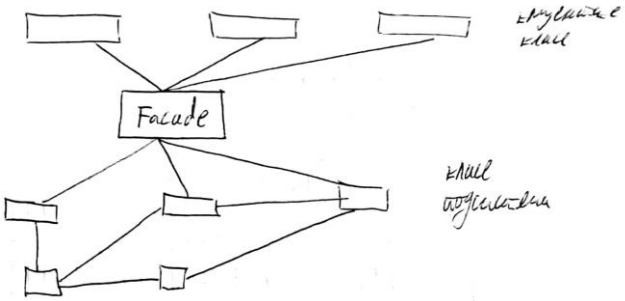
Ovaj obrazac se koristi kako bismo razdvojili podsistem od klijenata koji koriste usluge podsistema i na taj način im omogućili jednostavan interfejs ka podsistemu. Ovako dobijamo implementaciju koja je kompletno sakrivena od klijenata, a takođe i dobijamo jednostavniju organizaciju sistema po slojevima. Ovaj obrazac u neku ruku predstavlja i neku vrstu API-ja. Prema GoF, ovaj obrazac ima strukturnu namenu u domenu objekta.

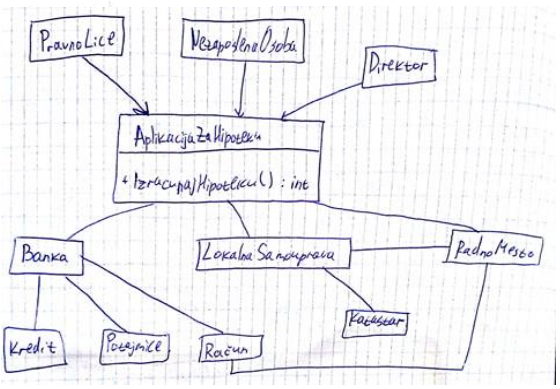


Elementi ovog obrasca su fasada koja omogućava pristup podsistemu klasa. Fasada implementira sve pristupne metode i tako obezbeđuje lako i jednostavno korišćenje i pozivanje klasa iz podsistema.

Ovaj obrazac se može koristiti kod softvera koji treba da obezbede jednostavan interfejs ka klijentu, recimo kod softvera za izračunavanje kreditne sposobnosti klijenta.

Projektni obrazac Facade koristi se u slučaju kada veliki broj klijentskih klasa treba da komunicira sa velikim brojem sistemskih klasa. Bez korišćenja ovog obrasca ovakav slučaj odlikuje velika zamršenost veza između brojnih klasa koje u njima učestvuju što bilo kakve promene u sistemu čini izuzetno teškim. Facade podrazumeva razdvajanje klijentskog dela od čitavog podsistema koji nudi usluge. Između njih nalazi se samo jedna pristupna tačka – objekat klase Facade koji klijentima nudi usluge podsistema u vidu intuitivnih funkcija koje oni mogu da pozivaju dok se implementacija ovih funkcija realizuje interno unutar samog podsistema. Na ovaj način je dodavanje klijenata izuztetno lako, a unapređenje funkcionalnosti podsistema dovodi samo do promena u načinu komunikacije između objekata i klasa iz podsistema ni na koji način ne utičući na klijenta.





Može da se iskoristi na primer u realizaciji aplikacije za računanje hipoteke. Klasa AplikacijaZaHipoteku pruža jedinstven i jednostavan interfejs svim tipovima klijenata i skriva čitav podsistem međusobdno povezanih klasa koje moraju da razmene informacije kako bi se izvršilo izračunavanje hipoteke. Klijent nema nikakvo znanje o tome, on jednostavno poziva funkciju IzracunajHipoteku() i očekuje da dobije rezultat.

40. Objasniti šta su to *anti-obrasci* i navesti neke primere njih.

Anti-obrasci su projektni obrasci koje bi trebalo izbegavati pri razvoju softvera. Možda nam se čini da ćemo na taj način napisati lakši i bolji kod, ali je praksa pokazala da je taj kod nestabilan i težak za održavanje.

Neki od njih su:

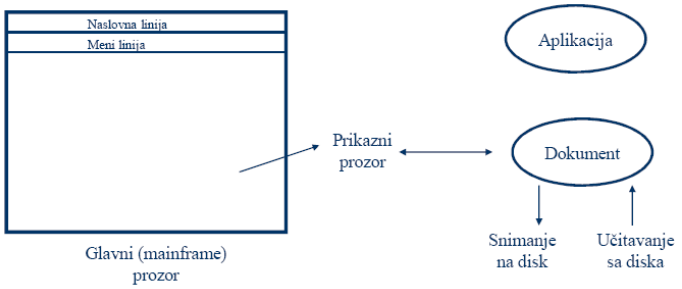
- DLL hell – Izuzetno veliki procenat koda se oslanja na eksterne DLL-ove u čiju sadržinu sam projektant nije upoznat već samo zove funkcije ponuđene u interfejsu. Može da se desi da neke od ovih biblioteka ostanu bez podrške što onda dovodi do pada čitave aplikacije i primorava projektanta da ili prepiše čitav deo koda koji je izgubio podršku ili izmeni čitav sistem tako da se uklopi u neki nov sistem biblioteka koji je našao kao zamenu za onaj koji je izgubio podršku
- Had-kodiranje – hard-kodiranje podrazumeva direktno unošenje numeričkih vrednosti u kod, npr. Za broj elemenata u nizu, granicu niza, ... Ukoliko dođe do promene u ovoj vrednosti projektant bi morao da uđe u sam kod i u njemu direktno da vrši promene što nije dobra praksa. Mnogo je bolje rešenje da se takav podatak čuva eksterno u nekom fajlu iz kojeg se onda učitava po potrebi, ukoliko dođe do promene možemo samo da zamenimo vrednost u fajlu i aplikacija će pri pokretanju iz njega da pročita ažuriranu vrednost.
- Špageti kod – Javlja se kada se pri pisanju koda ne vodi računa o struktuiranosti i podeli koda na logičke celine, može da dovede do loše čitljivosti i uopšteno do poteškoća u izmeni i razumevanju koda.

41. Primeniti *abstract factory* obrazac za čuvanje liste najboljih rezultat kod neke igre. Potrebno je rezultate sačuvati u bazu podataka kao i u posebnoj XML datoteci.

Pojavljivanje: 2

Ovo teško da će da bude.

42. Šta je *MFC*? Objasniti arhitekturu 4 klasa kod MFC baziranih aplikacija



MFC je C++ API za programiranje Microsoft Windows programa. Objektno je orijentisan i enkapsulira veći deo Windows API-ja nudeći korisniku skup klasa preko kojih ima olakšan pristup različitim resursima koje nudi operativni sistem. Pored toga što predstavlja biblioteku, klasa MFC je takođe i framework za razvoj Windows aplikacija jer zahteva organizaciju u formi arhitekture 2 ili 4 klase.
Arhitektura 4 klase podrazumeva postojanje 4 osnovne klase:

- Mainframe klasa (izvedena iz CFrameWnd) – zadužena je za celokupni ekran aplikacije

- Aplikacijska klasa (izvedena iz CWinAPP) – „pozadinska“ klasa koja obezbeđuje ostale neophodne funkcije koje nisu u direktnoj vezi sa prikazom
- Prikazna klasa (izvedena iz Cview ili njenih derivata) – zadužena je za celokupnu radnu oblast mainframe prozora (za dete-prozor glavnog prozora)
- Dokumentna klasa (izvedena iz CDocument) – „pozadinska“ klasa koja je zadužena za pamćenje stanja aplikacije i centralna je u procesu snimanja i učitavanja dokumenata

43. Šta je *MFC*? Objasniti interakciju MFC-a i Windows operativnog sistema.

MFC, Micorosoft Foundation Classes, je C++ API za programiranje Microsoft Windows programa koji u isto vreme predstavlja i objektno-orijentisani okvir oko Win32 API-ja. On obezbeđuje skup klasa koje olakšavaju programiranje Windows aplikacija i enkapsulira veći deo Windows API-ja.

MFC interaguje sa Windows operativnim sistemom kroz pozive Windows API-ja koji čine User modul (zadužen za rad sa ulaznim uređajima), GDI modul (zadužen za rad sa izlaznim uređajima) i Kernel modul (zadužena za rad sa datotekama i upravljanje memorijom). MFC klase i aplikacije pozivaju ove module i njihove funkcije po potrebi. Ove tri komponente su realizovane kao DLL-ovi i nalaze se u sistemskom direktorijumu. Kada iz spoljašnjeg sveta dođe poruka, MFC čita poruke i proverava da li je upućena nekoj aplikaciji. Ako jeste, poruka se prosleđuje aplikaciji koja je čita i izvršava naredbu eventualno pozivajući Windows API. Ukoliko nije, poruka se prosleđuje Windows OS-u.

44. Navesti vrste prozora kod MS Windows operativnih sistema i objasniti svaki od njih.

Postoje dve osnovne grupe prozora (osnovna klasa je CWnd):

1. Kontejnerski prozori (container windows) – prozori koji unutar sebe mogu sadržati druge prozore i služe za kontrolu i upravljanje nad tim prozorima koje sadrže. Oni obezbeđuju osnovnu strukturu korisničkog interfejsa. Frame prozor (CFrameWnd i CMDIFrameWnd) je vrsta kontejnerskog prozora i predstavlja glavni prozor aplikacije. Dijalog prozori (osnovna klasa CDialog) su takođe podvrsta kontejnerskih prozora, oni unutar sebe sadrže samo kontrole za dijaloge.
2. Prozori za podatke (data windows) – sadržani su u frame ili dijalog prozorima i zaduženi su za unos podataka od strane korisnika preko kontrolnih barova (CStatusBar, CToolBar, CDialogBar), view prozora (grafička reprezentacija i izmena skupova podataka preko CView klase tj. CScrollView ili CRecordView) i dialog box kontrola (kontrole za unos teksta, za razne boksove i slično)

45. Objasniti koncept poruke kod Windows/MFC aplikacija.

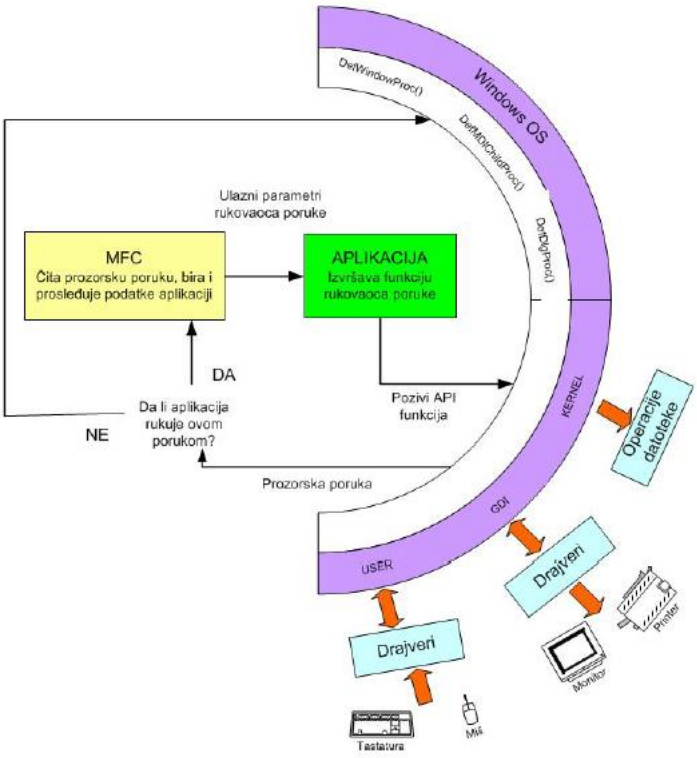
Poruka predstavlja centralni koncept u Windows-u jer kroz poruke se dešavaju sve akcije prozora ili aplikacija. Poruka je zapravo neki događaj na koji aplikacija odgovora, a u MFC aplikacijama se definišu rukovaoci porukama koji prihvataju određeni tip poruke i definišu ponašanje aplikacije za taj tip poruke. Pri prijemu poruke, izvršava se kod funkcije koja rukuje tom porukom.

46. Objasniti strukturu poruka kod MS Windows operativnih sistema i objasniti svaku od komponenti.

Poruka je struktura koja sadrzi sledece podatke:

- hwnd – identifikator prozora kome je poruka namenjena
- message – tip poruke definisan u zaglavlju windows.h; Određuje tip poruke. Za svaki tip poruke u zaglavlju windows.h je definisana konstanta C preprocesora. Većina konstanti za poruke koje šalje operativni sistem počinje sa WM_ što predstavlja skraćenicu od Windows Message.
- wParam i lParam– dodatne informacije o poruci čije tačno značenje zavisi od tipa poruke
- time – Određuje vremenski trenutak u kome je poruka poslata
- pt – Određuje poziciju kursora, u ekranskim koordinatama, odakle je poruka poslata

47. Nacrtati dijagram koji objašnjava interakciju MFC-a i Windows-a i objasniti ga.



Operativni sistem se sastoji iz 3 dela:

- User
- GDI (Graphics Device Interface)
- Kernel

User je modul koji je zadužen za rad sa ulaznim uređajima.

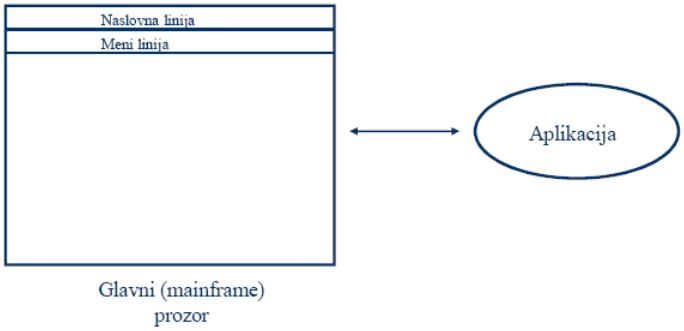
GDI je modul koji je zadužen za rad sa izlaznim uređajima.

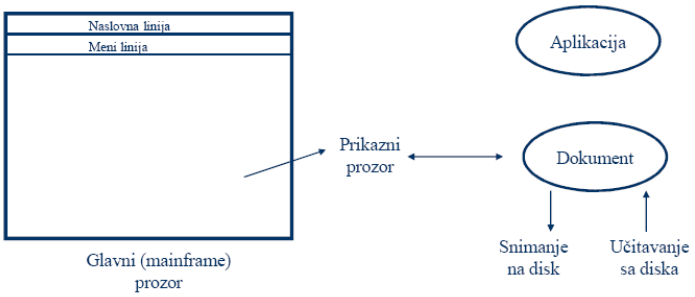
Kernel je modul koji je zadužen za rad sa datotekama i za upravljanje memorijom.

Ove 3 komponente zajedno čine Windows API i realizovane su kao DLL-ovi koji se nalaze u sistemskim direktorijumima operativnog sistema. MFC interaguje sa Windows operativnim sistemom kroz pozive Windows API-ja. MFC aplikacija i MFC klase pozivaju funkcije API-ja. Kada se javi neka prozorska poruka proverava se da li aplikacija kojoj je namenjena ima hendler za tu poruku. Ako nema poruka se prosleđuje operativnom sistemu koji onda izvršava default handler rutinu. Ako ima onda se ta poruka obrađuje u okviru MFC-a koji odgovarajuće podatke prosleđuje aplikaciji koja onda po potrebi potrebi poziva odgovarajuće API funkcije operativnog sistema.

48. Objasniti arhitekture 2 i 4 klasa MFC aplikacija.

Arhitektura 2 klase se oslanja na CFrameWnd i CWinApp klase (mainframe i aplikacijska klasa) i njihovu međusobnu komunikaciju. Mainframe je zadužena za celokupni ekran aplikacije, dok je aplikacijska klasa zadužena za pozadinske klase koje obezbeđuju funkcionalnost.





Arhitektura 4 klasa se oslanja na 4 klase MFC-a: CFrameWnd (mainframe klasa), CWinApp (aplikacijska klasa), CView (prikazna klasa) i CDocument (dokumentna klasa. Mainframe i aplikacijska klasa rade isto što i u arhitekturi 2 klase. Prikazna klasa je zadužena za dete-prozor glavnog prozora, dok je dokumentna klasa zadužena za pamćenje stanja aplikacije i centralna je u procesu snimanja i učitavanja dokumenta.