1. NAČELA PROGRAMSKOG OBLIKOVANJA

Notebook: OOUP

1. NAČELA PROGRAMSKOG OBLIKOVANJA

-> organizacijska pravila koja mogu zauzdati probleme vezane uz složenost organizacije među komponentama

organizacija određuje dinamička svojstva programa te sposobnost projekta za održivi razvoj zbog loše definiranih početnih zahtjeva, te činjenice da se zahtjevi s vremenom mijenjaju, program je teško organizirati kako treba iz prve

-> rješenje: organizaciju postupno usklađivati sa saznanjima o domeni

- programsko oblikovanje svodimo na traženje brzih i odgovarajućih odgovora na promjene okolnosti

SIMPTOMI

-> ukazuju na neprikladnu organizaciju programa

4 simptoma

- **krutost**
  - teško nadograđivanje uslijed krutosti postojećeg koda
  - kada je program krut?
    - kada lokalne promjene u jednom dijelu koda zahtjevaju daljne promjene u ostatku programskog koda (domino efekt)
      - zbog lanaca međusobnih ovisnosti
      - A ovisi o B, B ovisi o C, C ovisi o D.. sve do Z
      - onda promjena u Z direktno utječe na A
  - posljedica može biti strah od ispravljanja problema koji nisu kritični
  - **rješenja**
    - kraćenje lanaca ovisnosti primjenom **apstrakcije i enkapsulacije**
- **krhkost**
  - lako unošenje suptilnih pogrešaka zbog krhkosti organizacije
  - kada je programski sustav krhak?
    - kada je sklon pucanju pod pritiskom promjenivih zahtjeva i kratkih rokova
    - nastaju uslijed implicitne međuovisnosti uzrokovane ponavljanjem
    - jedna izmjena mora se uvesti na više mjesta
    - ponavljanje treba izbjegavati gdje god je to moguće
  - propusti kod ovakvog sustava ne uzrokuju probleme pri prevođenju, nego samo probleme pri izvođenju
- **nepokretnost**
  - teško višestruko korištenje postojećih komponenata
    - kad je lakše napisati novi kod nego koristiti postojeći
  - čest uzrok
    - pretjerana međuovisnost zbog neadekvatnih sučelja i neadekvatne razdiobe funkcionalnosti po komponentama
  - potiče ponavljanje, a time krhkost i krutost
- **viskonznost (trenje)**

    - intervencije koje čuvaju integritet programa zahtjevaju puno manualnog rada
    - **2 vrste**
        - **viskoznost programske organizacije**
            - znači da je program teško nadograđivati uz očuvanje oblikonvnog integriteta
            - kod takvih programa, novu funkcionalnost najlakše je dodati na dugoročno loš način
        - **viskoznost razvojnog procesa**
            - odnosi se na sporu i neefikasnu razvojnu okolinu

TEHNIKE

logičko vs. fizičko oblikovanje

- **logičko oblikovanje**
    - raspored funkcionalnosti programa po razredima i funkcijama
- **fizičko oblikovanje**
    - raspored funkcionalnosti po datotekama izvornog koda
    - komponenta je temeljna jedinica
        - sastoji se od sučelja (header -> .hpp) i implementacije - .cpp, .lib, .a, .dll, .so itd
        - sadrži jednu ili više logičkih jedinica
        - temeljna jedinica pri verziranju i testiranju

koristimo dijagrame OMT (prethodnik UML, jednostavniji)

    - koristmo ih za opis logičkih odnosa
        - izvođenje, referenciranje, sadrži, stvara...
        - https://youtu.be/81Xu0SrNuhU?list=PLkOLgurQ4FfMl7lhtSTImxdbe0o5RRtpo&t=177

za fizičku organizaciju koristimo hibridnu notaciju

- u sebi sadržavaju i logičku organizaciju
- komponenta A ovisi o komponenti B ako se A ne može ni prevesti ni ispitati bez B

slajd iz predavanja o polimorfizmu je važan, nema ga smisla cijelog kopirat pa ga samo pogledaj (slajd 29)

dinamički polimorfizam

- virtualne funkcije i njihove tablice u C++

statički polimorfizam

- odluku o odredištu poziva povjeriti prevoditelju
- predlošci u C++ (donekle slični <T> u Javi)
- prevođenje se odgađa do trenutka do kad parametri postaju poznati
- predlošci prikladni za manje, često korištene programske jedinice
- nedostatak: ukoliko želimo parametriziranu metodu pozvati s nekim drugim tipom, onda moramo ponovno pokrenuti prevođenje

OOP

- razmatra dinamiku sustava
    - kako postići da kod koji pišemo danas radi ispravno s kodom koji pišemo iduće godine?
    - pitamo se što će se mijenjati u budućnosti
    - na temelju te procjene pokušavamo se zaštititi od promjena
- za razliku od struktuiranog programiranja koje se jedino pita "što sve treba implementirati?"
- **ugovorno oblikovanje**
    - oblikujemo komponente koje surađuju ispunjavanjem obaveza definiranim eksciplitnim ugovorima
    - terminologija
        - ako komponenta **A** ovisi (poziva, referencira, stvara...) o komponenti **B,** onda je **A** klijent, a **B** pružatelj
    - 2 osnovna elementa ugovora između klijenta i pružatelja
        - **preduvjeti**
            - garantiraju primjenjivost komponente
                - (reguliraju obaveze klijenta prema pružatelju)
        - **postuvjeti**
            - garantiraju ispravnost rezultata
                - (reguliraju obaveze pružatelja prema klijentu)
    - za automatsko testiranje ugovora -> assertovi
        - nisu isti kao if -> ukoliko uvjet nije ispunjen, izvođenje se završava
        - moguće ih je isključiti prilikom prevođenja tako da kasnije ne utječu na prevođenje koda

ortogonalnost

- nema međuovisnosti algoritama i spremnika
- npr. algoritam standardne knjižnice reverse možemo zvati nad vektorom, poljem i listom

LOGIČKA NAČELA (SOLID principles of OOP)

načelka oblikovanja elemenata logičke organizacije (razredi i funkcije, ne datoteke)

pamtimo ih po kratici **SOLID**

- **1) Single responsibility principle (načelo jedinstvene odgovornosti)**
    - komponente modeliraju koncepte koji imaju jasnu odgovornost
    - najčešće se interpretira na način da klase trebaju imati samo jednu odgovornost
- **2) Open-close principle (načelo nadogradnje bez promjene)**
    - lako je dodati novu funkcionalnost (open dio), bez utjecaja na postojeću funkcionalnost tj. kod (close dio)
    - najvažnije, osnova za ostale
- **3) Liskov's substitution principle (načelo nadomjestivosti osnovnih razreda)**
    - ako je B izveden iz A, na svim mjestima gdje možemo koristiti A moramo moći koristiti i B
    - npr. tko god zna voziti auto, zna voziti i Fiat Punto
- **4) Interface segregation principle (načelo izdvajanja sučelja)**
    - ne tjerati klijente (one koji implementiraju sučelja) da moraju implementirati funkcionalnosti sučelja koje im neće trebati
- **5) Dependency inversion principle (načelo inverzije ovisnosti)**
    - kod više razine ne bi trebao ovisiti o implementaciji koda niže razine
    - usmjeravanje ovisnosti prema apstraktnim suleljima

NADOGRADNJA BEZ PROMJENE (NBP)

- **iliti Open-Close principle**

uči nas kako organizirati program tako da može primiti novu funkcionalnost bez izmjene postojećeg izvornog koda

cilj -> funkcionalnost komponente možemo proširiti bez mijenjanja njene implementacije

- tako postižemo fleksibilnost
    - nadogradnja ne utječe na klijente
- **ideja: stari kod radi s novim kodom**

2 pristupa za ostvarenje NBP

- **1) nasljeđivanje implementacije**
    - novi razredi pozivaju temeljnu implementaciju nasljeđivanjem starog razreda
    - nema polimorfnih poziva
    - postojeći klijenti ne mogu doći do nove funkcionalnosti
    - novi kod poziva stari kod -> boring, ne zanima nas previše
- **2) nasljeđivanje sučelja**
    - oslanja se na polimorfizam
        - bilo statičkim, bilo dinamičkim (najčešće)
        - statički se često koristi za spremnike (npr. u Javi ArrayList<T>)
    - klienti pristupaju novoj implementaciji preko starog sučelja
    - tj. samo trebamo osigurati da nova implementacija slijedi staro sučelje
        - onda i stari kod može poslati novi, jer zna da implementira to sučelje s kojim on zna raditi
        - a za konkretnu implementaciju tog sučelja ga onda nije briga
    - **stari kod poziva novi kod!**

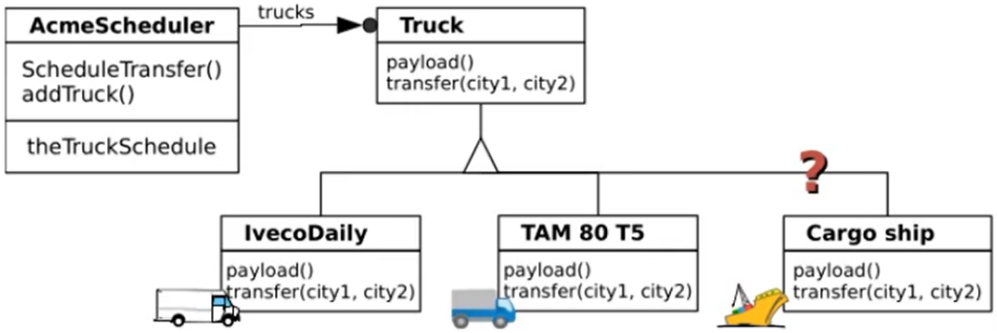
koncepti za pospješivanje NBP

- **enkapsulacija**
    - klijenti razreda ne smiju izravno referencirati podatkovne članove razreda
        - -> svi podatkovni članovi bi trebali biti privatni
- **virtualne funkcije**
    - moguće pozivanje modula napisanih godinama nakon klijenta
    - npr. imamo klasu Shape sa virtualnom metodom draw
        - i onda kasnije recimo napišemo implementaciju razreda (sučelja) Shape -> Square
        - koje nudi svoj kod za virtualnu funkciju draw
        - i na taj način nam virtualne funkcije i polimorfizam pomažu, jer onda neki kod za crtanje može primiti referencu na polje Shapeova i samo pozvati njihovu metodu draw
        - to što su ti objekti ustvari Square ga nije briga, jer on gleda kroz naočale razreda Shape
    - stari kod zove novi kod
- **apstraktni razredi**
    - nemaju podatkovnih elemenata -> enkapsulirani
    - imaju virtualne funkcije
- **statički polimorfizam**
    - injekcija novog koda u stari pri prevođenju
    - npr. napišemo naš novi razred A i parametriziramo ArrayList po A -> ArrayList<A>
    - kod ArrayList, koji je nastao prije našeg razreda A, radi s tim novim kodom

ovo načelo se najbolje postiže ako primijenimo pomoćna načela, opisana u nastavku

LISKOVINO NAČELO SUPSTITUCIJE (Liskov substitution principle)

najkraća definicija: **osnovni razredi moraju se moći nadomjestiti razredima izvedenim iz njih**

- dakle, ako imamo klasu Shape, i klasu Elipse izvedenu iz nje, onda ovo načelo kaže da se svugdje u kodu gdje se koristi Shape, mora moći koristiti i Elipse
    - naravno, obrnuto ne vrijedi
- npr. također, tko god zna voziti auto, zna voziti i Toyotu Auris
- 

    - konkretniji primjer -> svi podrazredi razreda Truck moraju moći surađivati sa razredom AcmeSchedulerom. koji samo zna pričati sa objektima tipa Truck
        - onda IvecoDaily. TAM i Cargo ship gledamo kroz naočale razreda Truck

načelo upućuje na pravilnu upotrebu nasljeđivanja

- dakle nasljeđivanje treba modelirati relaciju **IS_A_KIND_OF**
    - **izvedeni razredi moraju poštovati ugovore osnovnog razreda**
        - 1) preduvjeti izvođenju metoda izvedenih razreda moraju biti jednaki ili slabiji onima u osnovnom razredu
        - 2) postuvjeti moraju biti isti ili postroženi
            - primjer sa razredima Bird i Penguin u prezentaciji
        - tldr ovo dvoje -> **izvedeni razredi moraju imati barem istu primjenjivost, ali mogu imati i širu**
- izvedeni razred krši LNS ako
    - neki klijent koji ok radi s osnovnim razredom ne može raditi s izvedenim razredom
    - javlja se simptom **patologije**
        - klijenti moraju ispitivati rade li s izvedenim razredom ili ne
        - ovaj simptom krši načelo nadogradnje bez promjene -> moramo mijenjati klijenta
    - kršenje principa često posljedica slabog znanja o domeni

popravak kršenja LNS

- 1) smanjiti odgovornosti osnovnog razreda
- 2) povećati odgovornosti izvedenog razreda
- 3) odustati od izravnog nasljeđivanja

LNS i statičkin tipizirani jezici (Java, C++, C#)

- LNS -> recept za korištenje nasljeđivanja
- nasljeđivanjem modeliramo nadomjestive tipove
    - klijenti koriste izvedene razrede preko osnovnog sučelja
- za reusanje funkcionalnosti -> kompozicija

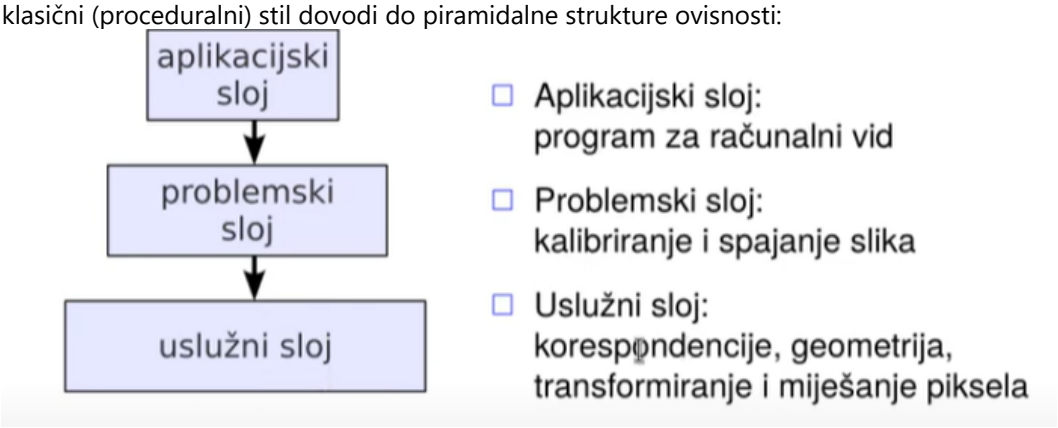
LNS i implicitno tipiziranje (Python)

- tip argumenata funkcija i atributa razreda nije statički određen
- nadomjestive tipove možemo graditi i bez nasljeđivanja
    - nasljeđivanje koristimo manje nego u statički tipiziranim jezicima
- LNS izražavamo pomoću **nadomjestivosti**
    - klijentima ne valja slati nenadomjestive inačice poslužitelja
    - ako klijent treba raditi s različitim pružateljima, pružatelji moraju biti međusobno nadomjestivi
        - inače klijent ne poštuje NNBP

NAČELO INVERZIJE OVISNOSTI (Dependency Inversion Principle)

implikacije NNBP i LNS vode na načelo inverzije ovisnosti

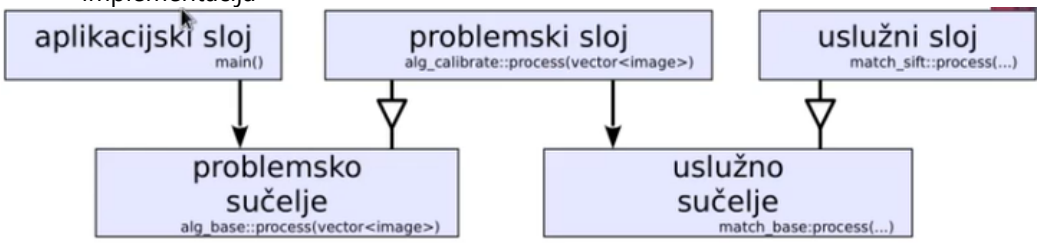
klasični (proceduralni) stil dovodi do piramidalne strukture ovisnosti:



- problem sa ovakvom struktruom ovisnosti
    - moduli više razine ovise o izvedbenim detaljima - modulima niže razine
        - ne možemo ni prevesti ni ispitati module više razine ako oni niže razine nisu dovršeni
    - pri mijenjanju modula niže razine često se javlja domino efekt -> promjene utječu i na module više razine

ideja: main neće zvati konkretne algoritme

- on i neće znati s kojom točno implementacijom on radi
- on zove metodu sučelja
    - koja onda ima konkretnu implementaciju, ali main nije briga koja je to implementacija



- **ovisnosti idu prema apstrakcijama - sučeljima**
    - sučelja imaju malo razloga za čestu promjenu, jer nemaju implementaciju
    - ne **ovisimo** više o konkretnim modulima sa svojim detaljnim implementacijama, nego o apstrakcijama, sučeljima -> ta ovisnost je invertirana
    - glavni program ne zna koja se konkretno implementacija poziva
    - prije je main ovisio o gigantskim modulima sa detaljnim implementacijama, a sad ovisi samo o malom sučelju

načelo kaže

- ovisnosti u projektu trebaju ići prerma apstrakcijama
- NE od modula visoke razine prema modulima niske razine
- jer se komponentre bez implementacije rjeđe mijenjaju
- ovo omogućava nadogradnju bez promjene
    - lako promijenimo koji algoritam implementira to naše sučelje
        - jer opet, main nije briga koja je konkretna implementacija, samo da je sučelje zadovoljeno

ali, ne treba pretjerivati

- ovisnost o modulima za koje znamo da se ne mijenjaju je OK (recimo komponente standardne biblioteke programskog jezika)

problem -> stvaranje objekata konkretnih razreda

- tko je zadužen za to?
    - stvaranje implicira ovisnosti
- neka rješenja
    - injekcija ovisnosti, obrasac tvornica

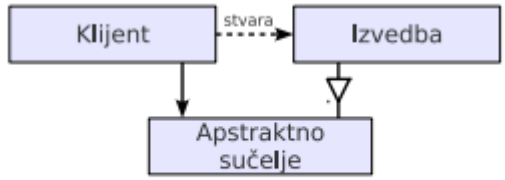
injekcija ovisnosti

- ne hardkodiramo član (tj. ne stvaramo ga), nego ga konfiguriramo preko reference na osnovni razred (sučelje), a konkretni objekt primimo od pozivatelja

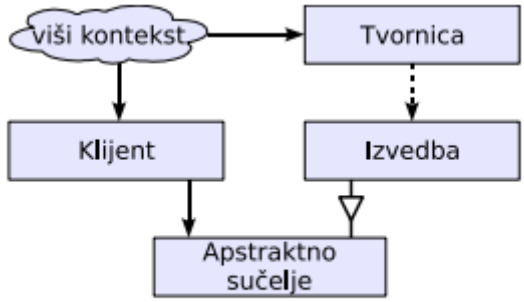
```
// without dependency injection
class Client1 {
  ConcreteDatabase myDatabase;
public:
  Client1():
    myDatabase() {}
public:
  void transaction() {
    myDatabase.getData();
    // ....
  }
};
```

```
// with dependency injection
class Client2 {
  AbstractDatabase& myDatabase;
public:
  Client2(AbstractDatabase& db):
    myDatabase(db) {}
public:
  void transaction() {
    myDatabase.getData();
    // ...
  }
};
```

-
    - Client1 u svom konkstruktoru stvara primjerak konkretnog razreda ConcreteDatabase
    - Client2 nema referencu na konkretnu implementaciju neke baze, nego ima samo referencu na apstraktni razred AbstracDatabase
        - i onda u svom konstruktoru, od metode koja ga poziva, prima referencu na konkretnu implementaciju baze i nju onda koristi
        - ali on nema pojma koja je to implementacija, samo zna da zadovoljava AbstractDatabase



1. bez injekcije ovisnosti

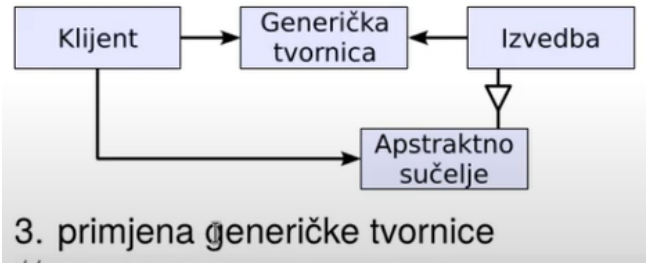


2. injekcija ovisnosti + tvornica

naravno, u 2 i dalje netko (viši kontekst) ovisi o konkretnoj izvedbi, ali to nije naš dio koda s kojim se trenutno bavimo

- mislim, viši kontekst može biti recimo Spring, koji ti povezuje cijelu aplikaciju i brine se za takve stvari autowiranjem

3. rješenje je korištenje generičke tvornice -> tada nitko ne ovisi o konkretnoj izvedbi



obrazac tvornica

NAČELO JEDINSTVENE ODGOVORNOSTI (Single Responsibility Principle)

načelo kaže: **programski moduli moraju imati samo jednu odgovornost**

zašto je to korisno?

- ako 1 modul = 1 odgovornost
  - ako dođe do promjene (recimo želimo dodati novu funkcionalnost, maknuti postojeću), potrebno je promijeniti samo jedan modul
  - ali ako modul ima više odgovornosti, među odgovornostima se javljaju (neprirodne) međuovisnosti
    - krutost, nepokretnost

odgovornost

- 1 "kvatnt" funkcionalnosti iz aplikacije
- svaka odgovornost je još jedna opasnost od potrebe izmjene modula
- svaki modul - jedna odgovornost
  - koju - to mi trebamo otkriti
  - što bolje razumijemo domenu, to je lakše ovo otkriti

NAČELO IZDVAJANJA SUČELJA (Interface Segregation Principle)

nije dobro raditi jednu klasu (ili sučelje) sa miljardu odgovornosti

- to je tzv. swiss army knife anti-pattern
- zbunjujemo korisnike tog sučelja
- omogućavamo suvišne ovisnosti

najbolje je koristiti single responsibility koncept, ali ako ne možemo, ovo je alternativa

načelo kaže: **nekoherentnim konceptima klijenti trebaju pristupati preko izdvojenih sučelja**

- tj. **ne želimo da klijenti ovise o funkcionalnostima koje ne koriste**

primjer sa vratima iz prezentacije

- ne želimo da klijenti kojima ne treba automatsko zaključavanje vrata ovise o promjenama u npr. TImerClient
- rješenje je tu funkcionalnost automatskog zaključavanja vrata izdvojiti u zaseban razred - TimedDoor, i tamo ponuditi funkcionalnosti za automatsko zaključavanje vrata

ovo načelo propisuje kad koristiti višestruka nasljeđivanja

FIZIČKA NAČELA

moramo još osigurati lako ispitivanje

- ispitivanje najlakše organizirati nad datotekama izvornog koda
    - analiziramo fizičku organizaciju

ako imamo sustav od n komponenata

- je li bolje testirati odvojeno ili zajedno?
- dvije opcije
    - **inkrementalno testiranje komponenata (unit testing)**
        - uvijek bolji odabir!!
        - također, dobro je testirati rano, automatski, regresijski (kad se nađe bug i riješi, dodaje se test za taj bug)
        - zašto se ovo zove inkrementalno testiranje?
            - ključna relacija - ovisnosti
            - ako je ovisnost **aciklička** (komponentama možemo dodijeliti razine)
                - razine definiraju redoslijed inkrementalnog testiranja komponenti
                    - prvo testiramo komponente na najnižoj razini, pa na višoj itd.
                    - kad pređemo na višu razinu, moramo uključiti i komponente niže razine, ali pokušavamo da ih bude što manje
            - **sveobuhvatno testiranje cijelog sustava (big bang integration testing)**
                - zove se big bang jer se uvijek nešto sruši
                - ako imamo cikličku ovisnost, moramo testirati sve zajedno

povoljna fizička organizacija

- **plitka i nepovezana struktura ovisnosti**
    - kad testiramo veći broj komponenti, možemo testirati samo njih, ne i komponente niže razine
        - zato je inverzija ovisnosti dobra za testiranje, činin ovisnosti plićima

ciljevi fizičkog i logičkog oblikovanja su kompatibilni
struktura međuovisnosti je dobar pokazatelj organizacijske kvalitete

ciklusi

- ciklička struktura ometa ispitivanje, ponovno korištenje i razumijevanje
    - što je ciklus veći, veći je problem
- većina programa ima ciklus od bar 100 razreda (sjeti se debuggiranja u Javi - sve i svašta se poziva u pozadini)
- jedna od motivacija oblikovnih obrazaca - ukloniti cikličke ovisnosti

---

C++ crash course

- *datoteka .hpp*
    - *u njemu pišemo specifikaciju razreda (kao sučelje, sve što razred treba imati)*
- *datoteka .cpp*
    - *moramo includati .hpp datoteku*
    - *pišemo implementaciju*
- *Base& solver_ -> solver_ je referenca na razred Base*
    - *reference C++ iste kao pokazivači u C-u, osim što se za pristupanje članskim varijablama koristi operator . a ne ->*
- *konstruktor*
    - *specijalna metoda koja se uvijek poziva prilikom stvaranja instanci razreda*
- *class Derived: public Base*
    - *ovim smo rekli da razred Derived naslijeđuje razred Base*

poziv metoda koje nisu virtualne se implementiraju običnim funkcijskim pozivima

- *to je recimo metoda operate*

2. OBLIKOVNI OBRASCI

Notebook: OOUP

2. OBLIKOVNI OBRASCI

oblikovni obrasac - općenito rješenje oblikovnog problema

arhitektonski obrasci

- struktura programa na najvišoj razini
- npr. MVC, klijent-poslužitelj, P2P network..

oblikovni obrasci

- srednja razina apstrakcije (cca. 1000 linija koda)
- nisu toliko ovisni o jeziku

programski idiomi

- organizacija sastavnih dijelova programa (1-100 linija koda)
- ovisni o jeziku

komponente opisa oblikovnog obrasca

- **funkcija** - opisati rješenje oblikobnog problema
- čIne ga: kratki naziv, problem, rješenje, rezultat

podjela OO

- ponašajni obrasci
- strukturni obrasci
- obrasci stvaranja

STRATEGIJA

najčešće korišten obrasac
pripada obitelji ponašajnih obrazaca

problem

- odvojiti klijenta od izvedbi algoritma
- koristimo kad treba dinamički mijenjati ponašanje neke komponente
    - ponašanje zadajemo odabirom postupka -> nema potrebe za mijenjanjem izvornog koda komponente!
- korisno kad imamo više nezavisnih obitelji postupaka

rješenje

- ukratko: dio posla povjerimo vanjskom pružatelju preko pokazivača na njegov osnovni razred (npr. sučelje, apstraktni razred)
    - dakle, glavna metoda ima referencu na abstraktni razred/sučelje, a konkretnu implementaciju onda možemo lagano mijenjati (odnosno dinamički, konkretna implementacija se određuje prilikom izvođenja programa)
- detaljnije

    - definiramo hijerarhiju postupka
    - enkapsuliramo konkretne postupke
    - omogućujemo biranje ponašanja zadavanjem željenog postupka (npr. preko konstruktora komponente, preko settera itd.)
    - kontekst povjerava zadatke konkretnim postupcima preko roditeljskog sučelja

rezultat

- pospješuje se nadogradivost bez promjene
    - lako dodati nove postupke
- inverzija ovisnosti
    - kontekst ovisi o apstraktnom sučelju, ne konkretnoj implementaciji
- ortogonalnost - razdvajanje postupka i konteksta

----malo detaljniji opis----

namjera

- definirati obitelj međusobno izmjenjivih algoritama
- omogućiti izmjenu algoritama neovisno o klijentu koji poziva algoritme

primjenjivost

- kada kontekst treba dinamički konfigurirati sa željenim postupkom
- kada su potrebne različite varijante (algoritme) istog postupka
    - npr. imamo izvedbu A (troši više memorije, ali je brža) i izvedbu B (sporija je, ali troši manje memorije)
- kada želimo odvojiti kontekst od konkretnih implementacija
- kada želimo različito ponašanje odabrati virtualnim pozivom a ne if-om

sudionici

- **kontekst**
    - dio programa koji ima pokazivače na strategije (koji zove metode iz strategije)
- **(apstraktna) strategija**
    - najčešće sučelje - npr. IApstraktnaStrategija
    - ono što klijent poziva kad treba neku od metoda strategija
- **(konkretna) strategija**
    - konkretna implementacija
    - implementira sučelje IApstraktnaStrategija, preko kojeg onda klijent zna komunicirati s njom

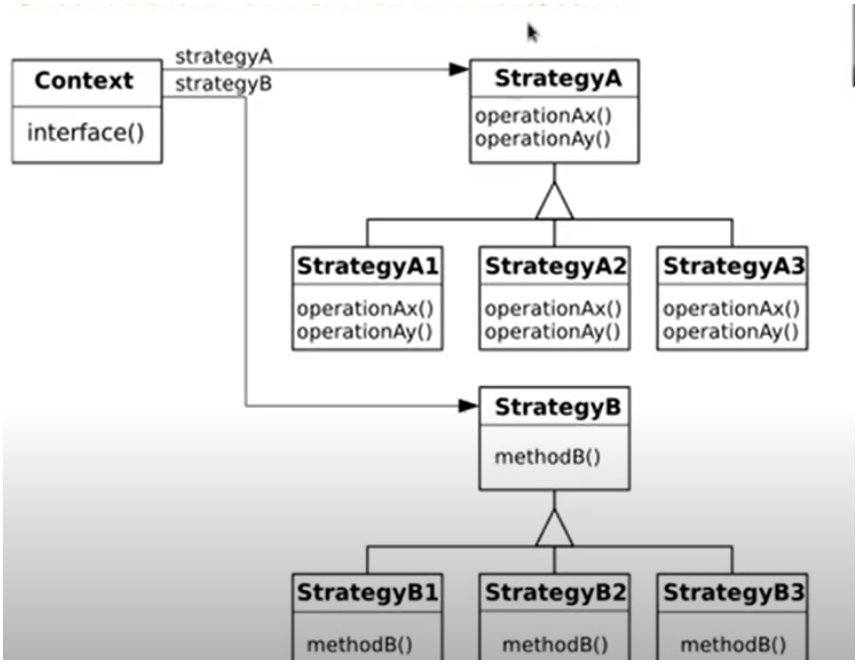
suradnja

- **prijenos parametara**
    - kako kontekst strategiji prenosi parametre?
        - 1) izravno - kao argumente poziva metode strategije
            - puno bolji način od 2)
        - 2) neizravno slanje - kontekst šalje referencu na sebe, a strategija uzima što joj treba
            - korisno većinom samo u razvoju, za lakše testiranje, ne u konačnom programu
    - **konfiguracija**
        - tj. kako kontekst dobiva konkretnu strategiju

- za to su odgovorni klijenti konteksta
  - onaj koji je instancirao kontekst, postavlja strategiju konteksta
- jednom kad konfiguriraju kontekst, klijenti komuniciraju samo sa kontekstom

posljedice

- ako postoji zajednička funkcionalnost, ona se može izdvojiti u apstraktnu strategiju
- ponekad bolja alternativa nasljeđivanju
  - jer alternativa bi bila da umjesto jednog konteksta koji može pozvati N implementacija strategije, imamo N podrazeda konteksta koji podržavaju sva ta različita ponašanja
- mičemo if-ove



- dijagram strategije
- npr. StrategyA i StrategyB su sučelja

često kad vidiš switch izraze u kodu, možeš ih zamijeniti sa OO Strategija

- probaj to napraviti u onoj JNotepad++ dz iz Jave, na onom jednom mjestu gdje si koristio onaj veliki switch

OKVIRNA METODA (eng. TEMPLATE METHOD)

tldr definirati okvirni postupak koji neke korake prepušta izvedenim razredima
slična strategiji

dakle kod okvirne metode imamo npr. apstraktni razred koji propisuje neki **okvir ponašanja**

- a izvedeni razredi onda to detaljnije određuju
- recimo imamo razred IgrajIgru, koji ima recimo neki init i for petlju u kojoj je makeMove naredba
- a konkretne implementacije onda mogu biti IgrajMinesweeper, IgrajŠah itd.
- metode apstraktnog razreda mogu biti

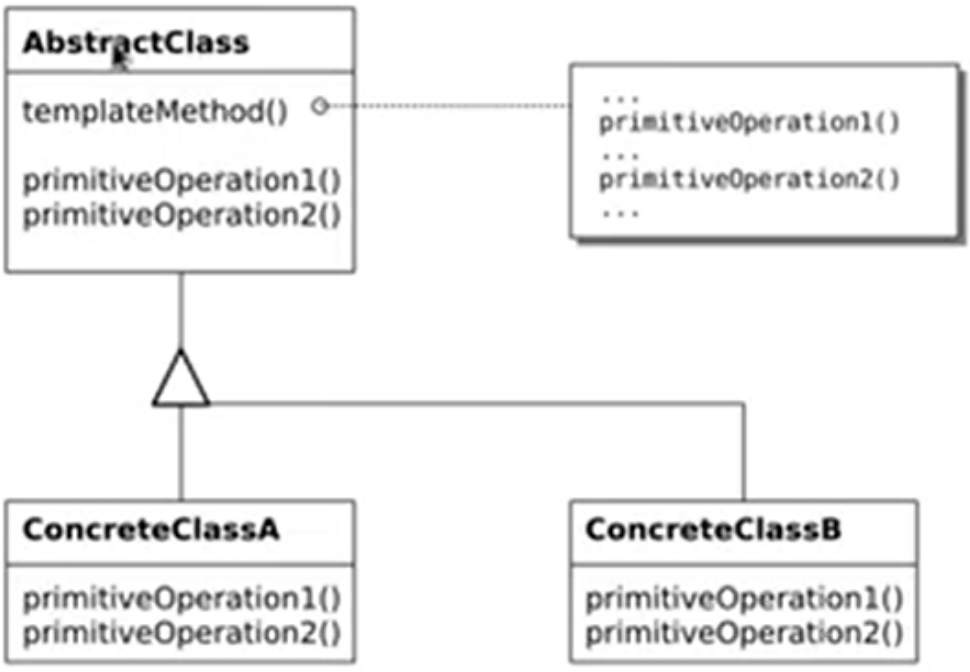
    - **apstraktni primitivi**
        - čiste virtualne metode, izvedeni razredi moraju ponuditi svoju implementaciju
    - **nadomjestive metode**
        - metode za koje apstraktni razred nudi neku implementaciju, koju izvedeni razredi mogu ali ne moraju nadjačati

dok kod strategije, imamo sučelje i kontekst onda poziva metode tog sučelja, a konkretna implementacija ovisi koja se implementacija strategije poslala kao argument kontekstu

- dakle kontekst poziva metode objekta, koji predstavlja strategiju

primjenivost

- kad istu strukturu postupka dijeli više konkretnih razreda
- ili kad stalne dijelove postupka želimo skupiti na jednom mjestu
    - zajedničko ponašanje izdvojimo da izbjegnemo dupliciranje koda
- kad izvedeni razredi trebaju mijenjati osnovnu funkcionalnost



- primitiveOperation1 i 2 su čiste virtualne metode

sudionici

- **apstraktni razred**
    - modelira npr. zajednički dio svih igara
    - deklarira apstraktne primitive koje će izvedeni razredi kasnije nadjačati
- **konkretni razredi**
    - implementiraju primitive, izvode konkrente korake postupka

suradnja

- konkretni razredi se oslanjaju na implementaciju zajedničkih dijelova postupka u apstraktnom razredu

prednosti i mane u odnosu na strategiju

- dopiši

PROMATRAČ

ponašajni obrazac u domeni objekata

tldr: namjera je ostvariti ovisnost 1:n

- glavni objekt i ovisni objekt
- ovisni objekti treba nešto raditi kad glavni objekt promijeni stanje
- listeneri u Java swingu i ono što smo radili u JNotepad++ zadaći su basically to

glavni objekt (**subjekt**) - izvor informacija
ovisni objekti (**promatrači**) - obrađuju informacije

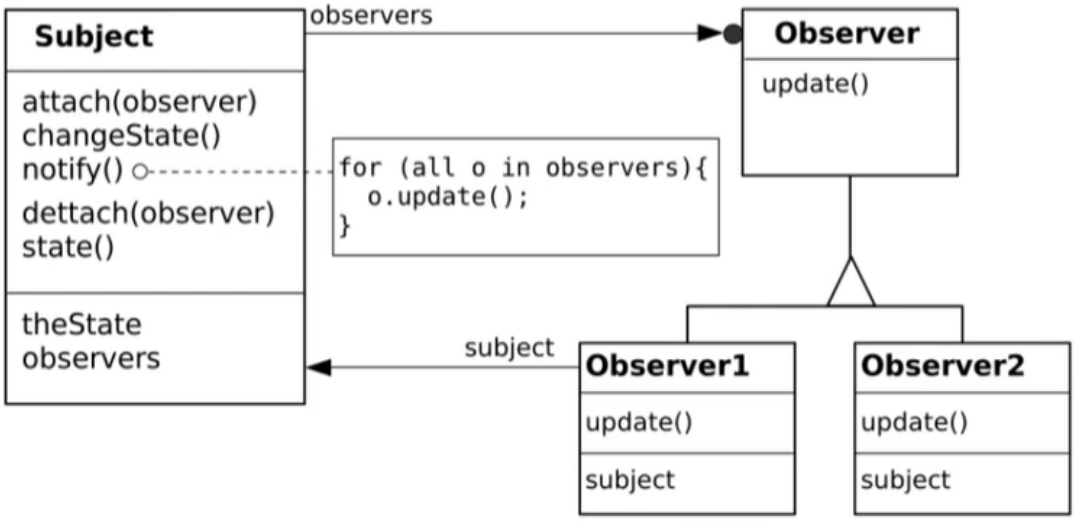
promatrači se pretplaćuju kod subjekta
subjekt ne zna za konkretni tip promatrača

- zna samo da npr. implementiraju neko sučelje koje ima metodu JaviPromatraču, i onda kad dođe do neke promjene on nad svakim od tih objekata pozove JaviPromatraču

primjenjivost

- kad promjena jednog objekta zahtjeva promjene u drugim objektima
  - a ti objekti nisu unaprijed poznati
- kad objekt treba obvaviještavati druge objekte
  - a želimo da bude što neovisniji o njima

strukturni dijagram



- vidiš da promatrači implementiraju neko sučelje, i preko tog sučelja subjekt komunicira s promatračima
  - gleda ih kroz naočale tog sučelja
- promatrač poziva metodu attach nad subjektom
- kad dođe do promjene, subjekt zove notify(), čime obaviještava sve promatrače
- vidimo da subjekt ovisi o sučelju -> inverzija ovisnosti

sudionici

- **subjekt**
  - poznaje promatrače preko apstraktnog sučelja
  - nudi metodu za dodavanje promatrača
  - obaviještava promatrače o promjenama stanja
- **promatrač - sučelje**
  - sučelje za obavještavanje o promjenama u subjektu
- **konkretan promatrač**
  - sadžre referencu na subjekt

suradnja

- promatrači se dinamički prijavljuju i odjavljuju kod subjekta
- subjekt obaviještava promatrače kad se dogodi promjena
- promjene subjekta mogu biti inicirane i od strane promatrača
  - npr. GUI aplikacija (prikaz aplikacije = promatrač, ali i komponenta koja mijenja stanja)
- obaviještavanje provodi subjekt, ali mogu i njegovi klijenti

posljedice

- ukida ovisnosti između subjekta i promatrača
  - subjekt ne ovisi o implementaciji promatrača
  - promjene koda u promatračima ne utječu na subjekt
- dinamička prijava i odjava promatrača

implementacijska pitanja

- odlučiti tko je odgovoran za obavještavanje
  - najčešće subjekt nakon svake promjene stanja
    - može dovesti do redundantnih obavijesit
  - ponekad mogu i klijenti subjekta biti odgovorni za obavještavanje, nakon niza promjena
    - ovo se ne koristi prečesto
- protokol obaviještavanja može biti detaljan ili siromašan
  - možemo npr. uz obvavijest o promjeni stanja slati i nekakve dodatne informacije
- prijenos parametara
  - neizravan (pull) - promatrači imaju referencu na subjekt
  - izravan (push) - stanje se prenosi preko argumenata metode update

DEKORATOR (eng. decorator, wrapper)

tldr dinamičko dodavanje i povlačenje odgovornosti **objektima** (ne razredima!)

- različite dodane odgovornosti po volji kombinirati s različitim oblicima osnovne funkcionalnosti
- po ovome se razlikuje od strategije - kod strategije imamo jedno sučelje i različite implementacije tog sučelja
- a ovdje razredni izvedeni iz, ajd, sučelja, implementiraju metode tog sučelja, ali da im se mogu (dinamički) dodati dodatne funkcionalnosti

strukturni obrazac u domeni objekata

tražimo da rješenje ima svojstva

- dodatne odgovornosti definirane u zasebnim komponentama
  - NJO, NBP
- dodatne odgovornosti primjenjive na sve temeljne funkcionalnosti
  - da svaku osnovnu odgovornost možemo kombinirati sa svakom dodatnom odgovornošću
  - NJO, NBP
- mogućnost dinamičke konfiguracije dodatnih odgovornosti
- klijenti ne moraju znati ni za temeljne funkcionalnosti ni za dodatne odgovornosti
  - NIO

rješenje

- razredi s dodatnim odgovornostima **umataju** objekt na koji djeluju
- bez znanja o njegovom konkretnom tipu
- ovaj primjer iz prezentacije

```
public class Gunzip {
  public static void main ( String [] args ) {
    FileInputStream fis = new FileInputStream(args[0]);
    BufferedInputStream bfis = new BufferedInputStream(fis);
    GZIPInputStream gbfis = new GZIPInputStream(bfis);
    // clients transparently read from gbfis...

    FileOutputStream fos = new FileOutputStream(args[1]);
    BufferedOutputStream bfos = new BufferedOutputStream(fos);
    // clients transparently write to bfos...

    copy ( gbfis , bfos );
  }
```

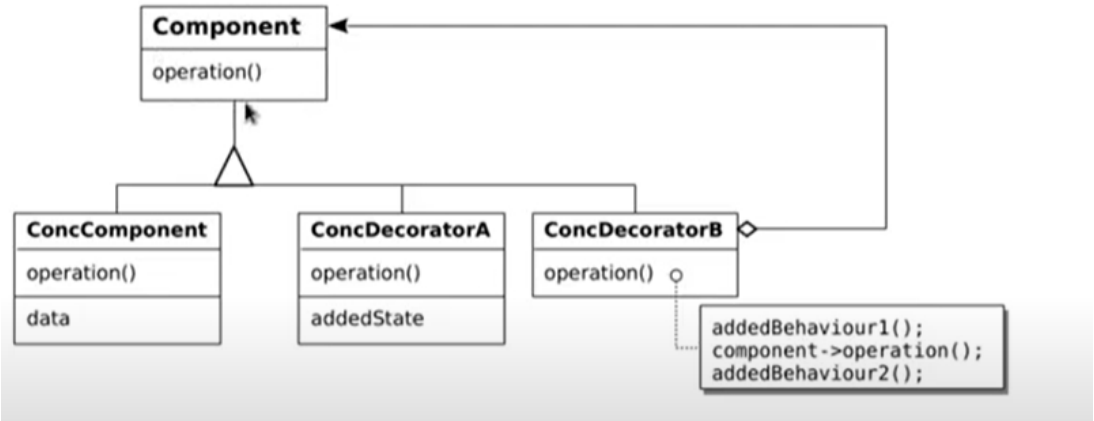
  - BufferedInputStream omata FileInputStream - BufferedInputStream jedino zna da fis implementira InputStream sučelje
    - bufferedinputstream dodaje funkcionalnost bufferiranja
  - GZIPInputStream omata buffered stream (GZIPInputStream je dekorator)
    - dodaje funkcionalnost unzipanja
  - svi oni nasljeđuju InputStream, pa klijenti mogu čitati i Buffered i GZIP InputStream jer znaju da implementiraju InputStream sučelje
    - klijenta i ne zna koja je točno
- rezultat
  - dodatne odgovornosti u odvojenim komponentama
    - npr. BufferedInputStream je u svojoj komponenti
  - slobodno kombiniranje s osnovnim odgovornostima
    - npr. BufferedInputStream možemo koristiti s ostalim InputStreamovima
  - dinamičko konfiguriranje dodatnih odogovrnosti
  - klijent može znati samo za osnovno sučelje

namjena

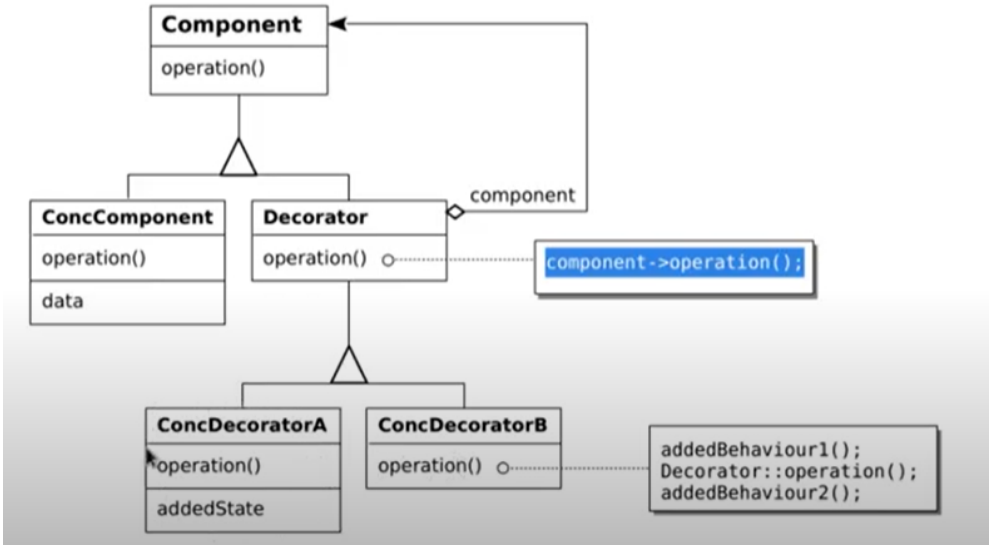
- dodijela nezavisnih funkcionalnosti objektima (ne cijelim razredima)
- temeljna ideja - rekurzivna kompozicija
  - ako A omata B
  - i A i B imaju isti osnovni razred
    - A = matični objekt
    - B = podatkovni član
  - u gornjem primjeru - BufferedInputStream omata FileInputStream
  - ali oboje imaju isti osnovni razrd - InputStream
- dekorator A implementria dodatnu funkcionalnost

    - ostale odgovornosti proslijeđuje omotanoj komponenti B
- mogućnost rekurzivnog umetanja

dijagram



zajedničke operacije svih dekoratora možemo izdvojiti u zajednički osnovni razred



primjena

- kada **objektima** trebamo dinamički dodavati odgovornosti
    - odgovornosti potrebno moći povući
- zašto ne nasljeđivanje osnovne komponente?
    - želimo dinamičku konfiguraciju
    - ne želimo višestruko nasljeđivanje
- zašto ne unutar osnovnog razreda?
    - kršenje načela jedinstvene odgovornosti
    - ponekad neki dekoratori nisu primjenjivi na sve objekte

sudionici

- **(abstraktna) komponenta**
    - definira sučelje za objekte kojima želimo dinamički dodavati funkcionalnosti
    - u gornjem primjeru - InputStream
- **konkretna komponenta**
    - dijelovi kojima možemo dinamički dodati funkcionalnosti
    - u gornjem primjeru - FileInputStream, BufferedInputStream itd.
- **(abstraktni) dekorator**
    - nasljeđuje sučelje (razred) komponente

    - ima referencu na umetnutu komponentu
    - može delegirati posao unutarnoj komponenti
- **konkretni dekorator**
    - izvodi dodatne odgovornosti komponente
    - u gornjem primjeru npr. BufferedInputStream

suradnja

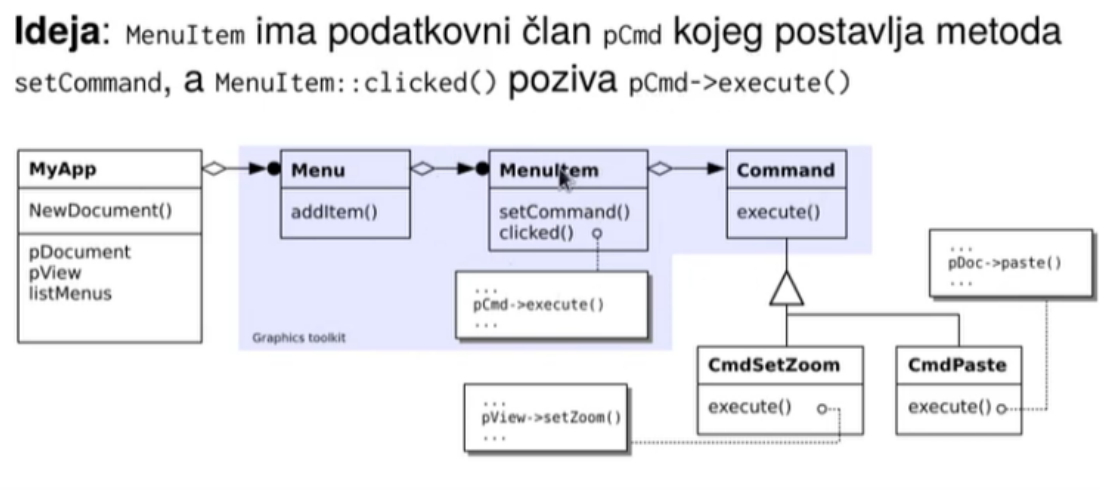
- dekoratori - prosljeđuju zahtjeve primljene preko sučelja komponente sadržanim konkretnim komponentama
- prije i poslije mogu obaviti tu dodatnu funkcionalnost
- klijenti mogu konfigurirati dodatne funkcionalnosti
- ako ne trebaju dodatnu funkcionalnost, mogu koristiti osnovnu funkcionalnost preko apstraktnog sučelja

na primjeru na str 59 primijeti inverziju ovisnostj - copy ovisi o apstraktnim komponentama InputStream i OutputStream, ne zna za konkretne implementacije

NAREDBA

tldr omogućiti unificirano baratanje raznim zahtjevima

ponašajni obrazac u domeni objekata

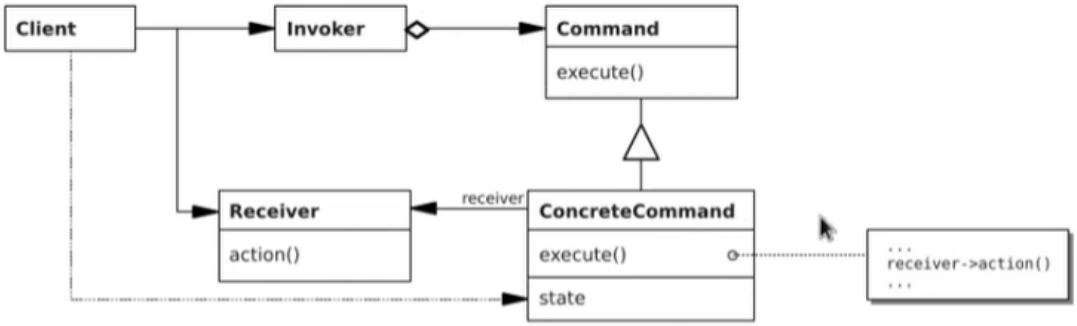


- menu item konkretnu naredbu zna samo kroz sučelje Command
- primijeti da sad upravljač menijima (Menu, MenuItem i Command) su NBP, pa ih možemo zasebno izvodjiti

naredba razdvaja zaprimanje, pozivanje i obradu zahtjeva

- zaprimanje - setCommand()
- pozivanje - pCmd->execute()
- obrada - konkretna implementacija metode execute()

postižemo rukovanje poslovima bez ovisnosti o konkretnim operacijama



- primijeti da invoker poziva i naredbu i (indirektno) recievera (kojeg poziva konkretna naredba) bez da zna za njihove konkretne implementacije
- client - glavni program, on daje konkretnu naredbu invokeru

hint: za undo funkcionalnost (tražiti će se u nekom od labosa)

- https://youtu.be/a1SuSDeWzOA?list=PLkOLgurQ4FfMI7lhtSTImxdbe0o5RRtpo&t=692

fleksibilnost ostvarena kombinacijom povjeravanja i nasljeđivanja

primjena

- kad je neku komponentu potrebno parametrizirati s nepoznatom akcijom, koju komponenta treba pozvati
- komponente želimo proizvoljno kombinirati s akcijama

sudionici

- **naredba**
  - sučelje za izvođenje operacija
  - npr. metoda executa
- **konkretna naredba**
  - definira vezu između pozivatelja i primatelja
  - implementira sučelje naredbe, poziva primatelja
  - pozivatelj zna samo za sučelje, ne za konkretnu naredbu
  - npr. CmdPaste
- **pozivatelj**
  - u primjeru MenuItem
  - inicira zahtjev preko sučelja naredbe
- **primatelj**
  - objekt koji zna izvršiti zahtjev
- **klijent**
  - prima konkretnu naredbu i predaje je pozivatleju

suradnja

- klijent - kreira konkretnu naredbu, navodi njenog primatelja, registrira naredbu kod pozivatelja
- pozivatelj - poziva konkretnu naredbu preko osnovnog sučelja
- konkretne naredbe - pozivaju metode primatelja da zadovolje zahtjev
- pozivatelj i primatelj se ne poznaju

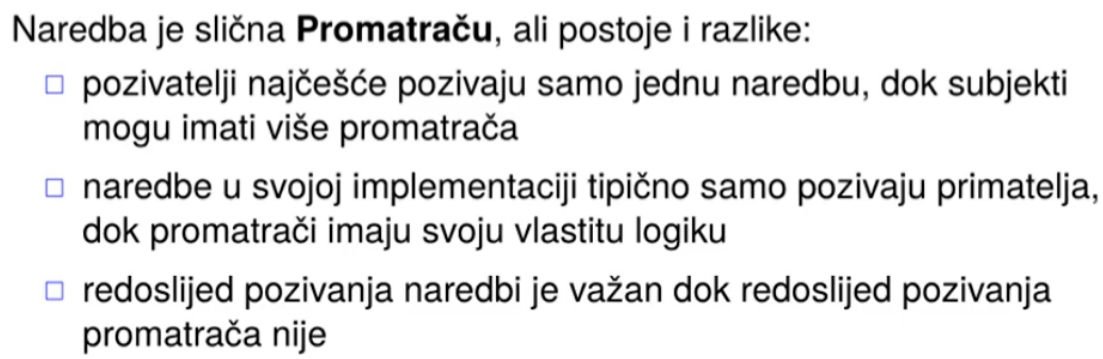
posljedice

- odvaja pozivatelja od objekta koji zna izvršiti operaciju (primatelj)
- dodavanje novih naredbi je lako - samo implementiraju sučelje naredba

recimo primjena ovog obrasca su oni AbstractActioni iz JNotepad++ dz iz Jave

- i recimo stvaranje bezimenog razreda (Action myAction = new AbtsractAction(){...}) je jedna stvar koja olakšava pisanje ovog obrasca

OO naredba možemo promatrati kao specijalni slučaj strategije gdje sučelje naredba ima samo jednu metodu - execute



TVORNICA

kreacijski obrazac

namjena - na odgovarajući način riješiti problematiku stvaranja primjeraka razreda
iako većina koda može ovisiti o apstraktnim sučeljima, netko negdje mora stvoriti konkretne objekte
želimo imati mogućnost stvaranja konkretnih objekata bez da ovisimo o konkretnom tipu koji želimo stvoriti

osnovna ideja

- stvaranje objekata izdvojimo u zasebnu metodu
- toj metodi predajemo neke argumente
- ona na osnovu tih argumenata određuje koji će točno objekt stvoriti
- ta funkcija - **parametrizirana tvornica**
  - osnovni kreacijski obrazac
  - namjera: lokalizirati odgovornosti za stvaranje objekata
- objekt koji želi dobiti primjerke tih razreda zove metodu parametrizirane tvornice

ovu osnovnu ideju kasnije razrađuju konkretni kreacijski obrasci

- **metoda tvornica**
  - prepušta implementaciju kreacijske metode klijentima
- **apstraktna tvornica**
  - kada je poterebno riješiti stvaranje većeg broja povezanih razreda
- **jedinstveni objekt**
  - onemogućuje stvaranje više od jednog primjerka nekog razreda
- **prototip**
  - stvara objekte iz unaprijed pripremljenih prototipova
  - pozivatelj onda ne ovisi o konkretnom razredu objekta koji želi stvoriti

ovi obrasci smanjuju ovisnost o konkretnim razredima, ali ipak ju potpuno ne uklanjanju

- jer opet tvornice ovise o tim konkretnim razredima

možemo li ukloniti ovisnost tvornice o konkretnim razredima?

- to su **generičke tvornice**

generička tvornica

- tvornica koja je pozna proizvode
- prilikom prevođenja **ne zna** kojeg su razreda objekte koje će stvarati
- kako to izvesti?
    - 1) učitati izvedbe metoda iz zadane dinamičke biblioteke (npr. .dll)
        - dinamičke biblioteke: kod koji nema main prevedemo i pohranimo u izvršnom formatu
            - to su dinamičke biblioteke
            - njih onda pokreće tj. uključuje u sebe neki kod koji ima main
        - iz te dinamičke biblioteke onda možemo dobiti adresu tj, pointer na funkciju, koju onda pozovemo
        - i ta funkcija onda stvara objekt
    - 2) koristimo introspekciju preko simboličkih imena razreda i paketa
        - npr. u Javi: Class.forName(strClassName)
        - Python: imp.load_module(strModuleName), getattr(module, strName)
        - 3) registriramo konstrukcijsku funkciju iz statičkog inicijalizatora (C++)
            - ideja: statički inicijalizator poziva generičku tvornicu
            - argumenti: konstrukcijska funkcija, simboličko ime (string)

SINGLETON (JEDINSTVENI OBJEKT)

osigurati da razred ima samo jednu instancu
omogućiti toj instanci globalni pristup
dakle, to je globalna varijabla sa zabranom daljnjeg instanciranja i odgođenim instanciranjem

dakle, ovo želimo spriječiti (postojanje više instanci):
Singleton s1 = new Singleton();
Singleton s2 = new Singleton();

želimo da se ovakav kod uopće ne želi ni prevesti!

kako to možemo riješiti?

- uporabom privatnih konstruktora - tako korisnik iz vana ne može pozvati konstruktore
    - onda samo programski kod iz tog razreda može pozvati taj konstruktor
- zatim definiramo statičku referencu na Singleton i instanciramo ju tim konstruktorom;

```
public class Singleton {

    private final static Singleton instance = new Singleton();

    private Singleton() {

    }
```

    - dakle, varijabla se incijalizira kada se inicijaliziraju statičke varijable razreda Singleton
        - kada točno?
            - ovisi o jeziku

        - C++ odmah nakon početka izvođenja maina inicijalizira statičke varijable
        - Java statičke inicijalizatore razreda radi onog trenutka kada se razred doista počne koristiti u programu
- sada moramo korisniku omogućiti da pristupi toj varijabli
- dodajemo javnu **statičku** metodu getInstance()

```
private final static Singleton instance = new Singleton();

private Singleton() {

}

public static Singleton getInstance() {
    return instance;
}
```

    - i preko te metode korisnik pristupa jednoj jedinoj instanci razreda Singleton!
    - mora biti statička jer kako bi došli do inicijalnog objekta da nije statička

ukoliko baš želimo da se inicijalizacija Singletona događa tek prilikom prvog korištenja tog razreda (npr. jer je konstruktor jako skupa operacija), to možemo ovako

```
private static Singleton2 instance = null;

private Singleton2() {
    System.out.println("Pozvan je konstruktor i stvoren je objekt!");
}

public static Singleton2 getInstance() {
    if(instance == null) {
        instance = new Singleton2();
    }
    return instance;
}

public void demoMetoda() {
    System.out.println("Ja sam pozvana!");
}
```

- to se naziva **Singleton s odgođenim stvaranjem**
- u Javi je pristup 1 i 2 jednak, objašnjeno gore zašto
    - jer se inicijalizacije statičkih članova događaju tek kad se objekt prvi put koristi

sudionici

- **jedinstveni objekt (singleton)**
    - definira javnu statičku metodu za pristup objektu instance()
    - definira privatni konstruktor
        - da ga korisnik ne može pozivati i stvarati nove primjerke tog razreda
    - pristupna metoda može implementirati odgođeno instanciranje
- **klijenti**
    - pristupaju Singletonu preko metode getInstance()

posljedice

- kontrolirani pristup jedinoj instanci razreda
- nema opterećenja namespacea
- konkretni tip Singletona može se odabrati tvornicom
- Singleton je u neku ruku globalna varijabla
    - povećava međuovisnost i smanjuje modularnost
    - treba ga koristiti s mjerom

varijanta - **objekt s dijeljenim stanjem**

- možemo imati više instanci, ali sve one dijele isto stanje
- stanje izvedeno kao podatkovni član razreda
- npr. svi objekti kad se nad njima pozove neka metoda vraćaju isti rezultat
- popularno u Pythonu

ITERATORI

ponašajni obrazac

ideja

- neovisno o implementaciji skupa objekata, omogući prolaz po svim njegovim objektima

U Javi - objekt tipa Iterable je objekt nad kojim možemo pozvati metodu tvornicu
iterable.iterator() koja nam vraća iterator nad kolekcijom
sve kolekcije u Javi implementiraju sučelje Iterable

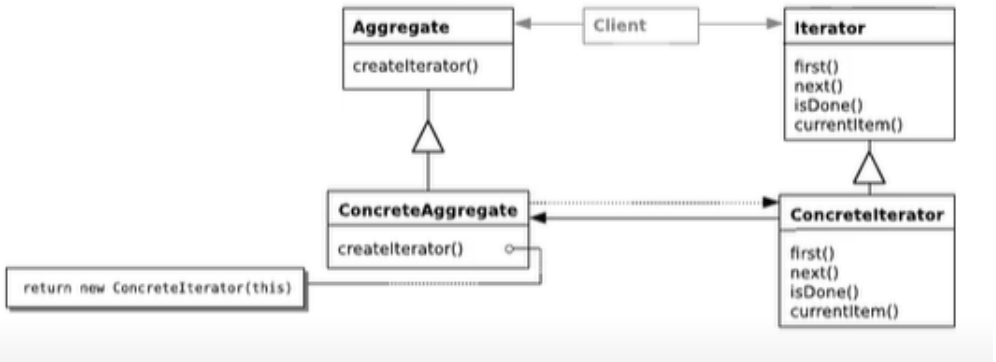
namjera

- omoguíti uniformni slijedni obilayak kolekcije bez otkrivanja njene interne strukture
- sličan obilasku pokazivačem
    - prednosti
        - uniforman obilazak različitih kolekcija
        - enkapsulran pristup element
        - modularna organizacija
        - ...

primjena

- namjena
    - enkapsuliran i uniforman pristup elementima kolekcije
    - polimofrni obilazak kolekcije
        - klijent aptrahira vrstu kolekcije i način oblilaska
    - usporedni obilasci kolekcije

strukturni dijagram



-
    - **(asptraktni) iterator**
        - model (sučelje) koji definira kako klijent obavlja interakciju s iteratorom
        - dakle sučelje koje definira metode iteratora
    - **konkretni iterator**

        - konkretna implementacija sučelja iterator
        - najčešće ga stvara konkretni agregat preko metode tvornice .iterator()
        - evidentiraju tekući položaj u kolekciji koju iteriraju
    - **(apstraktni) agregat (skupni objekt na hrv. - dakle neka kolekcija)**
        - sučelje koje specificira da klase koje ga implementiraju moraju nuditi metodu za stvaranje iteratora
        - to je ono da sve kolekcije implementiraju **sučelje Iterable**
            - sučelje Iterable == apstraktni agregat
    - **konkretni agregat (skupni objekt)**
        - agregat - npr. neka kolekcija, lista, set itd.
        - konkretne implementacije sučelja agregat
        - npr. konkretne implementacije kolekcija u Javi koje specificiraju svoju metodu .iterator()
        - stvara konkretni iterator
        - konkretni iterator mora imati referencu na konkretni agregat (tj. kolekciju koja sadrži elemente po kojima iterira)
    - **klijent**
        - onaj koji poziva iterator nad kolekcijom

posljedice

- pospješuje se jedinstvena odgovornost
    - jer agregat ne definira sučelje za pristup svojim objektima
    - već to prepušta iteratoru
    - a on ga samo stvara
    - i onda pristup elementima agregata ovisi samo o jednom objektu - iteratoru
- omogućen usporedni prolaz kroz agregat
- uska povezanost iteratora i agregata

imati na umu da operacije nad spremnikom (agregatom) mogu poništiti valjanost iteratora

- recimo iterator se nalazi na i-toj poziciji
- korisnik odluči izbrisati element na pozicij i+1
- što napraviti kad se nad iteratorom pozove next, hasNext...?
    - jedna opcija da iterator zapamti početno stanje
    - i uspoređuje ga sa trenutnim stanjem
    - ako shvati da je došlo do promejen, baci iznimku (ConcurrentModificationException)

kraj obilaska

- što se događa ako iteriramo nakon kraja kolekcije (pozovemo next kad više nema ničega)
- C++ - nedefinirano ponašanje
- Java i Python bacaju iznimke
    - Java - iznimka se baca samo iznimno - preporuča se provjera sa hasNext
    - Python - kraj kolekcije se uvijek signalizira iznimkom StopIteration
        - njegovi iteratori nemaju metodu hasNext
        - ovakvi iteratori (samo u Pythonu!) mogu modelirati procese koji generiraju podatke
            - npr. čitaju ih iz socketa
            - nazivamo ih **generatorima**
            - dopiši malo o njima

srodni obrasci

- polimorfne iteratore ponekad kreiramo u metodi tvornici konkretnog skupnog objekta
- kompozit može omogućiti obilazak elemenata prikladnim iteratorom

PRILAGODNIK (ADAPTER)

strukturni obrazac (omata vanjsku komponentu)

namjera

- prilagoditi **postojeći** razred **sučelju kojeg očekuju klijenti**
- najčešće korišten obrazac
- češće se isplati novi prilagodni kod nego modificirati postojeći
- recimo da imamo više biblioteka koje nude neke funkcionalnosti
    - razvijemo neko sučelje koje odražava potrebe naše aplikacije
    - za svaku biblioteku napišemo prilagodnik
    - konkretnim implementacijama (bibliotekama) sad možemo pristupiti kroz njihove prilagodnike
    - a kako svi prilagodnici implementiraju isto sučelje, svima im možemo pristupati preko tog sučelja (OO Strategija)

primjer - imamo sučelje Stog koje želimo implementirati, te imamo razred Vektor koje već nudi praktički svu tu funkcionalnost, ali ne kroz sučelje Stog

objektni prilagodnik

- primjerak razreda koji implementira sučelje kojem se želimo prilagoditi
- a interno koristi primjerak nekog drugog razreda
- i onda u svim metodama interno delegira pozive metoda primjerku drugog razreda

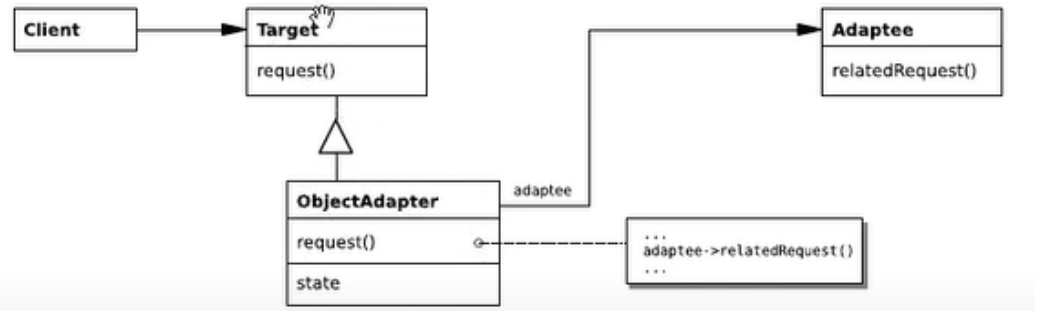
```
public static class StogObjectAdapter<T> implements Stog<T> {
    private Vektor<T> vektor = new Vektor<>();

    @Override
    public boolean isEmpty() {
        return vektor.size() == 0;
    }

    @Override
    public void push(T elem) {
        vektor.add(elem);
    }

    @Override
    public T pop() {
        if(isEmpty()) throw new EmptyStackException();
        return vektor.removeLast();
    }
```

- }



    - Target - ciljno sučelje (Stog iz primjera)
    - Adaptee je npr. razred Vector iz primjera - razred koji želimo prilagoditi sučelju
    - primijeti da Adapter ima referencu na razred koji se adaptira

- Stog je sučelje kojem se želimo prilagoditi, a vektor primjerak nekog drugog razreda koji nudi tu funkcionalnost ali ne implementira sučelje Stog
- prednosti
    - može se primijeniti i na objekte razreda izvedenih iz vanjskog razreda

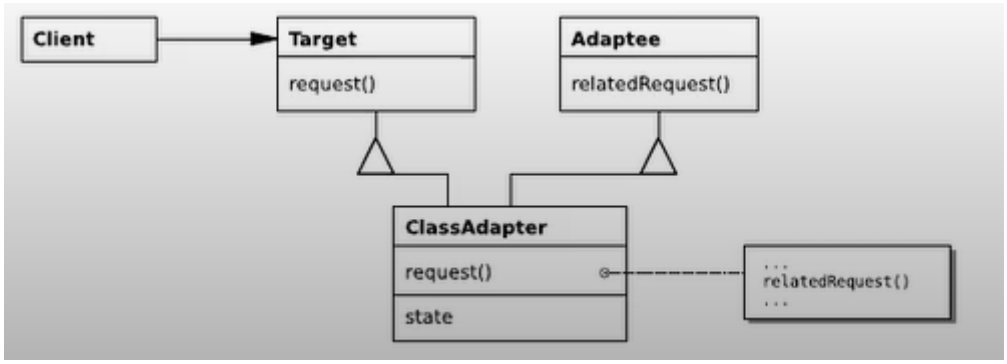
razredni prilagodnik

- razred koji nasljeđuje razred "pogrešnog tipa", a implementira sučelje kojem se želimo prilagoditi
    - odnosno razred koji nam nudi funkcionalnost koja nam treba, ali ne implementira odgovarajuće sučelje
    - to je npr. razred Vector u onom primjeru - imao je svu funkcionalnost koja nam je trebala, ali nije implementirao sučelje Stag
- sad nema potrebe stvarati novi primjerak razreda
    - jer je primjerak tog objekta upravo primjerak našeg adaptera

```
public static class StogClassAdapter<T> extends Vektor<T> implements Stog<T> {
    @Override
    public boolean isEmpty() {
        return this.size() == 0;
    }

    @Override
    public void push(T elem) {
        this.add(elem);
    }

    @Override
    public T pop() {
        if(isEmpty()) throw new EmptyStackException();
        return this.removeLast();
    }
}
```

- 

    - Adapter sad radi višestruko nasljeđivanje
    - ostalo je slično
- prednost nad objektnim prilagodnik - ako razred Vektor ima neke korisne zašitćene metode, možemo ih koristiti
    - ili ih čak možemo i nadjačati
    - također, rukujemo samo jednim objektom
- no ipak postoji cijena
    - (bar u Javi, C++ je malo bolji - privatna i javna nasljeđivanja)
    - nad razrednim prilagodnik možemo pozivati i metode razreda Vektor, ne samo Stog!
    - a to ne najčešće želimo

primjenjivost

- kad nas nekompatibilno sučelje sprječava u korištenju postojećeg koda
- kad je potrebno uniformno pristupati raznim resursima (bez da mijenjamo kod tih resursa)
    - primjer s kamerom

- kad je potrebno istovremeno prilagoditi više izvedenih razreda, a sučelje kojem se treba prilagoditi nalazi se u osnovnom razredu

sudionici

- **ciljno sučelje (Target)**
  - definira sučelje koje želimo implementirati
  - sučelje Stog iz primjera
- **Klijent**
  - npr. main program
  - surađuje s objektima koji implementiraju ciljno sučelje
- **Vanjska komponenta (Adaptee)**
  - implementira korisnu funkcionalnost kroz nekompatibilno sučelje
  - razred Vector iz primjera
    - nudio je mogučost dodavanja i uklananja elemenata, ali ne kroz sučelje Stog
- **Prilagodnik (Adapter)**
  - prilagođava sučelje Vanjske komponente Ciljnom sučelju

suradnja

- klijenti pozivaju metode ciljnog sučelja
- prilagodnik pozive implementira preko poziva sučelja vanjske komponente

posljedice

- potpora korištenju apstraktnih sučelja (dakle inverzija ovisnosti)
- bolja razdioba odgovornosti

implementacija

- piše i gore ali ovdje je tldr
- **razredni prilagodnik**
  - javno nasljeđivanje ciljnog sučelja
  - privatno nasljeđivanje vanjskog razreda (ili samo extend ako je Java)
- **objektni prilagodnik**
  - javno nasljeđivanje ciljnog sučeja
  - povjeravanje objektu vanjskog razreda

KOMPOZIT

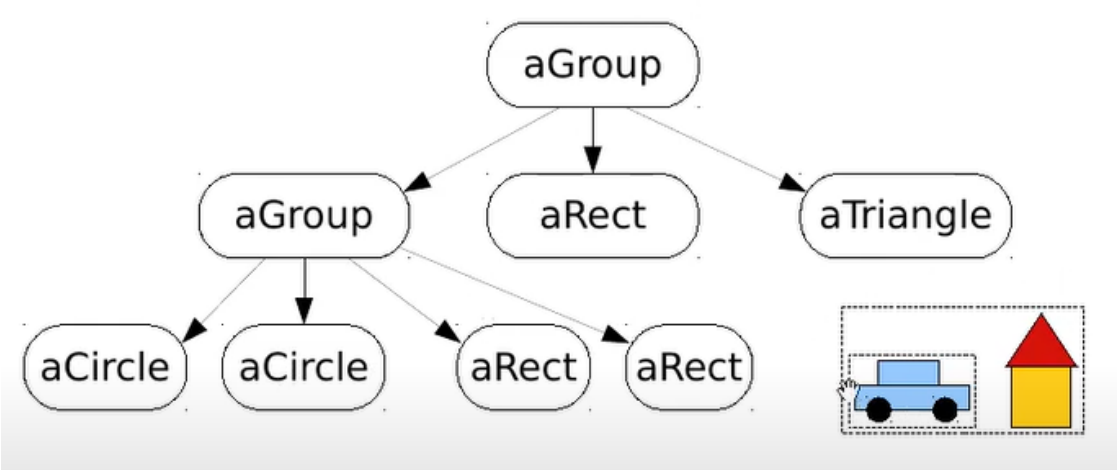
strukturni obrazac

- kompzit sadrži komponente
- temeljna ideja - rekurzivna kompozicija

namjera

- omogučiti da se grupom objekata može baratati na isti način kao i s instancama pojedinih objekata
- unutar jedne grupe može biti i još grupa
- primjer
  - grupiranje pojedinačnih elemenata vektorskog crteža
  - kompozitni objekt postaje punopravni element crteža

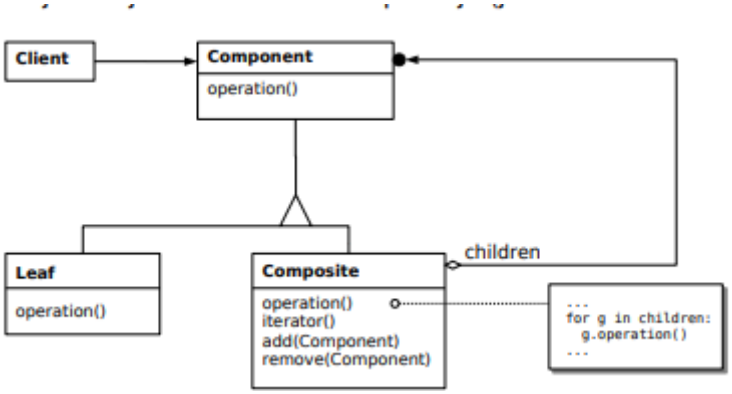
        - tretira se isto kao pojedinačni objekt
            - iscrtavanje, pomicanje...
    - operacije nad kompozitnim objektom delegiraju se sastavnim objektima



primjenjivost

- predstavljanje hijerarhije cjelina i pojedinačnih dijelova
- nudi transparentne operacije nad elementima takvih hijerarhija

struktura



- kompozit sadrži komponente Component
- ali je i sam primjerak razreda Component
- zato

sudionici

- **komponenta (component)**
    - deklarira zajedničko osnovno sučelje za primitive i kompozite
        - dakle deklarira koje sve metode ima itd.
    - u primjeru - osovni razred za element vektorskog crteža (GeometricShape)
- **primitiv (leaf)**
    - atomarna komponenta
    - implenitra sučelje komponente
    - npr. Circle, Square u primjeru
    - nemaju djecu!
- **kompozit (composite)**
    - definira ponašanje složenih komponenata - roditelja
    - odgovoran za pohranjivanje sastavnih komponenata - djece
    - delegira operacije sastavnim komponentama

    - u primjeru - razred Group
    -
- **klijent**
    - manipulira složenim objektima (i primitivima i kompozitima) preko sučelja komponente
    - ne zna i nije ga briga radi li se o primitivu ili kompozitu

suradnja

- klijenti koriste elemente hijerarhije preko sučelja Komponenta
- pozivi nad primitivima - izravna obrada
- pozivi nad komponentama - delegiranje poziva sastavnim komponentama
- kompoziti mogu obaviti dodatne operacije prije ili poslije delegiranja

posljedice

- definira se hijerarhija koja se sastoji od primitiva i kompozita
- klijent koji radi s primitivima može primiti i kompozite
- pojednostavljuje se odgovornost klijenta
    - i primitive i kompozite možemo tretirati na jednak način
- olakšano dodavanje novih komponenata
    - novi kompoziti i primitivi automatski se uklapaju u postojeći kod

implementacija

- jedna stvar koju možemo napraviti je da djeca imaju reference na roditelja
    - omogućava prolazak kroz hijerarhiju u oba smjera
    - mjesto za implementaciju - klasa Komponenta
    - recimo kažemo da svaka Komponenta može biti dio kompozita
        - i da ima mjesto za referencu na roditelja
- gdje deklarirati upravljanje sastavnim dijelovima komozita?
    - npr. poziv iteratora, dodavanje, uklananje itd.
    - mogućnosti
        - **u kompozitu**
            - rješenje u strukturnom dijagramu
            - problem - ako samo kompozit npr. ima metodu add
            - klijent mora biti svjestan razlike između nekompozita i kompozita
            - jedno rješenje - u razred Komponenta dodamo metodu getComposite
                - podrazumijevana implementacija vraća null - to vraćaju primitivi
                - kompozit vraća this
                - i samo kompozit (ne Komponenta!) onda ima metode add, remove itd.
        - **kroz razred Komponenta**
            - razrded Komponenta uključuje metode add, remove itd.
            - primitivi pozive tih metoda implementiraju tako da bacaju iznimku

jedna česta primjena - datotečni sustavi

srodni obrasci

- dekorator je strukturno sličan kompozitu
    - dekorator - reference 1:1, a kompozit 1:n
    - ali funkcionalno su jako različiti

STANJE

ponašajni obrazac - kontekst referencira apstraktno sučelje stanja

namjera

- omogučiti dinamičko mijenjanje ponašanja objekta
- objektno-orijentirana implementacija konačnog stroja
- efekt - kao da objekt dinamički mijenja klasu iz koje je instanciran

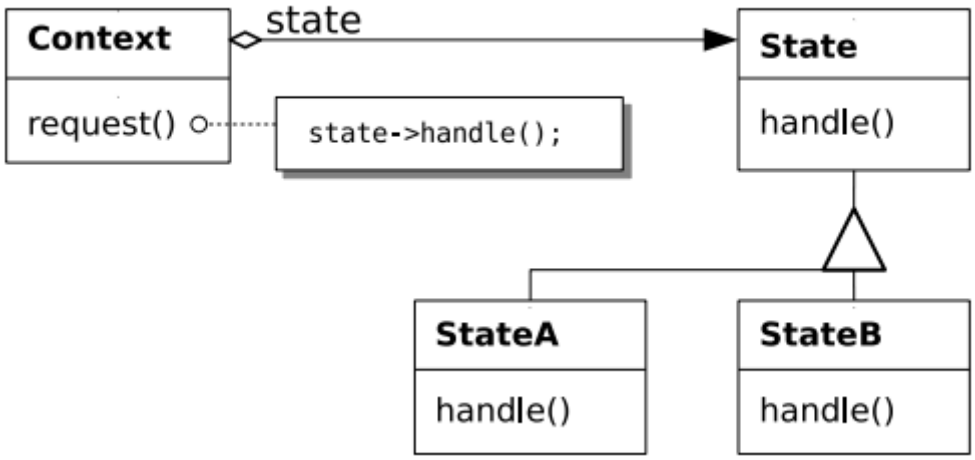
ideja - funkcije čije ponašanje ovisi o stanju izdvojiti u poseban apstraktni osnovni razred ponašanje u okviru svakog zasebnog stanja definirati odgovarajućim konkretnim razredom

primjenjivost

- kad se ponašanje objekta mora dinamički uskladiti s tekućim stanjem
    - jedna stvar koja ukazuje na potrebu ovog obrasca - imamo metode koje imaju switch pitalice
- metode imaju istovrsen uvjetne izraze koji ispituju stanje objekta
- OO smješta uvjetne grane u izdvojeni razred
    - time omogućava dinamičku konfiguraciju ponašanja delegacijom

sudionici

- **Kontekst**
    - definira sučelje koje koriste klijenti
    - održava referencu na konkretni razred koji definira trenutno stanje
    - u primjeru s programom za crtanje, kontekst je bio sam program za crtanje
- **Stanje**
    - deklalira sučelje za enkapsuliranje ponašanja koje ovisi o trenutnom stanju
    - u primjeru s programom za crtanje - sučelje Tool
- **Konkretno stanje**
    - svaka klasa koja implementira sučelje stanje
    - te definira skup ponašanja vezanih uz pojedinačno stanje
    - u primjeru s programom za crtanje - AddCircleTool, AddSquareTool, EditTool



suradnja

- kontekst - delegira zahtjeve trenutnom Konkretnom stanju
    - može poslati sebe kao argument metode Konkretnom stanja
- klijenti
    - mogu konfigurirati Kontekst s početnim stanjem
    - kasnije klijenti ne barataju izravno s konkretnim stanjem
- daljnje prijelaze stanja vrši kontekst ili konkretna stanja

implementacija

- tko definira prijelaze stanja?
    - često sam kontekst, ponekad i sama stanja

MOST

strukturni obrazac

namjera

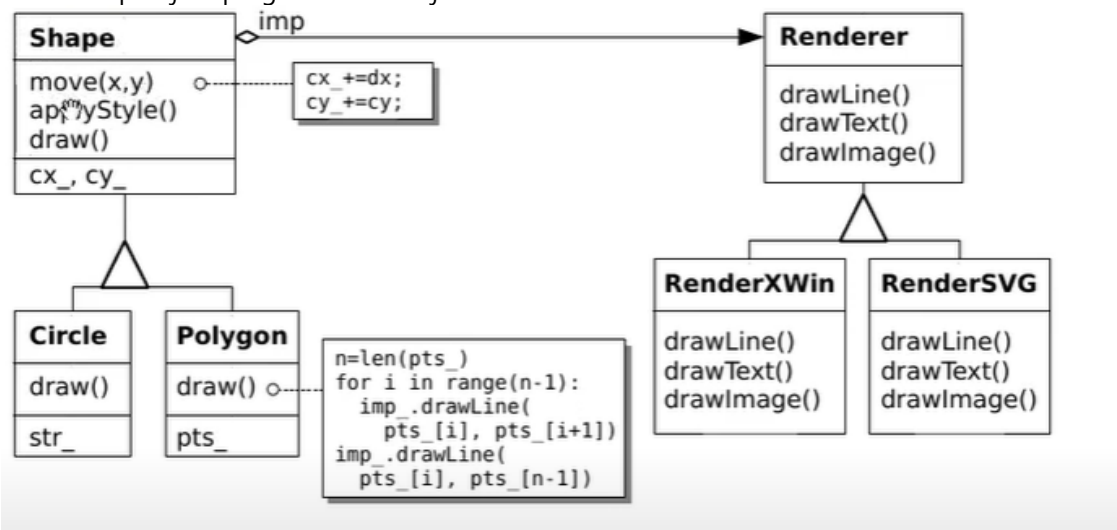
- odvojiti obitelji komponenata od implementacijskog detalja tako da se oboje mogu mijenjati nezavisno

primjer

- u našem programu za crtanje želimo (u isto vrijeme!)
    - dodavati nove likove
        - bez da mijenjamo rezultate prikaza
    - omogućiti različite načine iscrtavanja tih likova
        - npr. jedan način - crtanje na prozoru, drugi način - crtanje u SVG format...
        - dakle želim transparentno dodavati nove načine crtanja
            - bez da mijenjam funkcionalnost dodavanja novih likova
- dakle želimo da os programa koja se bavi dodavanjem novih likova bude nezavisna od osi programa koja se bavi crtanjem tih likova

za labos: crtanje likova na https://youtu.be/s9pf6aZBuOo?list=PLkOLgurQ4FfMl7IhtSTImxdbe0o5RRtpo&t=1122

na našem primjeru programa za crtanje:



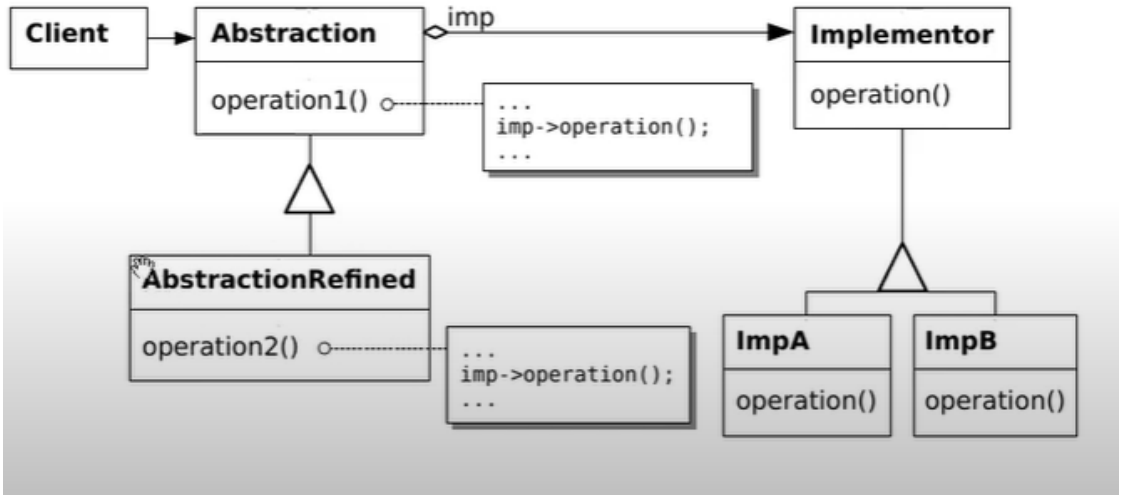
- primijeti - desna strana (sučelje Renderer i razredi izvedeni iz njega) nema pojma da lijeva strana postoji

- on samo nudi neke metode
- lijeva strana zna da desna postoj, ali ne zna za konkretne implementacije
    - vidiš da lijeva strana ima referencu na neki konkretni Renderer
        - ali ga gleda kroz naočale sučelja Renderer
    - omogućava veliku prenosivost i dodavanje novih načina crtanja

primjenjivost

- želimo izbjeći čvrsto vezanje komponente s izvedbenim detaljem
    - kad izvedbeni detalj treba mijenjati tijekom izvršavanja
    - kad klijente treba izolirati od izvedbenih detalja
- može se koristiti kao alternativa višestrukom nasljeđivanju

dijagram



- lijeva hijerarhija sadrži izvođača iz desne hijerarhije
    - dakle ima referencu na primjerak jedne implementacije desne hijerarhije
    - ali opet, gleda ga kroz sučelje Implementator
- operation() je drawLine i drawDashedLine iz primjera za crtanje

sudionici

- **Apstrakcija (sučelje(apstraktni razred) Shape)**
    - definira sučelje prema klijentima
    - sadrži referencu na jednu konkretnu implementaciju Izvođača
- **Prilagođena (konkretna) apstrakcija (Polygon)**
    - konkretni geometrijski likovi
    - proširuje sučelje Apstrakcije (konkretne implementacije metoda)
    - sadrži referencu na jednu konkretnu implementaciju Izvođača (jer to sadrži apstrakcija)
- **(apstraktni) Izvođač (sučelje Renderer)**
    - definira sučelje za implementacijske razrede
- **Konkretni izvođač (npr. RendererSVG)**
    - izvodi sučelje Izvođača, definira konkretnu implementaciju
    - ne zna da postoji lijeva strana

suranja

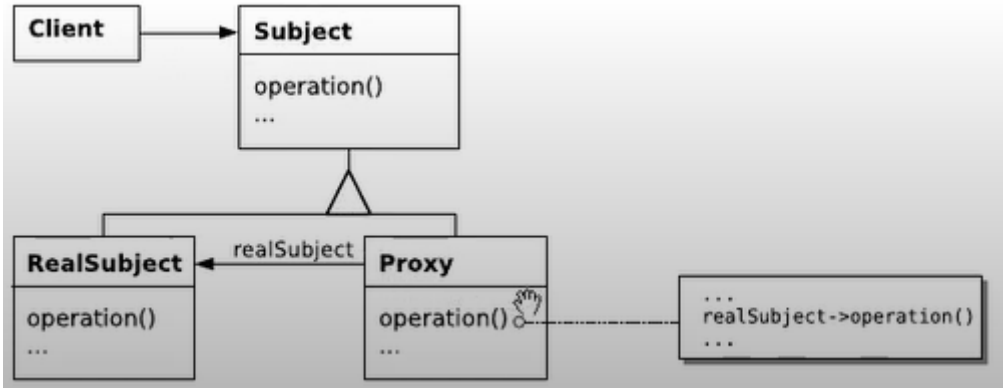
- apstrakcija proslijeđuje zahtjeve zadanom izvođaču

PROXY (ZAMJENSKI OBJEKT)

strukturni obrazac (proxy umata stvarni objekt)
omogućava softiciran pristup resursu

ideja

- klijent želi koristiti komponente (subjekti) preko reference na apstraktno sučelje koje modelira dotičnog subjekta
    - taj subjekt nudi odgovarajuću operaciju (metodu, npr. operation)
- imamo par izvedeni razreda - npr. RealSubject
- prilikom stvaranja objekta s kojim će raditi kljent, nekad je interesantno klijentu poslati ne referencu na RealSubject, nego na razred Proxy, koji također nasljeđuje sučelje Subject
    - a prilikom poziva nad razredom Proxy on ima referencu na primjerak stvarnog subjekta i njemu prepušta izvođene operacija
- pa zašto klijent ne bi izravno pozivao operacije nad RealSubject? Zašto uvodimo stupanj indirekcije?
    - primjerice
        - **udaljeni proxy**
            - nemamo dovoljno resursa da izvodimo RealSubject (recimo da je operation() memorijski zahtjevna operacija)
                - podvalimo mu Proxy, jer isto nasljeđuje sučelje Subject
                - prilikom pozivanja opertion(), on preko TCP veze poziva RealSubject koji se izvodi na udaljenom računalu
                - ovime smo dobili mogućnost da
                    - a) nalazimo se na "jakom" računalu
                        - klijentu samo pošaljemo referencu na RealSubject
                    - b) nalazimo se na "slabom" računalu
                        - klijentu pošaljemo referencu na Proxy, koji se preko TCP-a spaja na server na kojem se vrti RealSubjet
                - prednost je što klijent ne ovisi o načinu na koji je to implementirano
        - **virtualni proxy**
            - stvaranje RealSubject je primjerice vremenski zahtjevno
            - želimo odgoditi stvaranje dok klijent ne zatraži operaciju nad RealSubject
            - podvalimo klijentu Proxy
                - Proxy stvara RealSubject tek kad klijent prvi put zatraži operation()
            - inače ovo je ono što fetchType.LAZY radi kod JPA
            - često u aplikacijama za obradu dokumenata
                - za lazy učitavanje slika npr.
        - **sigurnosni proxy**
            - omogućavaju da se pozivi samo odabranih klijenata proslijeđuju na stvarnog klijenta
            - primjerice, Proxy određuje ima li klijent autorizaciju, te ako ima, proslijedi na izvođenje RealSubject-u
        - cacheirajući, sinkronizacijski, podstrukturni...
            - podstrukturni je zanimljiv
                - recimo RealSubject nudi neku matricu A
                - možemo napraviti Proxy koji će klijentu nuditi samo dio te matrice A



- ova iscrtkana strelica desno dolje (od operation() u Proxy-u) označava delegaciju realSubjectu
- naravno, to ne mora biti prava referenca
  - jer recimo ako delegacija ide preko TCP-a, to nije prava programska referenca, ali je i dalje OO Proxy

primjer - pametni pokazivač u C++

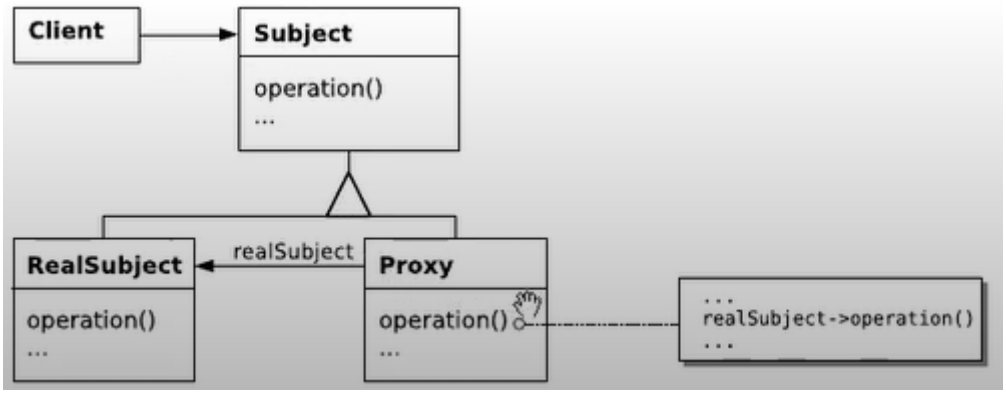
- želimo izgraditi omotač oko sirovog pokazivača da možemo imati dodatnu funkcionalnost

namjera

- zamjenski objekt upravlja pristupom stvarnom objektu
- uvodi dodatnu razinu indirekcije
- delegirajući omot pojednostavljuje implementaciju ciljnog subjetka
  - dakle, ciljni subjekt se mora baviti problematikom npr. cachiranja, autorizacije itd.
  - za taj dio možemo napisati različite proxye

sudionici

- **klijent**
  - zove metode sučelja Subject
- **sučelje subjekt**
  - deklarira metode
- **stvarni subjekt (RealSubject)**
  - implementira Subject, nudi konkretnu funkcionalnost
  - nema pojma da ga Proxy predstavlja
- **zamjenski objekt (Proxy)**
  - implementira Subject!
  - imaju (konceptualnu, ne nužno stvarnu) referencu na RealSubject
  - kontrolira pristup stvarnom subjetu
  - može biti odgovoran za stvaranje i brisanje



suranja

- Proxy kontrolira pristup RealSubjectu te mu proslijeđuje pozive

posljedice

- ovisi o vrsti proxya, ali primjerice
    - transparentan rad s objektima u različitom adresnom prostoru - udaljeni proxy
    - odgođeno kreiranje objekata - virtualni proxy
    - izvršavanje administrativnih zadataka prije/poslije pristupa objektu
        - **za razliku od dekoratora, zadaci nemaju veze s osnovnom odgovornošću komponente!!!!**
    - transparentan pristup podstrukturama kao samostalnim objektima - podstrukturni proxy
        - korisno za npr. računanje determinante, da nemoramo kopirati matricu X puta
- bolja preraspodjela odgovornosti u sustavu
- ali
- veća složenost programa, više objekata, sporiji pristup

srodni obrasci

- **adapter**
    - željeni objekt predstavlja pod **različitim** sučeljem
- **dekorator**
    - strukturno vrlo sličan osnovnoj verziji proxya
    - ali se prvenstveno razlikuju **u namjeri**

PROTOTIP

kreacijski obrazac

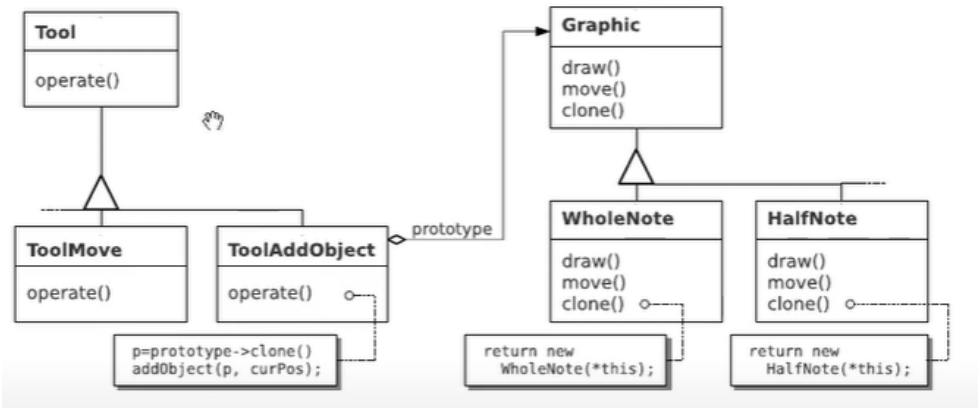
problem - kako podržati dodavanje novih klasa (koje recimo implementiraju neko sučelje) u aplikaciju ako ih aplikacija još nije vidjela?
tj. recimo želimo dodati novi lik u naš program za crtanje - kako osigurati da program zna koju klasu treba stvoriti?
rješenje - prilikom pokretanja programa damo mu listu prototipa - stvorimo po jedan primjerak svakog od objekata čije daljnje stvaranje mora biti podržano u našoj aplikaciji

- na temelju tog jednog objekta onda aplikacija mora moći stvoriti još x takvih

ideja - svi objekti čije primjerke želimo moći stvarati nude metodu za stvaranje novih primjeraka tog razreda

- ta metoda - metoda tvornica

namjera - odrediti razrdd novog objekta polimorfnim kloniranjem protopia
klijent samo zna da svi ti prototipi imaju metodu clone
on ne zna koji su konkretno razred ti objekti



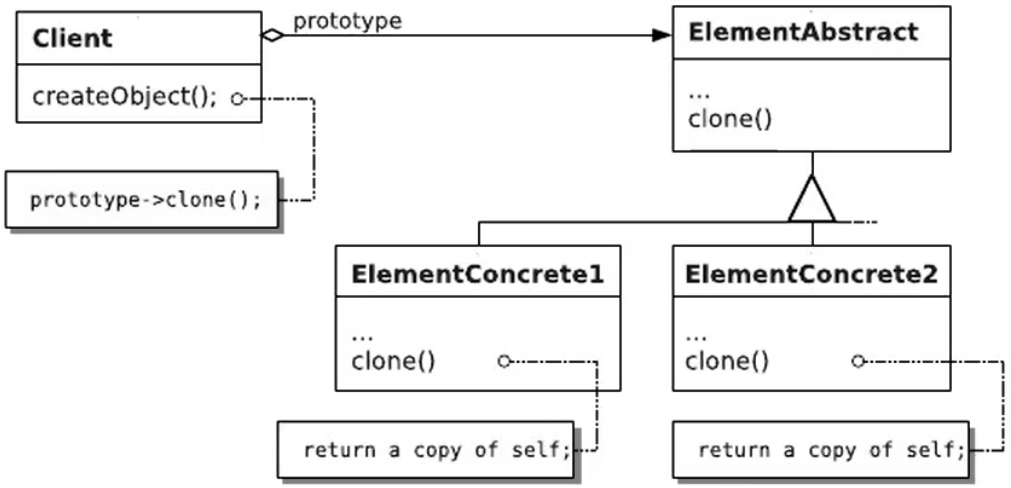
- primjer sličan primjeru sa alatom za crtanje

primjenjivost

- kad sustav mora biti neovisan o tome kako se elementi predstavljaju, komponiraju i kreiraju

sudionici

- **apstraktni element**
  - nudi sučelje za kloniranje
- **konkretni element**
  - implementira sučelje za kloniranjke
- **klijent**
  - održava pokazivač na aktivni prootip apstraktnog elementa
  - instancira nove elemente kloniranjem prototipa



suradnja

- klijent poziva operaciju kloniranja nad prototipom

posljedice

- skriva konkretne razrede od klijenata i tako pospješuje nadogradivost bez promjene
- omogućuje klijntima da instanciraju naknadno razvijene elemente (npr, iz dinamičkih biblioteka)
- prototip može biti i složeni objekt (kompozit, dekorator...)
    - kad pozovemo metodu clone, što ćemo klonirati?
    - složeni objekti imaju i reference na druge objkete - hoćemo li i njih klonirati?

srodni obrasci su npr. tvornice

POSJETITELJ (VISITOR)

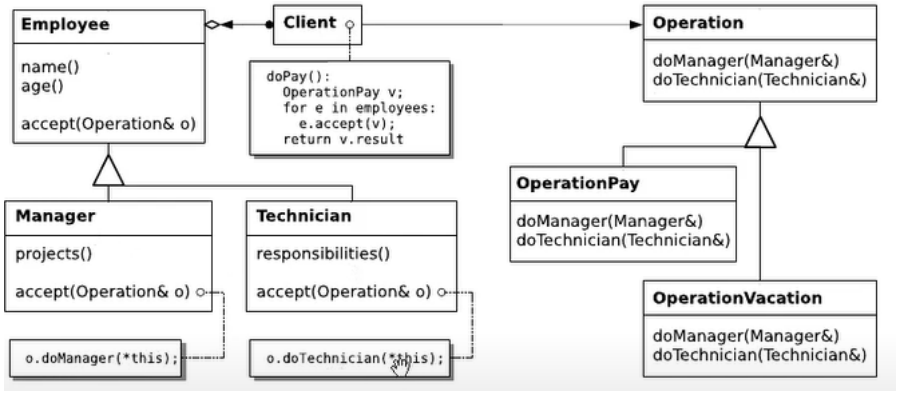
ponašajni obrazac (lijeva porodica prima objekt desne porodice)

namjera

- modelirati operacije nad polimorfnim elementima skupnog objekta bez promjene sučelja
    - kad operacije ovise o konkretnom razredu
- polimorfni poziv ovisi o konkretnom razredu 2 objekta (visitora i objekta koji posjećujemo)
    - dakle, npr. PrintVisitor i TextNode
    - ovo se naziva double dispatch

primjer

- imamo sučelje Employee koje modelira zaposlenike (Manager, Techician) itd.
- želimo nad njima provoditi neke operacije
- želimo te operacije transparentno moći dodavati u budućnosti **bez mijenjanja koda Manager, Techician itd.**
- predlažemo ovakvo rješenje



- u sučelje Employee smo dodali metodu accept(Operations o)
    - Operation je razred koji modelira apstraktnog visitora
    - u sebi ima metodu visitManager, visitTechnician..., odnosno po jednu visit metodu za svaki konkretan razred izveden iz Employee, koja određuje na koji način treba obraditi taj razred
- konkretni poslovi koje želimo obavljati (OperationPay, OperationVacation)I su konkretni visitori
    - oni za konkretne razrede (Manager, Techician), određuju kako ih taj konkretni visitor (npr. OperationPay) treba obraditi
- double dispatch ostvarujemo tako da nad svakim konkretnim Managerom, Techicianom, itd. pozovemo accept(Operation o), gdje je o neki konkretan visitor (npr. OperationPay)
    - onda metoda accept u npr. Manager razredu poziva o.doManager, odnosno poziva odgovarajuću metodu visitora za razred Manager
    - u razredu Techician poziva se doTechician itd.

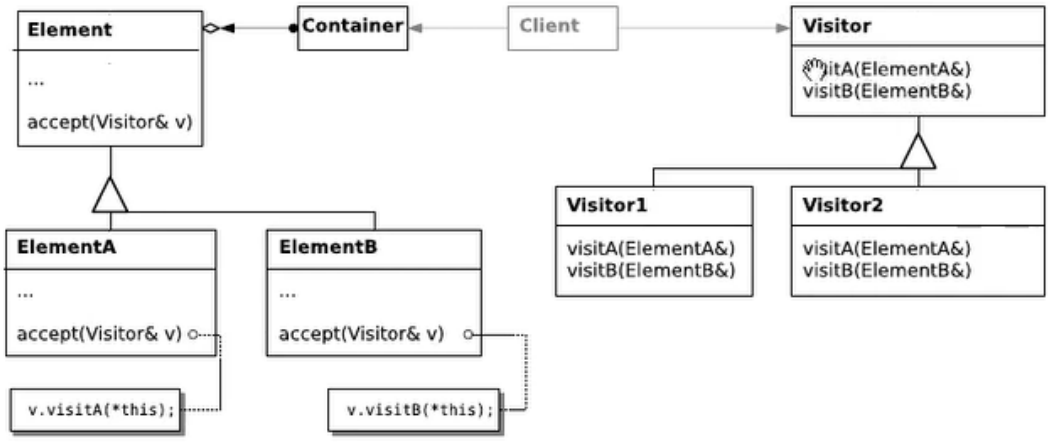
- na ovaj način u budućnosti lako možemo dodavati nove načine obilaska, odnosno nove poslove
    - bez da mijenjamo kod Employeea

primjenjivost

- kad polimorfni poziv ovisi o dva konkretna razreda
    - npr. u gornjem primjeru koja će se točno metoda pozvati ovisilo je o konkretnom Employeeu i konkretnom visitoru
- kad operacija ima mnogo pa ih ne želimo gurati u sučelje elemenata (sučelje Employee npr.)
- **kad su razredi elemenata stabilni** (dakle najvjerovatnije nećemo imati nove implementacije sučelja Employee), dok se mogu pojaviti nove operacije (OperationPay, OperationVacation, pa dodamo npr. OperationPromote)

sudionici

- **Posjetitelj/Visitor** (npr. Operation u gornjem primjeru)
    - deklarira po jednu varijantun operacije za svaki **konkretan** objekt
        - dakle operacija za ElementA, operacija za ElementB itd.
- **Konkretni posjetitelj** (u primjeru OperationPay)
    - modelira konkretnu operaciju, čuva kontekst, izvodi varijante te akumulira rezultat
- **Element** (u primjeru Employee)
    - deklarira sučelje za konkretne elemente, te obavezno metodu accept
- **Konkretni element** (u primjeru Manager i Techician)
    - izvodi metodu accept tako da nad visitorom pozove odgovarajuću metodu (v.acceptManager itd.)
- **Klijent**
    - kreira odgovarajućeg posjetitelja
    - iterativno ga šalje elementima



suradnja

- klijent mora
    - instancirati konkretni posjetitelj
    - proći kroz sve elemente kolekcije

posljedice

- lako dodavanje novih operacija dodavanjem novog posjetitelja

    - pospješuje se načelo jed. odgovornosti
- elementi moraju izložiti sve relevantne dijelove svog stanja
- teško dodavanje novih konkretnih elemenata!
    - potrebne izmjene u svim konkretnim posjetiteljima
    - jer svaki konkretan posjetitelj ovisi o konkretnom elementu
    - dakle, ako se često dodaju novi elementi domene ovo nije baš najsretniji obrazac

implementacija

- pojedine metode visitora mogu imati iste ili različite nazive
    - ako imaju isto ime (npr. sve se zovu visit), onda se razlikuju po argumentima (npr, jedna prima Manager m, druga Techician t itd.)
- tko je zadužen za obilazak djece kompozita?
    - često posjetitelj, no onda kompozit mora izložiti metode za dohvat djece
    - a može i kompozit nad djecom pozivati accept (delegirati poziv njima)

primjene

- prevoditelji
    - za opis višestrukih operacija nad sintaksnom strukturom programa
    - sjeti se vistora u onoj zadaći s HTTP poslužiteljem iz OPRPP2
- operacije nad elementima vektorskog crteža
    - iscrtavanje, pretraživanje...
- operacije nad stablom datotečnog sustava

srodni obrasci

- često se koristi u kombinaciji s Kompozitom te Iteratorom
- Most
    - sličnosti - isto ima lijevu i desnu hijerarhiju, kao i Most
    - razlike
        - kod Posjetitelja, konkretni elementi primaju posjetitelje preko argumenata metode accept
        - Most je imao metode i referencu na konkretni tip desne strane koji će koristiti
        - konkretni posjetitelji znaju za konkretne elemente (dakle ovdje desna strana zna za lijevu)
            - kod Mosta, lijeva strana nije imala pojma da desna postoji
            - zato je Most bolji kad se lijeva strana često mijenja
                - jer desna strana ne zna za lijevu, i Most omogućava da se desna i lijeva strana neovisno mijenjanju
- https://youtu.be/ed_ZuvrDRml?list=PLkOLgurQ4FfMl7lhtSTImxdbe0o5RRtpo
    - od 1:31:00 priča o 4. labosu i zašto rješenje nije pravi Most, ali ni pravi Posjetitelj
    - tj. tehinčki je most, ali lijeva strana nema stalnu referencu na Renderer, nego prima tu referencu kroz metodu (slično kao visitor)
    - ali nije visitor jer desna strana (Renderer) ne zna za lijevu stranu
    - ovo smo napravili iz razloga što želimo malo crtati oblike na ekranu, malo ih eksportati npr. u SVG

METODA TVORNICA

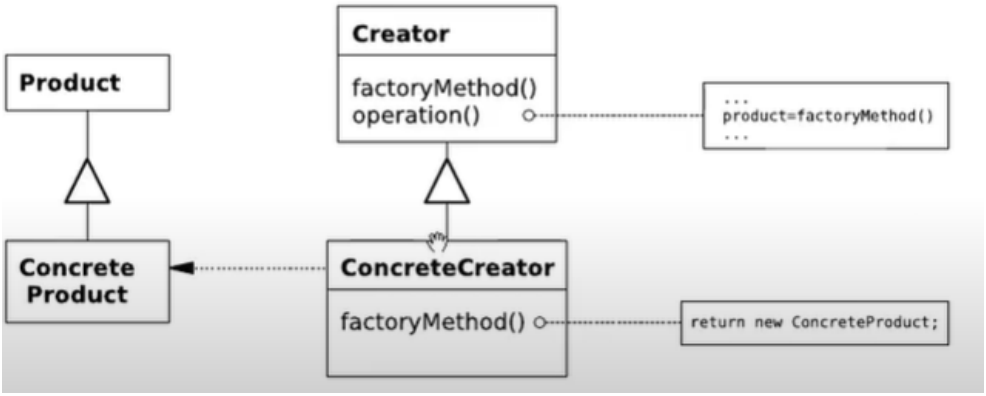
kreacijski obrazac (kreator stvara prikladne proizvode)

dopiši primjer sa početka slajdova, ili bolje Čupkov primjer

primjenjivost

- osnovni razred (Kreator) instancira objekte (na njih gleda kroz sučelje Proizvod) čiji konkretni razred nije poznat
    - dakle, ima metodu za stvaranje objekata tipa Proizvod, koja je apstraktna
- izvedeni razred (Konkretni kreator) definira konkretni tip objekta (Konkretni proizvod) kojeg osnovni razred kreira
    - dakle ima konkretne implementacije metoda za stvaranje Konkretnih proizvoda

dijagram



- fleksibilnost se ostvaruje nasljeđivanjem
- Creator - abstraktno sučelje, ima i druge metode
- ConcreteCreator - stvara ConcreteProduct

sucionici

- **Proizvod**
    - definira sučelje za objekte koje stvara metoda tvornica
- **Konkretan proizvod**
    - implementira sučelje Proizvoda
- **(abstraktni) Kreator**
    - deklarira metodu tvornicu koja vraća objekt razreda Proizvod
    - može pozvati metodu tvornicu kako bi instancirao novi Proizvod
- **Konkretni kreator**
    - izvodi metodu tvornicu koja instancira Konkretni proizvod

suradnja

- kreator izvedenim razredima prepušta izvedbu metode tvornice, odnosno instanciranje odgovarajućeg Konkretnog proizvoda

MVC

arhitektonski programski obrazac

primjer

- onaj Čupkov primjer sa prikazom krugova
    - model je razred koji sadrži popis svih krugova, te opcije za dodavanje krugova

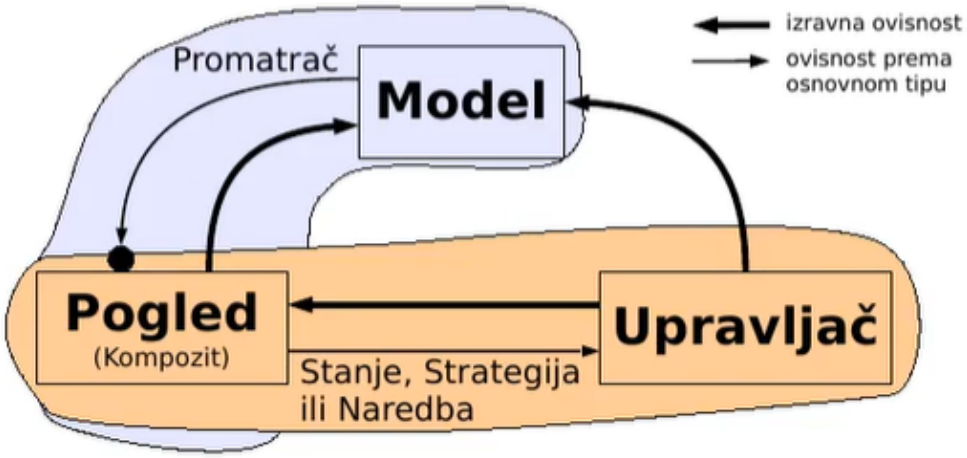
        - također je subjet u OO promatrač koji svojim promatračima javlja promjene u modelu
        - **ne zna** (bar po točnom tipu) za konkretan view
    - view je JComponent za prikaz tih krugova na ekranu,implementira sučelje CirclesChangesListener kako bi znala kad je došlo do promjene u modelu
    - controller je tipka za dodavanje i brisanje elemenata crteža

primjenjivost

- kad želimo međusobno izolirati logiku programa, korisničko sučelje i prikaz
- cilj je lakše održavanje programa

često kombinacija Promatrača, Stanja i Kompozita

- Stanje se često koristi za modeliranje upravljača
    - primjerice u 4. labosu - oni gumbi za dodavanje, brisanje, selektiranje itd. su modelirani preko OO Stanje
        - ali oni su također Upravljač u OO MVC!



- upravljač ne mora imati vezu na pogled
- model vidi pogled kroz sučelje promatrača (načelo inverzije ovisnosti jej)
- pogled može ali i ne mora imati referencu prema promatraču

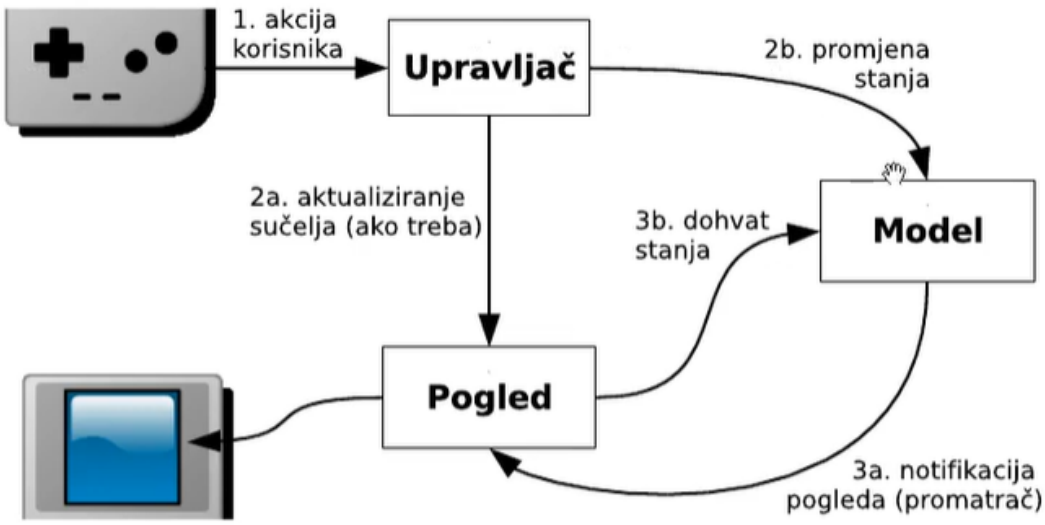
sudionici

- **Model**
    - sadrži podatke aplikacije
    - te pravila (metode) za njihovu manipulaciju
- **Pogled (View)**
    - prikazuje aplikacijske podatke na koristan i praktičan način
    - može prikazivati i elemnte GUI-a (dakle elemente modela!)
    - može biti više pogleda (istovremeno prikazanih ili ne)
- **Upravljač (Controller)**
    - odgovoran za komunikaciju s korisnikom

suradnja

- upravljač radi sinkrono s korisnikom
    - šalje korisničke akcije modelu, a model pri promjeni pogledu
- model evoluira ovisno o korisničkim akcijama (npr. korisnik dodaje elemente u crtež)

    - a može i neovisno o korisničkim akcijama (npr, web aplikacije gdje je model baza)
- pogled prikazuje elemente modela sinkrono s modelom
    - te elemente GUI-a sinkrono s korisnikom



posljedice

- smanjuje se arhitektonska složenost odvajanjem modela od upravljača i pogleda
- mogućnost višestrukih pogleda
- mogućnost različite obrade korisničkih akcija (recimo OO Strategija)
- nedostatak
    - složenost

implementacija

- upravljač može biti jedan od promatrača

primjene

- arhitektura programa s GUI-om
- kod web aplikacija sljedeća struktura
    - view - HTML stranica
        - ili npr. .jsp u Javi
    - controller - skripta koja prikuplja podatke od korisnika
        - u skladu s modelom generira HTML
        - ili npr. Servleti u Javi
    - model - podaci spremljeni u bazi podataka
    - ovdje pogled generira upravljač