Oblikovni obrasci u programiranju

Rješenja učestalih oblikovnih problema

Siniša Šegvić

Sveučilište u Zagrebu

Fakultet elektrotehnike i računarstva

Zavod za elektroniku, mikroelektroniku
računalne i inteligentne sustave

Sadržaj

Vrlo kratko, općenito o obrascima

Najčešće korišteni obrazac: Strategija

Osnovni obrasci:

- Okvirna metoda, Promatrač, Dekorator, Naredba

Ostali obrasci:

- Jednostavne tvornice, Jedinstveni objekt, Iterator, Prilagodnik, Kompozit, Stanje, Proxy, Most, Prototip, Posjetitelj, Metoda tvornica, Model-pogled-upravljač, Apstraktna tvornica

Obrasci: primjer

Interaktivni program s nadogradivim pribavljanjem i obradom slike:

```
class ComputerVisionShell{
  //...
  vs_base* pvs_;   // video source (getFrame)
  alg_base* palg_; // image processing algorithm (process)
  std::vector<win_wrap> wins_;

  void processOne(){
    img_wrap image;
    pvs_->getFrame(image);
    palg_->process(image);

    wins_.resize(palg_->nDst());
    for (int i=0; i<palg_->nDst(); ++i){
      wins_[i].putFrame(palg_->imgDst(i));
    }
  }
}
```

Pristup ostvarivanja nadogradivosti: **povjeriti** (delegate) dio posla vanjskom pružatelju preko pokazivača na njegov osnovni razred

Obrasci: generalizacija

Prikazani pristup prikladan kad god je potrebno **dinamički** mijenjati postupke koji se primijenjuju u aplikaciji

Vidimo da postoji potencijal za ponovno korištenje pristupa: želimo ga dokumentirati da ga što bolje iskoristimo!

I tako se susrećemo s idejom **oblikovnog obrasca** (naziv: strategija):

- "općenito rješenje učestalog oblikovnog problema"
  (problem: odvojiti klijenta od izvedbi algoritama)
- "široko primjenljivi organizacijski element"
  (ostvariti ortogonalne osi promjenljivosti modula)

Obrasci: uvod

Prednosti obrazaca:

- oblikovanje je teško: želimo zapamtiti **dokazana** rješenja
- produktivnost oblikovanja, obogaćeni rječnik, laka komunikacija

Kako koristiti obrasce:

- obrasci komplementarni **bibliotekama** (funkcionalnost vs. organizacija)
- primjena: osmišljavanje novog, prekrajanje starog kôda
- recepti za dobro organiziranje programa javljaju se na različitim **razinama** (idiomi, oblikovni obrasci, arhitektonski obrasci)

Obrasci: zrnatost

- Arhitektonski obrasci: struktura programa na najvišoj razini
  - najviša razina apstrakcije, organizacija podsustava (10 KLoC)
  - model-pogled-upravljač (model - view - controller)
  - klijent-poslužitelj (client - server)
  - ravnopravna mreža (peer-to-peer network)
  - slojevi (presentation - business logic - data)
  - oglasna ploča (blackboard)
- Oblikovni obrasci:
  - srednja razina apstrakcije, organizacija komponenti (1 KLoC)
  - predmet ovog kolegija
- Programski idiomi:
  - organizacija sastavnih dijelova komponente (1-100 LoC)
  - idiomi su uglavnom beznadno ovisni o jeziku:
    - `http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html`
    - `http://en.wikibooks.org/wiki/More_C++_Idioms`

Obrasci: osnovni opis

Oblikovni obrazac dokumentira aspekte rješenja oblikovnog problema

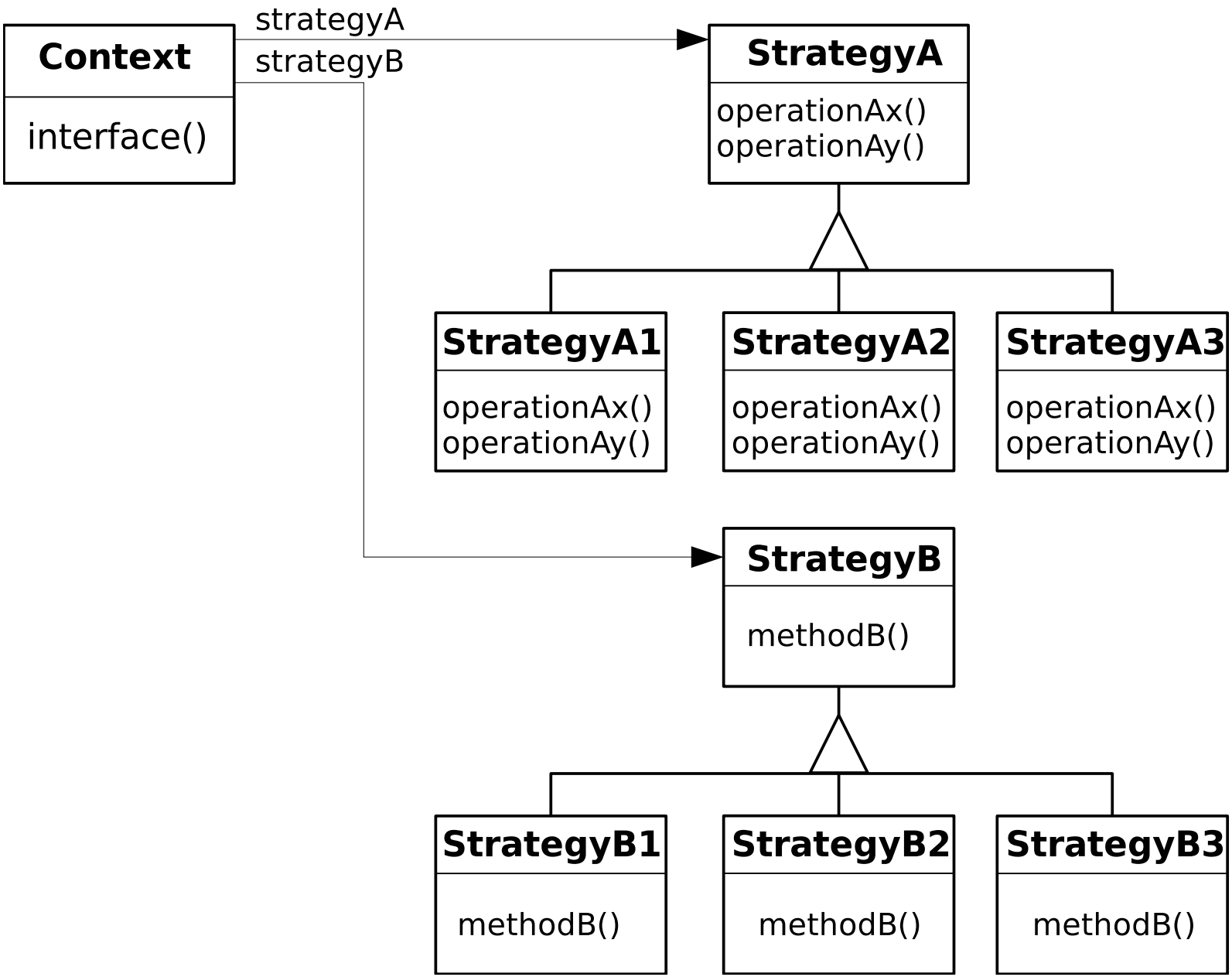
Ključne komponente opisa obrasca oblikovanja:

- **kratki naziv**:
  - obogaćuje oblikovni rječnik (razmišljanje, komunikacija)
  - pospješuje oblikovanje na višem stupnju apstrakcije
- **problem**: kad obrazac ima smisla primijeniti?
- **rješenje**: apstraktni opis, ne konkretno oblikovanje
  - opis sudionika koji čine obrazac
  - odnos među sudionicima (razdioba odgovornosti, suradnja)
- **rezultat**: što postižemo primjenom obrasca?
  - prednosti, nedostatci, kompromisi
  - utjecaj na **nadogradivost**, **proširivost**, višestruko korištenje

Strategija: osnovni opis

- **Problem**:
  Strategiju koristimo kad treba dinamički mijenjati ponašanje neke komponente (konteksta). Ponašanje zadajemo odabirom postupka, bez potrebe za mijenjanjem izvornog kôda konteksta. Obrazac je posebno koristan ako imamo više nezavisnih obitelji postupaka.
- **Rješenje**:
  Definirati hijerarhiju(e) postupaka, enkapsulirati konkretne postupke, omogućiti biranje zadavanjem željenog postupka. Kontekst povjerava zadatke konkretnim postupcima preko roditeljskog sučelja.
- **Rezultat**:
  Pospješuje se **nadogradivost bez promjene** (dodavanje novih postupaka), **inverzija ovisnosti** (kontekst ovisi o apstraktnom sučelju), te **ortogonalnost** (razdvajanje postupaka i konteksta).

Strategija: strukturni dijagram



Obrasci: potpuni GoF opis

- **Naziv**, alternativni nazivi (strategy, policy)
- **Problem**: namjera, motivacijski primjer, primjenljivost
- **Rješenje**: struktura, sudionici, suradnja
- **Rezultat**: posljedice, implementacijska pitanja, izvorni kod, primjeri upotrebe, srodni obrasci

Strategija: namjera

- Definirati obitelj(-i) međusobno izmjenljivih algoritama
- omogućiti izmjenu algoritma neovisno o klijentu(-ima) koji algoritme pozivaju

Strategija: primjenljivost

- dinamička konfiguracija konteksta sa željenim postupkom (školski primjer nadogradivosti bez promjene)
- potrebne su različite varijante istog postupka (npr. s obzirom na kompromis prostor-vrijeme)
- odvajanje konteksta od implementacijskih detalja postupka
- različita ponašanja se odabiru virtualnim pozivom: dinamički polimorfizam umjesto krutog odabira grananjem
- ako dinamička konfiguracija nije presudna $\Rightarrow$ statički polimorfizam (spremnici STL-a: strategija je alociranje memorije)

Strategija: sudionici

- **strategija** (pribavljanje slike)
  - deklarira zajedničko sučelje svih podržanih postupaka
  - preko tog sučelja kontekst poziva konkretne postupke
- **konkretna strategija** (pribavljanje slike iz datoteke avi)
  - implementira postupak preko zajedničkog sučelja
  - često je jedna od strategija nul-objekt koji ne radi ništa (to je korisno za ispitivanje)
- **kontekst**
  - konfigurira se preko pokazivača na konkretnu Strategiju
  - može definirati sučelje preko kojeg Strategije mogu pristupiti njegovim podatcima

Strategija: suradnja

- **prijenos parametara** između Konteksta i Strategije
  - izravno slanje parametara (*push*)
  - neizravno slanje (*pull*): Kontekst šalje sebe, Strategija uzima što joj treba (međuovisnost)
- **konfiguracija** Konteksta s konkretnom Strategijom
  - klijenti Konteksta odgovorni za postavljanje konkretne Strategije
  - kasnije, klijenti komuniciraju isključivo s Kontekstom
  - klijent često odabire iz čitave obitelji konkretnih Strategija
  - najveća korist: potreba za ortogonalnim obiteljima postupaka (ortogonalna konfiguracija, nadlinearni rast funkcionalnosti)

Strategija: posljedice

- obrascem modeliramo obitelji opcionalnih postupaka
  - ako postoji zajednička funkcionalnost, može se izdvojiti u apstraktnu strategiju
- podatnija **alternativa** nasljeđivanju:
  - razdvajanje konteksta od ponašanja
  - mogućnost ortogonalnog skupa postupaka
- eliminacija uvjetnih izraza
- veća **fleksibilnost** i složenost u odnosu na alternative (nasljeđivanje vs. povjera).

Strategija: implementacijska pitanja

Komunikacija između Konteksta i Strategije (izravan i neizravan prijenos parametara)

Statički ili dinamički polimorfizam?

Mogućnost specijalnih strategija:

- podrazumijevano ponašanje
- ispitna funkcionalnost: imitacijski (*mock*) objekt
- bez funkcionalnosti: nul-objekt

Strategiju u C-u možemo izvesti uz pomoć:

- strukture pokazivača na funkcije (`fs_library_vtable_t` u `libsvn`u)
- golog pokazivača na funkciju
  - argument `compar` funkcija `qsort` i `bsearch`

Strategija: izvorni kôd C++

Mogućnost izravnog i neizravnog prijenosa parametara konteksta u sučeljnu metodu strategije:

```
//Strategy.hpp
class Context;
class Strategy{
  virtual int push(int param) =0;
  virtual int pull(Context& c) =0;
};
//Context.hpp
class Context{
  //...
public:
  void doSomething();
  int state() const {return state_;}
private:
  Strategy& a_;
  int state_;
};
//Context.cpp
void Context::doSomething(){
  a_.push(state_);
  a_.pull(*this);
}
```

```
//StrategyConcrete.hpp
class StrategyConcrete:
  public Strategy
{
  virtual int push(int param);
  virtual int pull(Context& c);
};
```

```
//StrategyConcrete.cpp
int StrategyConcrete::push(
  int param)
{
  return param%42;
};
int StrategyConcrete::pull(
  const Context& c)
{
  return c.state()%42;
}
```

Strategija: izvorni kôd Python 3.x

Strategije često koristimo za zadavanje kriterija sortiranja:

```
>>> djeca=['špiro', 'boris', 'zvjezdana']

# obično sortiranje
>>> sorted(djeca) #bzš

# sortiranje po hrvatskim pravilima
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "hr_HR")
>>> sorted(djeca, key=locale.strxfrm) # bšz

# sortiranje preokrenutih riječi po hrvatskim pravilima
>>> sorted(djeca, key=lambda x: locale.strxfrm(x[::-1])) # zšb

# tko je najmlađi?
>>> djeca={'špiro':7, 'boris':9, 'zvjezdana':8}
>>> min(djeca, key=djeca.get) # š
```

Koje načelo oblikovanja poštuju funkcije `sorted` i `min`?

U Pythonu funkcijama možemo riješiti mnoge zadatke za koje u Javi ili C++-u moramo koristiti razrede!

Strategija: primjeri upotrebe

- **obrada teksta**: različiti načini lomljenja redaka
- **optimizirajući prevodioci**: različite tehnike za (i) dodjeljivanje registara, (ii) redosljed instrukcija, ...
- **logička sinteza**: postupci provlačenja vodova između elektroničkih elemenata
- **STL**: načini alociranja memorije standardnih spremnika
- **GUI biblioteke**: validacija korisničkog unosa u dijalozima (podrazumijevani postupak je bez validacije)
- korištenje strategija izvan OOP: callback funkcije (qsort), kontekstne funkcije (engl. closure)

Strategija: klasifikacija

Strategija: **ponašajni** obrazac u domeni **objekata**

Fleksibilnost se ostvaruje:

- kombinacijom povjeravanja i nasljeđivanja
- povjeravanjem parametru predloška

Povjeravanje omogućava **dinamičku** konfiguraciju

Predlošci pružaju manju fleksibilnost (statičko povezivanje), ali uz **maksimalnu performansu**

Obrasci: podjela

Na najgrubljoj razini, oblikovne obrasce možemo podijeliti u tri kategorije

- **Ponašajni** obrasci:
  - kako postići sofisticiranu funkcionalnost suradnjom objekata i razreda
- **Strukturni** obrasci:
  - kako od jednostavnih objekata sastavljati složenije ili hijerarhijske strukture
- Obrasci **stvaranja**:
  - kako prepustiti odgovornost za stvaranje konkretnih objekata posebnom objektu ili izvedenom razredu.

Obrasci: sažetak

- Obrasci su iskušani recepti za organiziranje komponenti
- Rezultat: fleksibilni, nadogradivi i ponovo iskoristivi kôd
  (u skladu s načelima oblikovanja)
- Obrasci dokumentiraju načine kako optimalno zadovoljiti načela oblikovanja kod klasa učestalih problema
- Konačno, obrasci pospješuju razumljivost, komunikaciju, dokumentaciju
- Nikad ne zaboraviti: temeljni zadatak programskog inženjera je **obuzdavanje složenosti**
  (lakmus papir za kritičku ocjenu bilo kojeg pristupa ili tehnologije)

Okvirna metoda: namjera

Definirati okvirni postupak koji neke korake prepušta izvedenim razredima.

Okvirna metoda pruža izvedenim razredima mogućnost promjene pojedinačnih koraka postupka uz zadržavanje osnovne strukture

Obrazac ćemo ilustrirati na primjeru komponente koja modelira glavnu petlju računalne igre za jednog igrača.

Okvirna metoda: motivacijski primjer

```
class GameWithTemplateMethod {
  //...
protected:
  virtual void init() =0;
  virtual void makeMove() =0;
  virtual bool endOfGame() =0;
  virtual void printWinner() =0;
public:
  // A template method:
  void playOneGame() {
    init();
    while (!endOfGame()){
      makeMove();
    }
    printWinner();
  }
};
```

```
class GameStrategy {
  virtual void init() =0;
  virtual void makeMove() =0;
  virtual bool endOfGame() =0;
  virtual void printWinner() =0;
};
class GameWithStrategy {
  GameStrategy* pgame_;
  // ...
  void playOneGame() {
    pgame_->init();
    while (!pgame_->endOfGame()){
      pgame_->makeMove();
    }
    pgame_->printWinner();
  }
};
```

- Apstraktni razred definira zajedničku strukturu toka izvođenja svih izvedenih razreda (pospješuje se **ortogonalnost**!)
- Izvedbe koraka okvirne metode definiraju izvedeni razredi
  - nazivamo ih apstraktnim primitivima i nadomjestivim metodama
- Veća **jednostavnost** u odnosu na Strategiju (manja prilagodljivost)

Okvirna metoda: primjenljivost

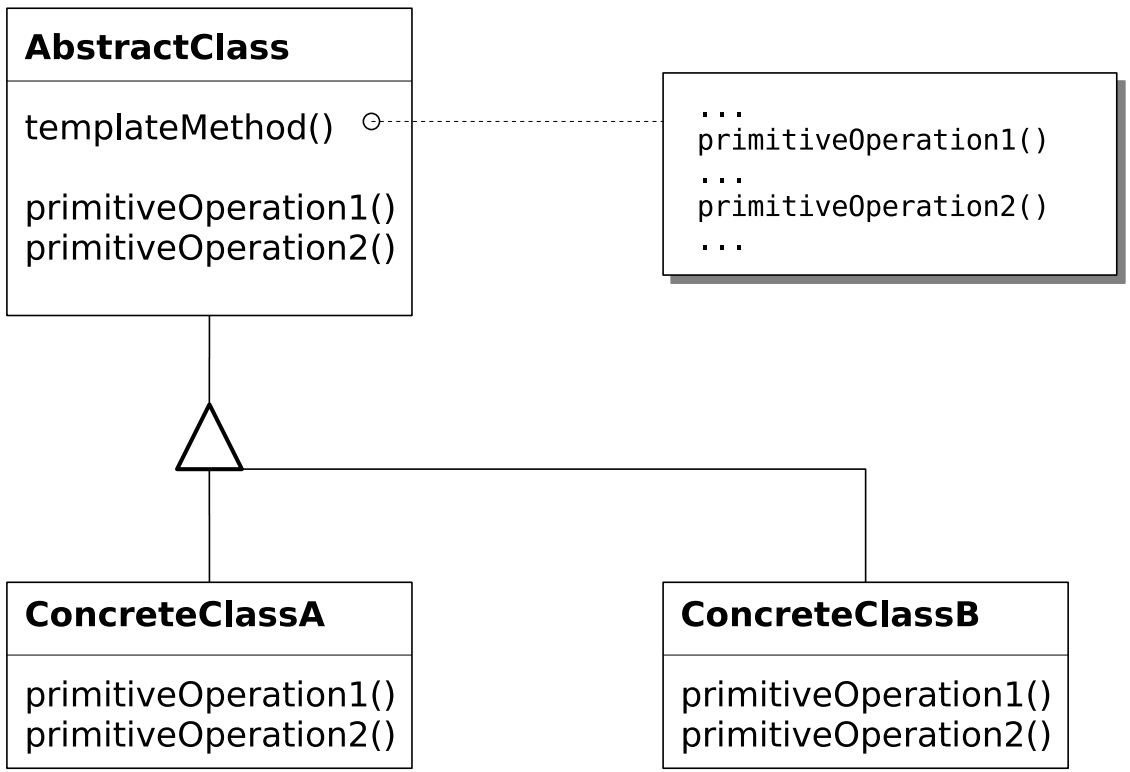
Okvirna metoda koristi se kad:

- istu grubu strukturu postupka (**okvir**) dijeli više konkretnih razreda
- stalne dijelove postupka valja prikupiti na jednom mjestu, dok izvedeni razredi definiraju promjenljive korake
- zajedničko ponašanje izvedenih razreda želimo izdvojiti da izbjegnemo dupliciranje kôda
- izvedeni razredi trebaju mijenjati osnovnu funkcionalnost:
  - osnovnu funkcionalnost smještamo u **nadomjestive metode** (*engl.* hooks) koje poziva okvirna metoda
  - osnovni razred pruža podrazumijevanu implementaciju (koja može biti i prazna)

Okvirna metoda: strukturni dijagram

Okvirna metoda (template method, self-delegation): **ponašajni** obrazac u domeni **razreda**

Prilagodljivost se ostvaruje nasljeđivanjem



Okvirna metoda: sudionici i suradnja

Sudionici:

- **Apstraktni razred** (zajednički dio svih igara)
  - deklarira **apstraktne primitive** u okviru kojih će izvedeni razredi definirati korake algoritma
  - izvodi **okvirnu metodu**, modelira grubu strukturu postupka
  - okvirna metoda koristi apstraktne primitive, te po potrebi ostale metode Apstraktnog razreda, odnosno metode ostalih objekata
- **Konkretni razredi** (labirint, konjićev skok, minesweeper, ...)
  - implementiraju primitive, izvode odgovarajuće korake postupka

Suradnja:

- Konkretni razredi se oslanjaju na implementaciju zajedničkih dijelova postupka u Apstraktnom razredu

Okvirna metoda: posljedice

Fundamentalna tehnika višestrukog korištenja korisna za:

- izdvajanje zajedničkog kôda iz skupa srodnih razreda koji modeliraju **jednu os promjene**
- ubacivanje klijentskog kôda u razrede biblioteke

Organizacijske prednosti:

- Izbjegava se potreba za repliciranjem kôda (**NJO**)
- Lako dodavanje novih konkretnih razreda (**NBP**)
- Klijenti ne znaju za konkretne razrede jer okvirnu metodu pozivaju preko osnovnog sučelja (**NIO**)

Okvirna metoda: implementacija

Okvirne metode pozivaju **apstraktne primitive** i **nadomjestive metode** osnovnog razreda

- izvedeni razredi *moraju* izvesti apstraktne primitive, za razliku od nadomjestivih metoda
- potrebno **dokumentirati** što je što, najbolje u okviru jezika:
  - okvirna metoda - običan poziv (C++) `final` (Java)
  - primitivi - zaštićene čiste virtualne metode
  - nadomjestive metode - zaštićene i virtualne
- C++: **`protected`**, **`virtual`**, `=0`
- Java: **`final`**, **`protected`**, **`abstract`**

U usporedbi sa strategijom: različite **mogućnosti**, veća međuovisnost:

- na raspolaganju su nam zaštićene metode osnovnog razreda
- možemo nadomještati metode osnovnog razreda

Okvirna metoda: Primjer, Python 3.x (1)

Ubacivanje klijentskog koda u biblioteku Pythona:

```
import http.server

# razred s okvirnom metodom: http.server.BaseHTTPRequestHandler
# konkretni razred: MyHandler
class MyHandler(http.server.BaseHTTPRequestHandler):
  def do_GET(self):
    myhtml="""<html><head><title>Naslov</title></head><body>
      <p>Pristup: {}</p></body></html>""".format(self.path)
    self.send_response(200)
    self.send_header("Content-type", "text/html; charset=UTF-8")
    self.end_headers()
    self.wfile.write(bytes(myhtml,'UTF-8'))
    self.wfile.close()

# pažnja: prenosimo *razred* MyHandler kao argument!
# (server stvara novi objekt za svaki konkurentni zahtjev)
server = http.server.HTTPServer(('localhost', 8000), MyHandler)
server.serve_forever()
```

Okvirna metoda: Primjer, Python 3.x (2)

Kad preglednik uputimo na `http://localhost:8000/proba` okvirna metoda `handle_one_request` poziva `do_GET` pa dobivamo:

```
<html><head><title>Naslov</title></head><body>
    <p>Pristup: /proba</p></body></html>
```

Osnovni razred na konzoli ispisuje dijagnostiku:

```
$ python3 ../code/python/myhttp.py
localhost - - [17/May/2013 13:54:46] "GET /proba HTTP/1.1" 200 -
```

Zgodan način za debuggiranje produkcijskog sustava

Okvirna metoda: primjene

Vrlo raširena u praksi, najjednostavniji OO recept za višestruko korištenje programskog kôda odnosno izbjegavanje ponavljanja

Npr. biblioteka Pythona: `cmd`, `http.server.BaseHTTPRequestHandler`

Npr. biblioteka Jave: `OutputStream`, `InputStream`

- `OutputStream.write(byte[])` za svaki bajt teksta zove `OutputStream.write(int)`
- konkretni razredi izvode `OutputStream.write(int)`

Okvirna metoda: srodni obrasci

Strategija kao složenije ali i općenitije rješenje:

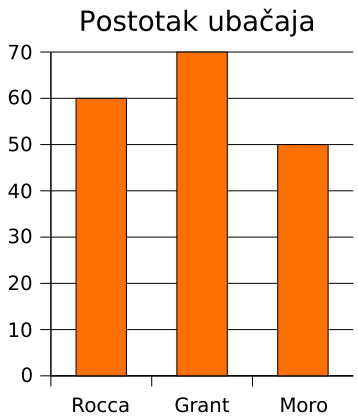
- prednosti delegacije umjesto nasljeđivanja:
  - mogućnost više nezavisnih osi promjene,
  - mogućnost dinamičke konfiguracije
- okvirna metoda ostvaruje jače povezivanje sa zajedničkim kôdom:
  - jednostavno pozivanje metoda osnovnog razreda
    - ekvivalentno implicitnom prijenosu u strategiji (pull)
  - mogućnost nadomještanja privatnih metoda osnovnog postupka
- okvirna metoda može biti jedan od postupaka u apstraktnoj strategiji

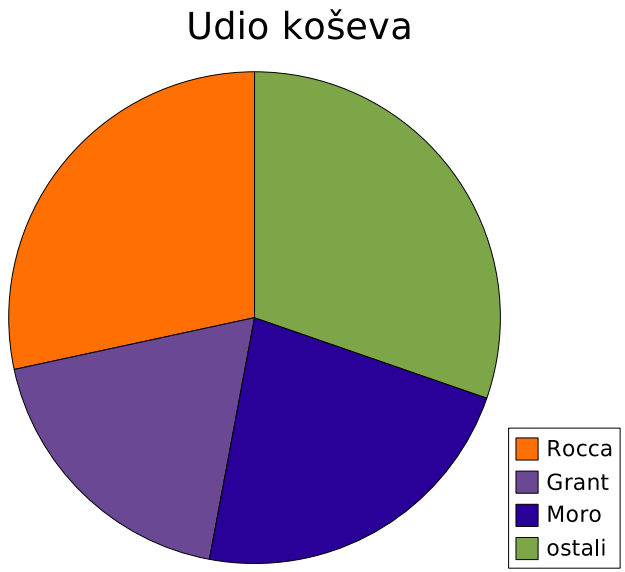
Pri kraju semestra ćemo naučiti da se metode tvornice najčešće pozivaju iz okvirnih metoda

Promatrač: namjera

Među objektima ostvariti ovisnost **1:n**: ovisni objekti poduzimaju akcije kad god glavni objekt promijeni stanje

	Rocca	Grant	Moro
šut [%]	**60**	70	50
koševi	**15**	10	12
skok	8	6	7
otete lopte	7	6	8





Promatrač: motivacija

Kako problem riješiti na krivi način:

```
void LiveReport::update(){
  // ...
  barChartDisplay.update(roccaPercent, grantPercent, moroPercent);
  pieChartDisplay.update(roccaPoints, grantPoints, moroPoints);
  matrixDisplay.update(double* data);
}
```

Što ako bude potrebno prikazivati različite kombinacije prikaza?
Što ako je prikaze potrebno dinamički stvarati?
Što ako je dinamika razvoja prikaza dominantna?

Rješenje: obrazac promatrač (observer, izdavač-pretplatnik)

- glavni objekt (subjekt, izdavač) predstavlja izvor informacija
- ovisni objekti (promatrači) obrađuju informacije
  - promatrači se pretplaćuju kod subjekta
  - subjekt ne zna za konkretni tip promatrača
- subjekt obavještava sve pretplatnike o promjenama;
- promatrači prekidaju pretplatu kad više ne trebaju obavještavanje

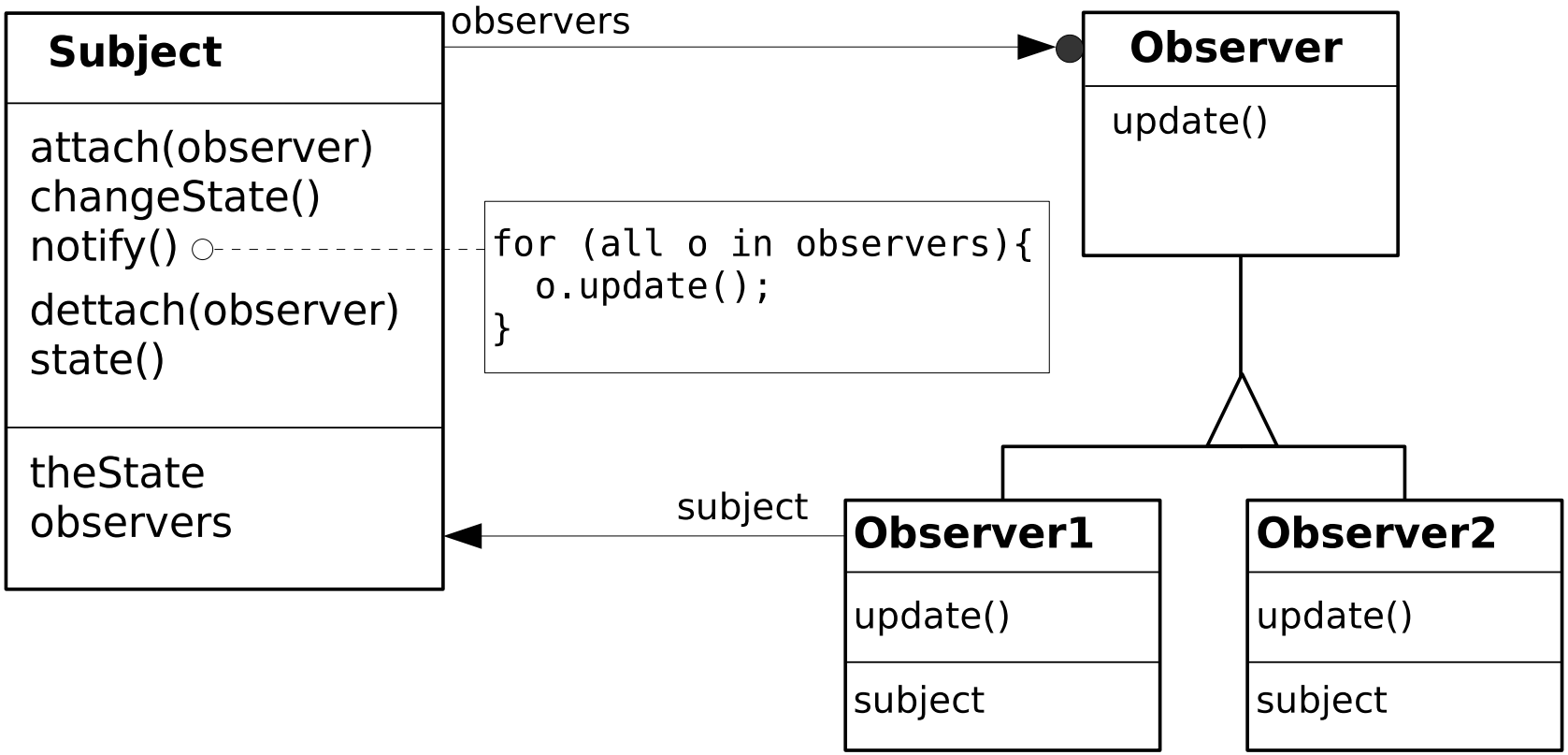
Promatrač: primjenljivost

- kad promjena jednog objekta zahtijeva promjene u drugim objektima koji nisu unaprijed poznati
- kad objekt treba obavještavati druge objekte, uz uvjet da bude što neovisniji o njima
- kad je potrebno prikazivati različite aspekte nekog dinamičkog stanja, a želimo izbjeći cikličku ovisnost između stanja i pogleda

Promatrač: strukturni dijagram

Promatrač: **ponašajni** obrazac u domeni **objekata**

Fleksibilnost se ostvaruje kombinacijom povjeravanja i nasljeđivanja



Promatrač: sudionici

- **Subjekt** (izdavač, izvor informacija)
  - poznaje promatrače preko apstraktnog sučelja
  - pruža sučelje za prijavu novih i odjavu postojećih promatrača
  - sadrži original stanja koje je od interesa promatračima (pruža sučelje za pristup tom stanju)
  - obavještava promatrače o promjenama stanja
- **Promatrač** (pretplatnik, element prikaza)
  - pruža sučelje za obavještavanje o promjenama u subjektu
- **konkretan Promatrač** (tablica, histogram, grafikon)
  - sadrži referencu na subjekt
  - može imati kopiju podskupa stanja subjekta
  - implementira sučelje za obavještavanje koje tipično obrađuje podatke subjekta

Promatrač: suradnja

- Promatrači se dinamički prijavljuju i odjavljuju kod Subjekta
- Subjekt obavještava promatrače kad god se dogodi promjena stanja
- nakon primanja obavijesti, Promatrači pribavljaju novo stanje i poduzimaju odgovarajuće aktivnosti
- promjene subjekta mogu biti inicirane i od strane Promatrača
  - Promatrač koji je inicirao promjenu obnavlja stanje tek nakon službene obavijesti
  - u suprotnom, postoji opasnost da se ista promjena evidentira dvaput
- obavještavanje najčešće inicira Subjekt, ali to mogu provesti i njegovi klijenti

Promatrač: posljedice

Ukidanje ovisnosti između Subjekta i konkretnih Promatrača

- Subjekt ne ovisi o složenoj implementaciji Promatrača
- promjene u promatračima ne utječu na subjekt

Mogućnost dinamičke prijave i odjave Promatrača

- promatrači se često prijavljuju iz konstruktora, a odjavljuju iz destruktora

Potreba za složenijim protokolom obavještavanja

- sve promjene se ne tiču svih promatrača

Usklađenost s oblikovnim načelima:

- **nadogradivost bez promjene**: subjekt nesmetano radi s novim promatračima
- **inverzija ovisnosti**: subjekt ovisi o apstraktnom sučelju
- **ortogonalnost**: subjekt je odgovoran za stanje, a promatrač za prikaz stanja

Promatrač: implementacijska pitanja

Odgovornost za obavještavanje odnosno obnavljanje pogleda (tko poziva `notify`?):

- subjekt automatski šalje obavijesti nakon svake promjene stanja (redundantne obavijesti)
- klijenti Subjekta su odgovorni za obavještavanje, nakon unosa niza promjena (mogućnost grešaka)

Protokol obavještavanja može biti detaljniji ili siromašniji (međuovisnosti Promatrača i Subjekta vs. nepotrebne obavijesti)

- promatrači mogu definirati svoje interese u trenutku prijave:

```
void Subject::attach(Observer*, Aspect& interest)
```

Kao i kod Strategije, **prijenos parametara** može biti:

- neizravan (*pull*), promatrači imaju referencu na subjekt
- izravan (*push*), stanje se prenosi preko argumenata metode `update`

Promatrač: implementacijska pitanja (2)

Metoda `update` može biti okvirna metoda:

- `http.server.BaseHTTPRequestHandler`: `handle_one_request`

Standardni promatrači rade sinkrono sa subjektom:

- nedostatak: mogućnost "izgladnjivanja" zbog rijetkih ažuriranja
- tom problemu možemo stati na kraj asinkronim promatračem

Asinkroni promatrači izvode se u vlastitoj dretvi:

- potrebno je osigurati isključivi pristup stanju (eng. mutex)
- radno čekanje na dojave možemo izbjeći blokirajućim čekanjem uvjetne varijable (eng. condition variable)

Promatrači mogu biti i u odvojenom adresnom prostoru:

- blokiranje možemo postići selektiranjem (eng. select)
- selektiranje čeka dojave operacijskog sustava ili mrežne zahtjeve preko TCP ili UDP protokola (eng. socket)

Promatrač: Izvorni kod, C++

Tipična sučelja apstraktnog pogleda i subjekta:

```
class Observer{
public:
  virtual void update() =0;
  virtual ~Observer(){}
};

class Subject{
  std::mutex mtx_;
  std::set<Observer*> sobs_;
  unsigned int state_;
public:
  Subject();
  std::mutex& mutex();
  unsigned int state();
  void attach(Observer* p);
  void detach(Observer* p);
  void change_state();
  void make_change();
  void notify();
};
```

Promatrač: Izvorni kod, C++ (2)

Elementi izvedbe subjekta:

```
void Subject::attach(Observer* p){
  sobs_.insert(p);
}
void Subject::detach(Observer* p){
  sobs_.erase(p);
}

void Subject::change_state(){
  make_change();
  notify();
}
void Subject::make_change(){
  std::lock_guard lock(mtx_);            // no partial readouts
  state_ = state_ * 48271 % 0x7fffffff; // Park-Miller RNG
}
void Subject::notify(){
  for (auto po: sobs_){
    po->update();
  }
}
```

Promatrač: Izvorni kod, C++ (3)

Konkretni sinkroni promatrač:

```
class SingleThreadObserver:
  public Observer
{
  Subject *psub_;
public:
  SingleThreadObserver(Subject* psub):
    psub_(psub)
  {
    psub_->attach(this);
  }

  virtual void update(){
    std::lock_guard lock(psub_->mutex());
    std::cout <<"[STO] novo stanje: "
      <<psub_->state() <<std::endl;
  }
};
```

Promatrač: Izvorni kod, C++ (4)

Razred konkurentnih promatrača:

```
class MultiThreadObserver:
  public Observer
{
  Subject *psub_;
public:
  std::thread thr_;
  bool stop_;
  std::condition_variable cv_;
  int state_;
public:
  MultiThreadObserver(Subject* ps);
  ~MultiThreadObserver();
  virtual void update();
private:
  void worker();
};
```

Promatrač: Izvorni kod, C++ (5)

Sinkrone metode konkurentnog promatrača:

```
MultiThreadObserver::MultiThreadObserver(Subject* ps):
  psub_(ps), stop_(false),
  thr_(&MultiThreadObserver::worker, std::ref(*this)),
{
  psub_->attach(this);
}

MultiThreadObserver::~MultiThreadObserver(){
   stop_=true;
   thr_.join();
}

void MultiThreadObserver::update(){
  cv_.notify_all();
}
```

Promatrač: Izvorni kod, C++ (6)

Asinkrona metoda konkurentnog promatrača:

```
void MultiThreadObserver::worker(){
  std::unique_lock lock(psub_->mutex());
  while (!stop_){
    // wait for notification or timeout
    std::cv_status rv=
      cv_.wait_for(lock, std::chrono::seconds(1));

    // print outcome
    if (state_ == psub_->state()){
      std::cout <<"[MTO] vrijeme isteklo: ";
    } else{
      std::cout <<"[MTO] novo stanje: ";
    }
    std::cout <<psub_->state() <<std::endl;
    state_ = psub_->state();
  }
}
```

Promatrač: Izvorni kod, C++ (7)

Glavni program:

```
int main(){
  Subject subject;
  SingleThreadObserver obs(&subject);
  MultiThreadObserver obs2(&subject);
  MultiThreadObserver obs3(&subject);
  while (true){
    // stochastic delay
    int delay = 1 + 2*(subject.state()/2%2);
    std::this_thread::sleep_for(
      std::chrono::seconds(delay));

    subject.change_state();
  }
}
```

Promatrač: primjer, Python

Biblioteka `watchdog` omogućava praćenje izmjena datotečnog sustava

- `Observer`: promatrač koji prima dojave i prosljeđuje ih klijentima
- `FileSystemEventHandler`: osnovni razred s nadomjestivim metodama

```
import watchdog, watchdog.events, watchdog.observers

class MyHandler(watchdog.events.FileSystemEventHandler):
  def on_any_event(self, e):
    print('Event: {} [{}]'.format(e.src_path, e.event_type))

myobs = watchdog.observers.Observer()
myobs.schedule(event_handler=MyHandler(), path='./')
myobs.start()
try:
  while True:
    time.sleep(1)
except KeyboardInterrupt:
  print('Stopped.')
  myobs.stop();
  myobs.join()
```

Promatrač: primjer, Python (2)

Pretpostavimo da smo program pokrenuli s:

```
$ python observer.py
```

Pretpostavimo da smo nakon toga izmijenili datotečni sustav:

```
$ touch a
$ rm a
```

Naš promatrač na konzoli ispisuje dijagnostiku:

```
Event: ./a [created]
Event: . [modified]
Event: ./a [modified]
Event: ./a [deleted]
Event: . [modified]
^CStopped.
```

Promatrač: primjeri upotrebe

Elementi GUI biblioteka često se ponašaju kao Subjekti, dok se klijentski kod prijavljuje kao Promatrač (npr. Java listeners)

U C-u, poziv promatrača se izvodi tzv. call-back funkcijama čiji prvi argument obično identificira subjekt.

Razred `http.server.HTTPServer` možemo promatrati kao blokirajući subjekt

- razred `http.server.BaseHTTPRequestHandler` je onda sinkroni promatrač.

Razred `watchdog.observers.Observer` je promatrač u odvojenom adresnom prostoru

- na Linuxu, blokirajući subjekt selektira jezgreni modul `inotify`

Promatrač: srodni obrasci

Promatrač je sličan strategiji (delegacija), ali:

- subjekt ima **polje** pokazivača na "strategije"
- specijalizirana primjena: "strategije" osluškuju stanje subjekta

Bibliotečni promatrač može povjeravati strategijama ili okvirnoj metodi s klijentskim primitivima:

- `http.server.BaseHTTPRequestHandler` $\longrightarrow$ `MyHandler.do_GET`
- `watchdog.observers.Observer` $\longrightarrow$ `watchdog.events.FileSystemEventHandler.dispatch` $\longrightarrow$ `MyHandler.on_any_event`

Promatrač je sastavni dio arhitektonskog obrasca model - pogled - upravljač (MVC) kojeg ćemo obraditi na kraju semestra

Dekorator: problem

Namjera:

- dinamičko dodavanje i povlačenje odgovornosti objektima
- različite dodane odgovornosti mogu se po volji komponirati s različitim oblicima osnovne funkcionalnosti

Primjer: implementiranje vlastitog komunikacijskog protokola

```
void MyRequest::send(OutputStream& os){
  os.write(ourHeader);
  os.write(ourRequest);
  os.write(ourData);
  os.flush();
}
```

Zahtjev 1: konkretne izvedbe `OutputStream`a podržavaju prijenos preko mreže, cjevovoda, datoteka, konzole (**zvuči poznato**?)

Zahtjev 2: konkretnim `OutputStream`ovima potrebno dodavati odgovornosti: bufferiranje, sažimanje, kriptiranje, digitalni potpis

Dekorator: motivacija

Zahtjev 1 rješavamo strategijom uz konkretne razrede:

`PipedOutputStream`, `SocketOutputStream`, `FileOutputStream`, `ConsoleOutputStream`

Zahtjev 2, ideja A: odgovornosti dodavati nasljeđivanjem

- svaku kombinaciju temeljne funkcionalnosti i dodatnih odgovornosti modelirati zasebnim konkretnim razredom

```
class BufferedPipedOutputStream: public OutputStream { /* ...*/};

class BufferedSocketOutputStream: public OutputStream { /* ...*/};

...

class SignedFileOutputStream: public OutputStream { /* ...*/};

...
...
class BufferedCompressedEncryptedSignedSocketOutputStream
 : public OutputStream { /* ...*/};
...
...
```

Dekorator: motivacija (2)

Nedostatci dodavanja odgovornosti nasljeđivanjem

1. Kombinatorna eksplozija broja potrebnih razreda:
   - $4 \cdot 2$ = 8 kombinacija s jednom dodatnom odgovornošću
   - $4 \cdot 2^4$ = 64 kombinacija sveukupno
2. Kršenje načela jedinstvene odgovornosti
   - ponavljajuće implementacije dodatnih odgovornosti u slučaju datoteke, cjevovoda itd.
3. Nemogućnost dinamičkog konfiguriranja odgovornosti
   - što ako npr. želimo kriptirati samo dio poruke?

Dekorator: motivacija (3)

Zahtjev 2, ideja B: sve dodatne odgovornosti smjestiti u razred kojeg izvode razredi s temeljnim funkcionalnostima.

```
class OutputStream{
  // ...
public:
  void configureBuffering(int bufsize);
  void configureCompression(int blocksize);
  void configureEncription(string key);
  void configureSignature(string key);
}

class FileOutputStream: public OutputStream{...}
...
```

Nedostatci smještanja dodatnih odgovornosti u osnovni razred:

- otežano dodavanje novih odgovornosti (npr. logiranja): krši se NBP
- pretrpan osnovni razred: krši se NJO

Ni ovo rješenje nas ne veseli...

Dekorator: motivacija (4)

Tražimo rješenje sa sljedećim svojstvima:

- dodatne odgovornosti definirane u zasebnim komponentama
  - NJO, NBP
- dodatne odgovornosti primjenjive na sve temeljne funkcionalnosti
  - modeliranje nezavisnih osi varijacije programa
  - osnovne odgovornosti čine os 1 (4 kombinacije):
    1. `Piped`, `Socket`, `File`, `Console`
  - dodane odgovornosti čine osi od 2 na dalje (2^4 kombinacija):
    2. `Buffered`
    3. `Signed`
    4. `Compressed`
    5. `Encrypted`
  - NJO, NBP
- mogućnost **dinamičke** konfiguracije dodatnih odgovornosti
- klijenti ne moraju znati ni za temeljne funkcionalnosti ni za dodatne odgovornosti (NIO)

Dekorator: motivacija (5)

Rješenje: razredi s dodatnim odgovornostima umataju objekt na kojeg djeluju, bez znanja o njegovom konkretnom tipu.

```
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 );
 }
```

Rezultat:

- dodatne odgovornosti u odvojenim komponentama ✓
- slobodno kombiniranje s osnovnim odgovornostima ✓
- dinamičko konfiguriranje dodatnih odgovornosti ✓
- klijent može znati samo za osnovno sučelje ✓

Dekorator: rješenje

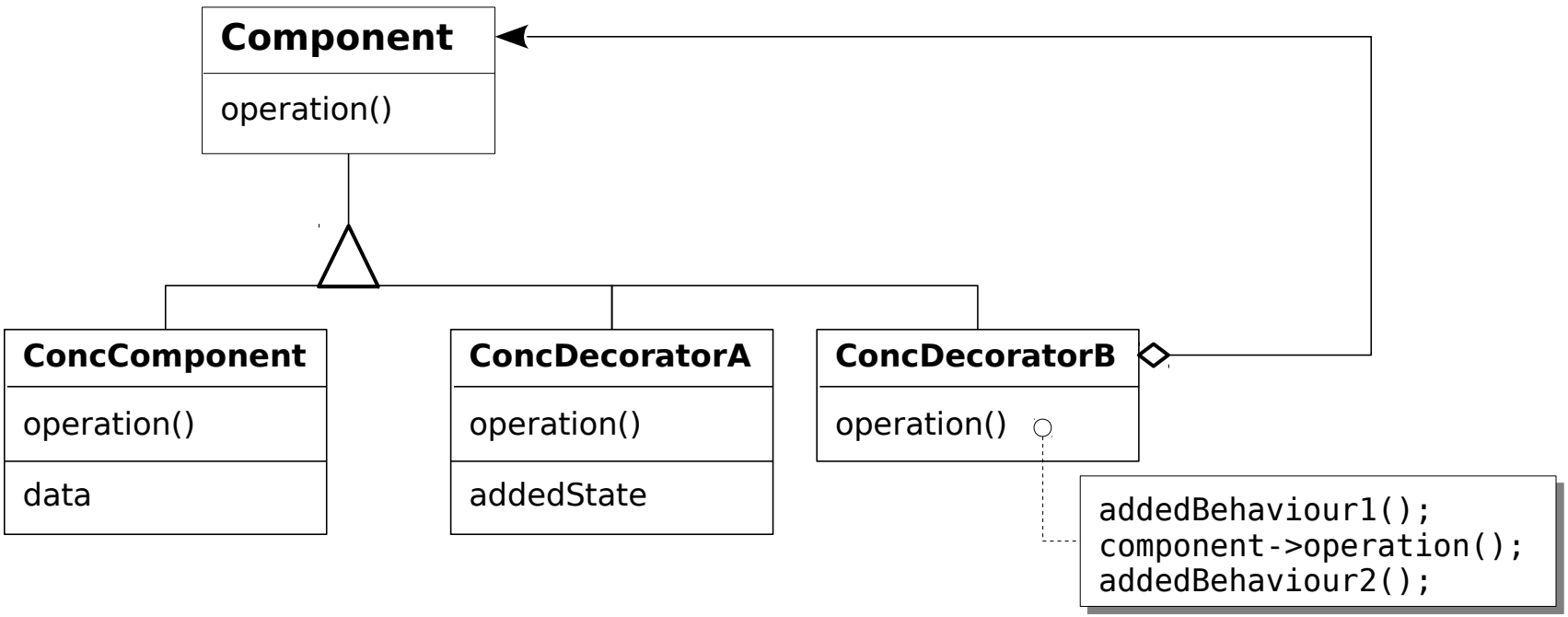
Obrazac dekorator (decorator, wrapper)

- dodjela nezavisnih odgovornosti pojedinim objektima
  (ne cijelim razredima!)
- temeljna ideja -- rekurzivna kompozicija:
  podatkovni član K ima isti osnovni razred kao i matični objekt D
- dekorator D implementira dodatnu odgovornost, a ostale
  odgovornosti prosljeđuje umetnutoj (omotanoj) komponenti K
- mogućnost **rekurzivnog** umetanja
  (pospješuje se ortogonalnost)

Dekorator: strukturni dijagram

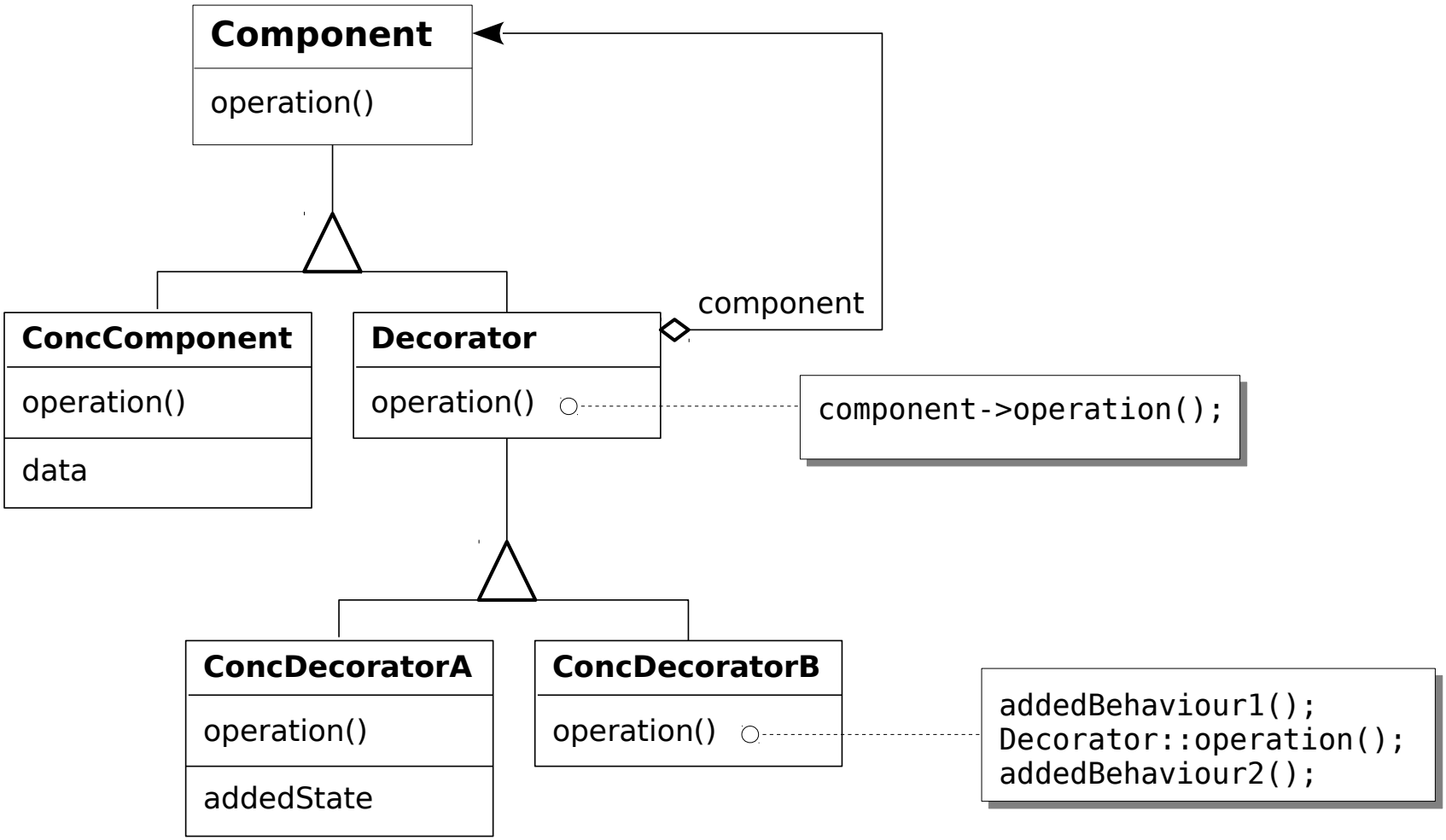
Dekorator: **strukturni** obrazac u domeni **objekata**

Fleksibilnost se ostvaruje rekurzivnom kompozicijom



Dekorator: strukturni dijagram (2)

Kako bismo smanjili ponavljanje, zajedničke operacije svih dekoratora možemo izdvojiti u zajednički osnovni razred:



Dekorator: primjenljivost

- objektima potrebno dodati odgovornosti **dinamički** i **transparentno**
- odgovornosti je potrebno moći **povući**
- nasljeđivanje konkretne komponente nepraktično:
  - želimo dinamičku konfiguraciju
  - nemamo (ne želimo) višestruko nasljeđivanje
- dodatke nije prikladno implementirati unutar osnovnog razreda jer:
  - preglomazni su i često se mijenjaju
    (kršenje načela jedinstvene odgovornosti)
  - neki dekoratori mogu ne biti primjenljivi na sve objekte
    (npr. scrollbar dekorator i nepravokutni prozor)

Dekorator: sudionici

- **Komponenta** (`java.io.OutputStream`)
  - definira sučelje za objekte kojima se odgovornosti mogu dinamički konfigurirati
- **Konkretna komponenta** (`java.io.FileOutputStream`)
  - definira izvedbu objekata s dinamičkim odgovornostima
- **Dekorator** (`java.io.FilterOutputStream`)
  - nasljeđuje sučelje Komponente
  - sadrži referencu na umetnutu komponentu
  - može pružiti implementacije metoda komponente koje posao delegiraju umetnutoj komponenti
- **Konkretni dekorator** (`java.io.BufferedOutputStream`)
  - izvodi dodatne odgovornosti komponente

Dekorator: suradnja

Tipično, Dekoratori prosljeđuju zahtjeve sučelja Komponente sadržanim Komponentama.

Prije i (ili) poslije prosljeđenog poziva, dekoratori obavljaju dodatne operacije za koje su odgovorni

Klijenti koji trebaju konfigurabilnost po volji dekoriraju konkretnu komponentu dodatnim odgovornostima

Klijenti koji ne trebaju konfigurabilnost transparentno koriste dekoriranu komponentu preko apstraktnog sučelja

Dekorator: posljedice

- u odnosu na dodatne odgovornosti u naslijeđenim razredima: **dinamička ortogonalna** konfiguracija
- u odnosu na konfigurabilnu konkretnu komponentu:
  - aplikacija ne mora održavati svojstva koja se ne koriste (NJO)
- oprez: dekorirana komponenta nije identična originalu!
- povećana složenost, puno malih objekata

Dekorator: implementacijska pitanja

U strogo tipiziranim jezicima Dekorator mora naslijediti Komponentu (ne možemo isti dekorator primijeniti na objekte koji nisu u rodu)

Samo jedan konkretni Dekorator $\Rightarrow$ apstraktno sučelje izostavljamo

Svaki Konkretni Dekorator je istovremeno i Komponenta:
$\Rightarrow$ dekoriranje implicira duplikaciju podatkovnih članova Komponente
$\Rightarrow$ Komponenta treba biti što apstraktnija

Dekorator vs Strategija:

- mijenjanje vanjskog izgleda vs. mijenjanje nutrine objekta
- ``teže'' Komponente (memorijski, izvedbeno) $\Rightarrow$ Strategija prikladnija
- kod Strategije, ekvivalent osnovne komponente dobiva se konfiguriranjem nul-objektima (složenije nego kod Dekoratora)

Dekorator: izvorni kôd (java)

```
public class Gunzip{
  public static void main(String [] args)
    throws IOException
  {
    FileInputStream fis = new FileInputStream(args[0]);
    BufferedInputStream bfis = new BufferedInputStream(fis);
    GZIPInputStream gbfis = new GZIPInputStream(bfis);

    FileOutputStream fos = new FileOutputStream(args[1]);
    BufferedOutputStream bfos = new BufferedOutputStream(fos);

    copy(gbfis, bfos);
    bfos.close();
  }

  public static void copy(InputStream is, OutputStream os)
    throws IOException
  {
    byte [] b = new byte [1024];
    while (true){
      int n=is.read(b);
      if (n<0) break;
      os.write(b,0,n);
    }
  }
}
```

Dekorator: izvorni kôd (2) (java)

```
JTextArea ta = new JTextArea("Ovo je jedan jako dugačak redak"+
  " koji se neće dobro prikazati bez mogućnosti skroliranja.");

JFrame frame = new JFrame();
frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
frame.setSize(200, 200);
frame.getContentPane().add(ta);     // JTextArea isa JComponent
frame.setVisible(true);
System.out.print("Prikaz teksta bez skroliranja (enter za dalje)");
String input = System.console().readLine();

JScrollPane jsp = new JScrollPane(ta);
frame.getContentPane().remove(ta);
frame.getContentPane().add(jsp);    // JScrollPane isa JComponent
frame.getContentPane().validate();
System.out.print("Dodana je traka za skroliranje (enter za dalje)");
input = System.console().readLine();

frame.getContentPane().remove(jsp);
frame.getContentPane().add(ta);
frame.getContentPane().validate();
System.out.print("Ponovo spartanski izgled (enter za kraj)");
input = System.console().readLine();

frame.dispatchEvent(new WindowEvent(frame, WindowEvent.WINDOW_CLOSING));
```

Dekorator: izvorni kôd (3) (Python)

Zbog implicitnog tipiziranja Dekorator ne mora naslijediti komponentu:

```
class ConcreteComponent:
  def msg(self):
    return "Hello world!"
class DecoratorBold:
  def __init__(self, c):
    self.c_=c
  def msg(self):
    return "<b>"+self.c_.msg()+"</b>"
```

```
def client(c):
  print(c.msg())

c=ConcreteComponent()
client(c) # Hello world!

dc=DecoratorBold(c)
client(dc) # <b>Hello world!</b>
```

U Pythonu su funkcije **punopravni objekti** pa ih možemo dekorirati:

```
def hello():
  return "Hello world!"
def makebold(fn):
  def wrapper():
    return "<b>" + fn() + "</b>"
  return wrapper
```

```
>>> hello()
Hello world!

>>> hellobold=makebold(hello)
>>> hellobold()
<b>Hello world!</b>
```

Dekorator: izvorni kôd (4) (Python)

Posebna sintaksa omogućava definiranje dekoriranih funkcija:

```
@makebold
def hellobold2():
    return "Hello world!"
# as if: hellobold2=makebold(hellobold2)

>>> hellobold2() # prints <b>Hello world!</b>
# http://wiki.python.org/moin/PythonDecoratorLibrary
```

Dekorator: izvorni kôd (5) (Python)

Neki jezici omogućavaju zadavanje atributa koji se koriste poput podatkovnih članova, dok njihovo prozivanje rezultira pozivom metoda

- motivacija: NBP klijenata razreda s javnim podatkovnim članovima
- ekapsulacija bez sveprisutnih gettera i settera!

Takve atribute nazivamo svojstvima (properties):

```
print(x.prop) # same as x.get_prop()
x.prop = 42   # same as x.set_prop(42)
x.prop += 5   # same as x.set_prop(x.get_prop()+5)
```

U Pythonu svojstva gradimo umatanjem metode dekoratorom `property`:

```
class Magic:

  @property
  def limit3(self):
    return self.x

  @limit3.setter
  def limit3(self, newx):
    self.x=min(newx,3)
```

```
>>> x=Magic()
>>> x.limit3 = 1
>>> x.limit3 += 1 # jasnije nego ...
>>> x.limit3 += 1 # x.set_limit3( ...
>>> x.limit3
3
>>> x.limit3 += 1
>>> x.limit3
3
```

Dekorator: izvorni kôd (6) (Python)

Python podržava funkcije s promjenljivim brojem argumenata

- ⇒ možemo oblikovati dekoratore za umatanje proizvoljnih funkcija

Takvim dekoratorima možemo preusmjeravati i ugrađene funkcije:

```
>>> import builtins
>>> builtins.sum=uključiBilježenje(sum)
>>> L=[1,4,3,2]
>>> x=sum(L)
Poziv funkcije sum:
  Pozicijski argumenti: ([1, 4, 3, 2],)
  Imenovani argumenti: {}
  Povratna vrijednost: 10
```

```
def uključiBilježenje(fun):
  def bilježi(*pozarg, **imarg):
    print("Poziv funkcije "+ fun.__name__+":",
     "\n  Pozicijski argumenti:", pozarg,
     "\n  Imenovani argumenti:", imarg)
    rv= fun(*pozarg, **imarg)
    print("  Povratna vrijednost:", rv)
    return rv
  return bilježi
```

Dekorator: primjeri upotrebe

- transparentno dodavanje grafičkih efekata u bibliotekama GUI elemenata
- dinamičko konfiguriranje dodatnih funkcionalnosti tokova podataka (Java streams)
- konfiguriranje pristupa bazi podataka:
  - sa ili bez logiranja
  - signalizacija pogrešaka povratnim kôdovima ili iznimkama
- Python:
  - svojstva: podatkovni članovi koji su zapravo metode
  - modificirati proizvoljnu funkciju vanjske biblioteke
  - debuggirati/logirati/profilirati više funkcija, i to bez ponovljenog kôda

Naredba: namjera, primjer, problem

Omogućiti unificirano baratanje raznorodnim zahtjevima

Motivacijski primjer: fleksibilno rukovanje izbornicima korisničkog sučelja

Primjer korištenja: korisnik odabire izbornik View->Zoom->120%

- bez smanjenja općenitosti pretpostavljamo da taj događaj obrađuje metoda `MenuItem::clicked()`

Kad bi `MenuItem::clicked()` izravno pozivala `pView->setZoom()`:

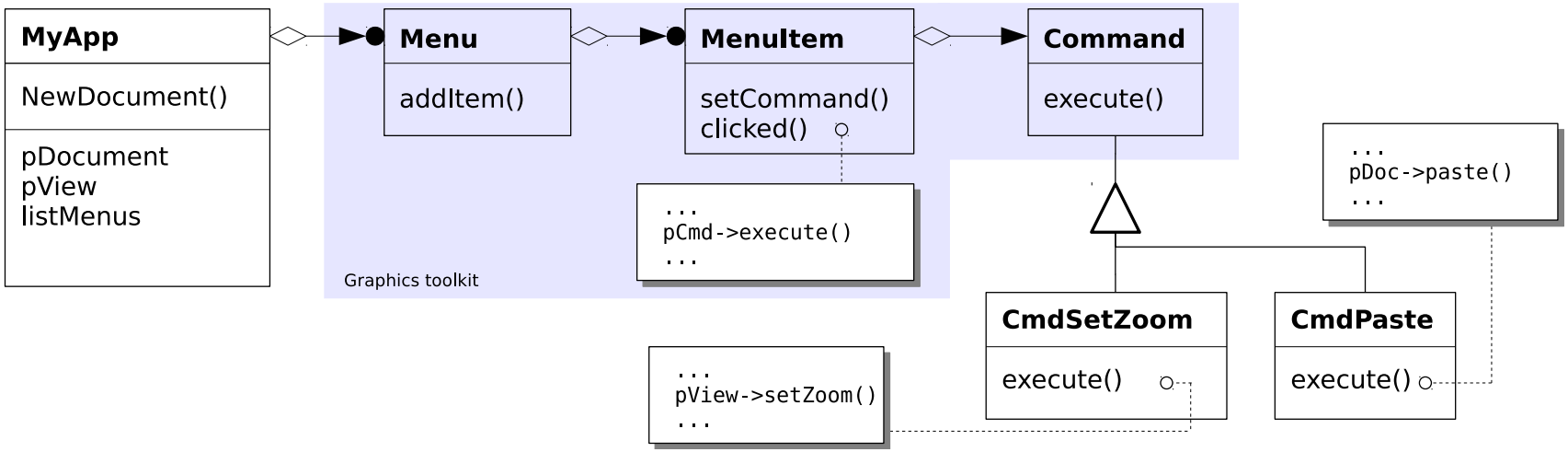
- otežano ponovno korištenje menija čija izvedba ne bi mogla biti zatvorena u biblioteci
- svaka aplikacija morala bi pisati svoje menije

Kad bi `MenuItem::clicked()` pozivao `Application::event` (win32 API), implementacija bi:

- imala puno `if`ova
- ne bi bila nadogradiva bez promjene

Naredba: rješenje

Ideja: `MenuItem` ima podatkovni član `pCmd` kojeg postavlja metoda `setCommand`, a `MenuItem::clicked()` poziva `pCmd->execute()`



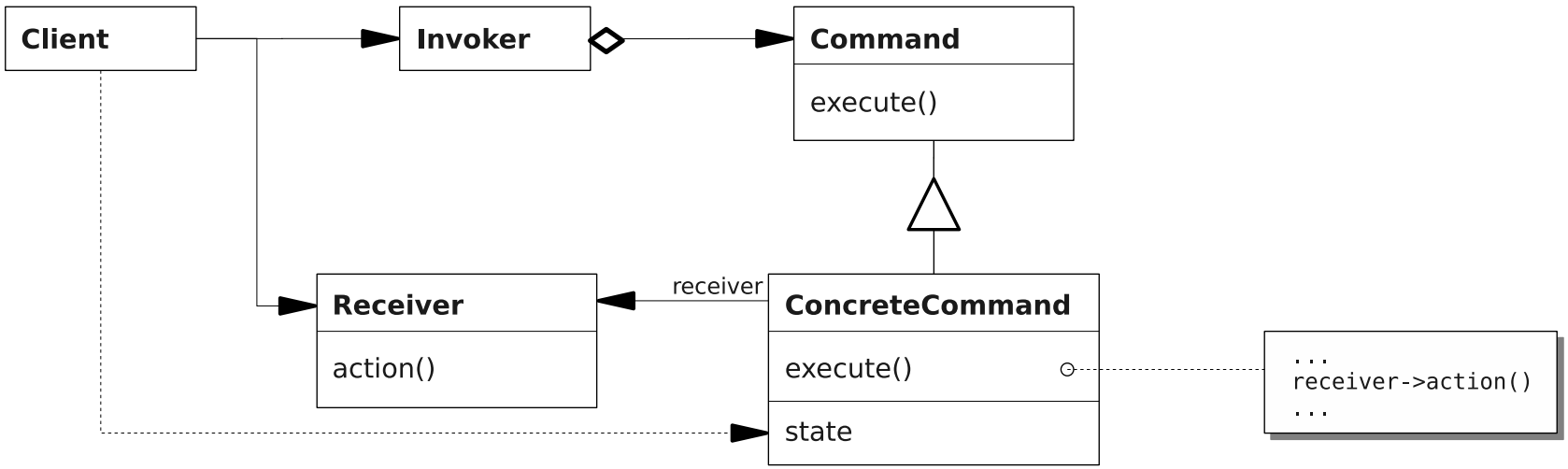
Rezultat: upravljač menijima je NBP pa ga stavljamo u biblioteku!

Naredba razdvaja zaprimanje, pozivanje i primanje zatjeva

- grafički podsustav poziva korisnički kod bez da o njemu ovisi
- rukovanje poslovima bez ovisnosti o konkretnim operacijama

Naredba: strukturni dijagram

Naredba: **ponašajni** obrazac u domeni **objekata** (akcija, transakcija)



Fleksibilnost se ostvaruje kombinacijom povjeravanja i nasljeđivanja

Naredba: primjenljivost

Promatranu komponentu potrebno je parametrizirati s nepoznatom akcijom koju komponenta treba pozvati

- odgovornost komponente je ortogonalna s obzirom na akciju
  - odgovornosti su ortogonalne ako se trebaju proizvoljno kombinirati
  - npr. grafička biblioteka bavi se iscrtavanjem, praćenjem miša, onemogućavanjem izbornika itd.
  - npr. akcije mogu konfigurirati pogled, lijepiti tekst, snimati na disk itd.

Zadavanje i izvođenje zadataka se zbivaju u različitim trenutcima

Potrebno podržati operaciju undo ili grupiranje operacija u makroe

Potrebno podržati složene atomarne operacije (transakcije)

Naredba: sudionici

Naredba (Command):

- deklarira jednostavno sučelje za izvođenje operacije

Konkretna naredba (CmdPaste):

- definira vezu između Pozivatelja i Primatelja
- implementira sučelje Naredbe pozivajući Primatelja

Pozivatelj (MenuItem):

- inicira zahtjev preko sučelja Naredbe

Primatelj (MyDocument):

- zna koje operacije mora obaviti kako bi se zadovoljio zahtjev

Klijent (MyApp):

- kreira Konkretnu naredbu i predaje je Pozivatelju

Naredba: suradnja

Klijent kreira Konkretnu naredbu, navodi njenog Primatelja, te registrira naredbu kod Pozivatelja

Pozivatelj poziva konkretnu Naredbu preko osnovnog sučelja

- naredba po potrebi sprema stanje kako bi se omogućio njen opoziv (*undo*) ili ponovno izvršavanje (*redo*)

Konkretne Naredbe pozivaju metode Primatelja da zadovolje zahtjev

Pozivatelj i Primatelj se ne poznaju

Naredba: posljedice

Naredba odvaja inicijatora zahtjeva (Pozivatelja) od objekta koji zna izvršiti potrebnu operaciju (Primatelja)

Zahtjevnije Naredbe mogu se složiti od jednostavnijih, kao u slučaju Makro-naredbe. Makro-naredbe odgovaraju obrascu Kompozitu.

Dodavanje novih Naredbi je lako jer ne zahtijeva mijenjanje postojećih razreda

Ako je interesantno da Pozivatelj može inicirati više Naredbi - dobivamo strukturu sličnu obrascu Promatrač

Nedostatak: velik broj malih razreda; može se ublažiti naprednim jezičnim konstruktima

Naredba: implementacijska pitanja

Preinake za omogućavanje opozivanja naredbi (undo):

- naredbe moraju spremiti stanje potrebno za opoziv te definirati metodu `unexecute`
- potrebno je evidentirati izvršene i opozvane naredbe u posebnom objektu (npr. `UndoManager`)
  - taj objekt agregira dva reda naredbi (npr. `redo_stack` i `undo_stack`)
- metoda `UndoManager::undo` uzima naredbu sa stoga `undo_stack`, izvršava metodu `unexecute` te sprema naredbu na `redo_stack`
- pokušajte pogoditi što trebaju napraviti metode `UndoManager::redo` i `UndoManager::do`!

Preinake za stvaranje makro-naredbi:

- za to ćemo pričekati obrazac Kompozit

Jezična podrška za zadavanje razreda naredbi:

- bezimeni razredi (Java), bezimene funkcije (C++, Python), predlošci (C++)

Naredba: izvorni kôd (1) (Java)

```
// import javax,swing*, java.awt.event*
public class MyMenuLab extends JFrame {
  public MyMenuLab() {
    super("MyMenuLab");
    setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    final JTextArea myText = new JTextArea(5, 30);
    getContentPane().add(myText);
    // setup menus
    JMenuBar myMenuBar = new JMenuBar();
    JMenu myMenu = new JMenu("Menu");
    Action myAction = new AbstractAction("MyAction", null) {
      public void actionPerformed(ActionEvent e) {
        myText.append("Got action from " +  e.getActionCommand() +"\n");
      }
    };
    myMenu.add(new JMenuItem(myAction));
    myMenuBar.add(myMenu);
    setJMenuBar(myMenuBar);
  }
  public static void main(String[] args) {
    MyMenuLab frame = new MyMenuLab();
    frame.pack();
    frame.setVisible(true);
  }
}
```

Naredba: izvorni kôd (2) (Python)

```
import tkinter as tk

class MyMenuDemo:
  def __init__(self, root_wnd):
    root_wnd.title('MyGuiDemo')
    self.textarea=tk.Text(root_wnd, width=50, height=5)
    self.textarea.pack(side=tk.LEFT)

    # setup menus
    self.menubar = tk.Menu(root_wnd)
    self.menu = tk.Menu(self.menubar, tearoff=0)
    self.menu.add_command(label="MyItem", command=lambda :
       self.textarea.insert(tk.INSERT, "Command called\n"))
    self.menubar.add_cascade(label="MyMenu", menu=self.menu)
    root_wnd.config(menu=self.menubar)

root_wnd = tk.Tk()
myapp=MyMenuDemo(root_wnd)
root_wnd.mainloop()
```

Naredba: primjeri upotrebe

Podržavanje opozivih akcija, uključujući transakcijsko poslovanje

- svaka izvedena naredba pohranjuje se u stog za opoziv (undo)
- prilikom opoziva, naredba se vadi iz stoga za opoziv, poništava se njen učinak, te se premješta u stog za ponovno izvođenje (redo)

Korisnička sučelja, npr. gtkmm ili Java Swing:

- postavljanje naredbe: `JMenuItem::AddActionListener`
- apstraktna naredba: `ActionListener`
- metoda `execute`: `ActionListener::actionPerformed`

Raspoređivanje poslova u stvarnom vremenu
(npr. thread pool unutar web servera)

U dinamičkim jezicima kôd je jednostavniji: apstraktna naredba je nepotrebna, a konkretna naredba može biti i funkcija

Naredba: usporedba

Naredbu možemo promatrati kao specijalni slučaj **Strategije**.

Naredba je slična **Promatraču**, ali postoje i razlike:

- pozivatelji najčešće pozivaju samo jednu naredbu, dok subjekti mogu imati više promatrača
- naredbe u svojoj implementaciji tipično samo pozivaju primatelja, dok promatrači imaju svoju vlastitu logiku
- redoslijed pozivanja naredbi je važan dok redoslijed pozivanja promatrača nije