Oblikovni obrasci u programiranju

rješenja učestalih oblikovnih problema

Siniša Šegvić

Zavod za elektroniku, mikroelektroniku,

računalne i inteligentne sustave

Fakultet elektrotehnike i računarstva

Sveučilište u Zagrebu

Sadržaj

Oblikovni obrasci u programiranju

- Uvodni primjer: obrazac **strategije**
- **Ponašajni** obrasci: Okvirna metoda, Lanac odgovornosti, Naredba, Prevodioc, Iterator, Posrednik, Djelomično stanje, Promatrač, Stanje, Strategija, Posjetitelj.
- **Strukturni** obrasci: Prilagodnik, Most, Kompozit, Dekoracija, Pročelje, Dijeljeni objekt, Surogat.
- Obrasci **stvaranja**: Apstraktna tvornica, Metoda tvornica, Jedinstveni objekt, Prototip, Graditelj.

OBRASCI: PRIMJER

Ponovo primjer programa s nadogradivim pribavljanjem slike, obradom slike i spremanjem slike:

```
class vs_base; // video source (getFrame)
class vd_base; // video destination (putFrame)
class alg_base;// image processing algorithm (process)
...
void mainLoop(vs_base& vs, alg_base& algorithm, vd_base& vd){
  std::vector<vd_win*> pvdWins(algorithm.nDst());
  for (int i=0; i<algorithm.nDst(); ++i){
    pvdWins[i]=new vd_win;
  }
  while(!vs.eof()){
    img_data img;
    vs.getFrame(img);
    algorithm.process(img);
    for (int i=0; i<algorithm.nDst(); ++i){
      pvdWins[i]->putFrame(algorithm.imgDst(i));
    }
    vd.putFrame(algorithm.imgDst(0));
  }
}
```

Pristup ostvarivanja nadogradivosti: **povjeriti** (delegate) dio posla dediciranom objektu preko pokazivača na osnovni razred

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 elemenata koji čine obrazac
  - odnos među elementima (razdioba odgovornosti, suradnja)
- **rezultat**: što postižemo primjenom obrasca?
  - prednosti, nedostatci, kompromisi
  - utjecaj na **nadogradivost**, **proširivost**, višestruko korištenje

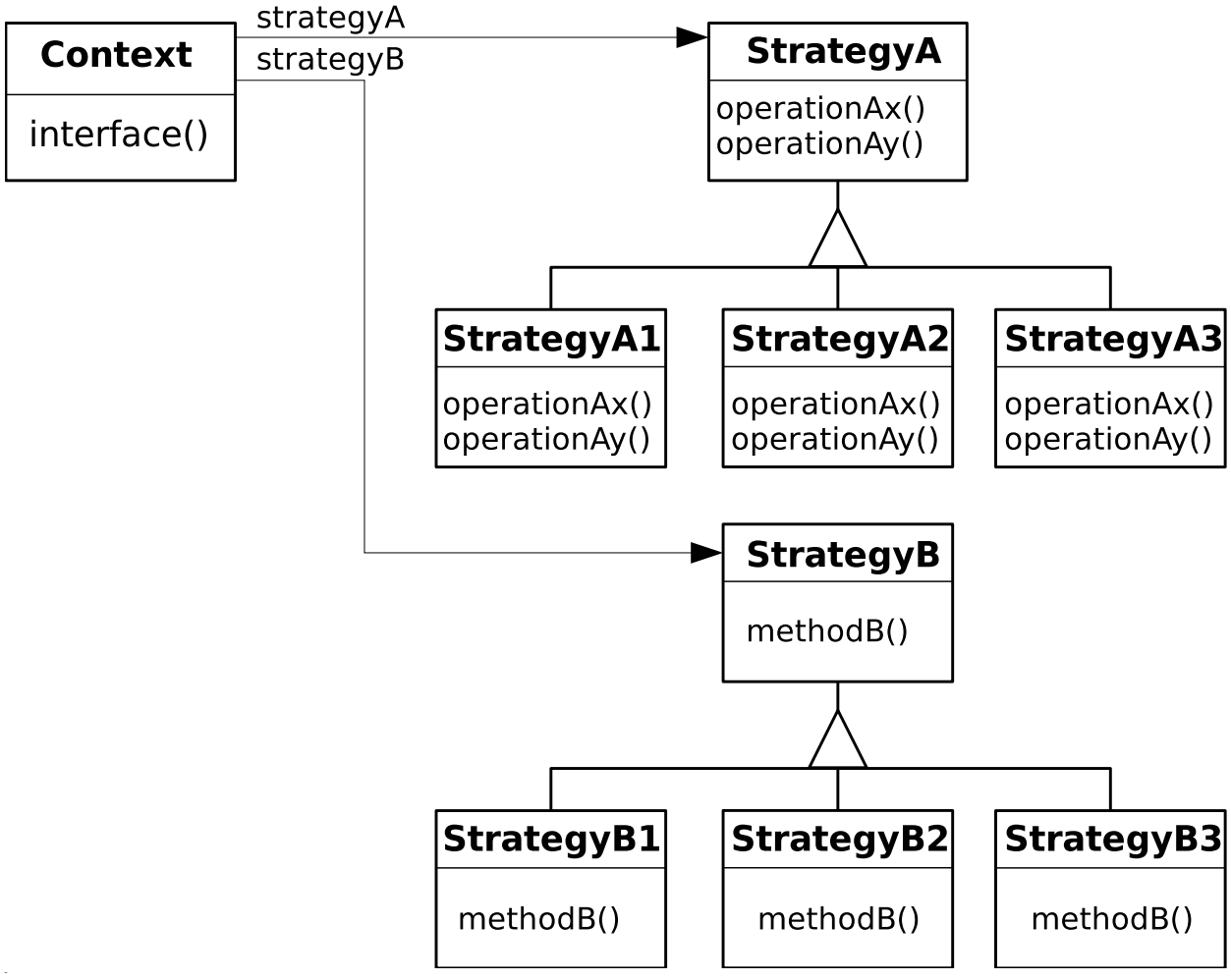
Obrasci: potpuni GoF opis

- **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: OSNOVNI OPIS

- **Problem**:
  Obrazac strategije je koristan kad je potrebno dinamički mijenjati postupke koji se primijenjuju u aplikaciji. Izmjena postupaka se odvija neovisno o kontekstu iz kojeg se koriste. Obrazac je posebno koristan ako kontekst zahtijeva više obitelji postupaka.
- **Rješenje**:
  Definirati hijerarhiju(e) postupaka, enkapsulirati konkretne postupke, omogućiti izmjenjivanje prema potrebi. Kontekst povjerava zadatke konkretnim postupcima preko sučelja osnovnog razreda.
- **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



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
  - eksplicitno slanje parametara (*push*)
  - implicitno 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 (zajednička funkcionalnost se može izdvojiti nasljeđivanjem)
- 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 (eksplicitan i implicitan prijenos parametara)

Statički ili dinamički polimorfizam?

Mogućnost specialnih strategija:

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

STRATEGIJA: IZVORNI KÔD

Priloženi kôd ilustrira mogućnosti eksplicitnog i implicitnog prijenosa parametara konteksta u sučeljnu metodu strategije:

```
//Strategy.hpp
class Context;    // Strategy depends on
class Strategy{   // Context in name only!
  virtual int pushMethod(int param) =0;
  virtual int pullMethod(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_.pushMethod(state_);
  a_.pullMethod(*this);
}
```

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

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

STRATEGIJA: PRIMJERI UPOTREBE

- **grafičke biblioteke**: različiti načini prikaza grafičkih elemenata
- **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)

STRATEGIJA: KLASIFIKACIJA

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

Fleksibilnost se ostvaruje:

- kombinacijom povjeravanja i nasljeđivanja
- generičkim programiranjem

Povjeravanje omogućava **dinamičku** konfiguraciju

Generici pružaju manju fleksibilnost (statičko povezivanje), ali uz **maksimalnu performansu** (teoretski, **cijena fleksibilnosti = 0**!)

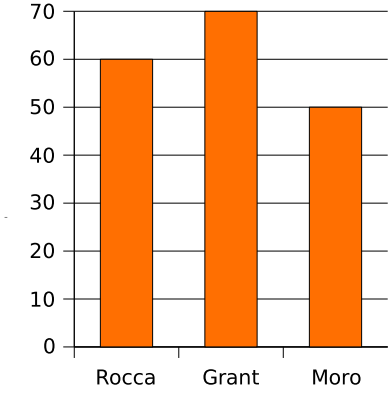
Obrasci: sažetak

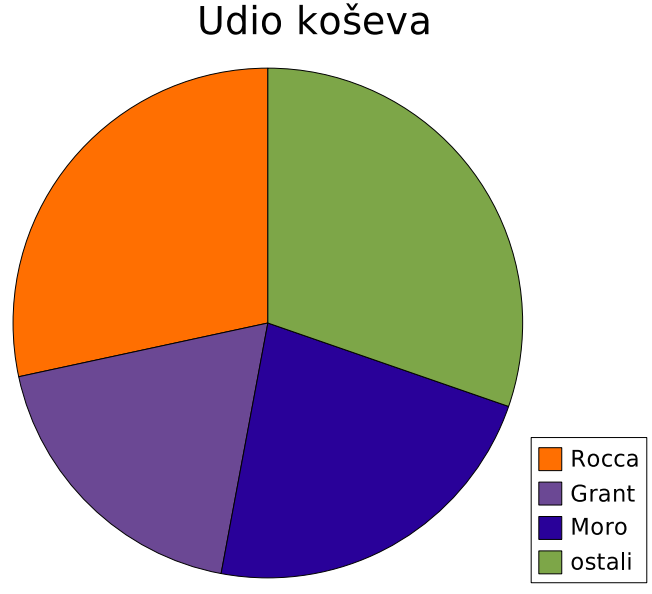
- Obrasci su iskušani recepti za organiziranje komponenti
- Rezultat: fleksibilni, nadogradivi i ponovo iskoristivi kôd
  (u skladu s načelima oblikovanja)
- Obrasci dokumentiraju načine kako optimalno zadovoljiti načela oblikovanja kod klasa učestalih problema
- Konačno, obrasci pospješuju razumljivost,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

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

Što ako poželimo imati različite prikaze?

Što ako je prikaze potrebno dinamički stvarati?

Što ako je dinamika razvoja prikaza dominantna?

Rješenje: obrazac promatrač (observer, izdavač-pretplatnik)

- glavni objekt igra ulogu izdavača (izvora informacija)
- ovisni objekti (promatrači) pretplaćuju se kod izdavača
- izdavači obavještavaju pretplatnike o promjenama;
- promatrači prekidaju pretplatu kad više ne trebaju obavještavanje

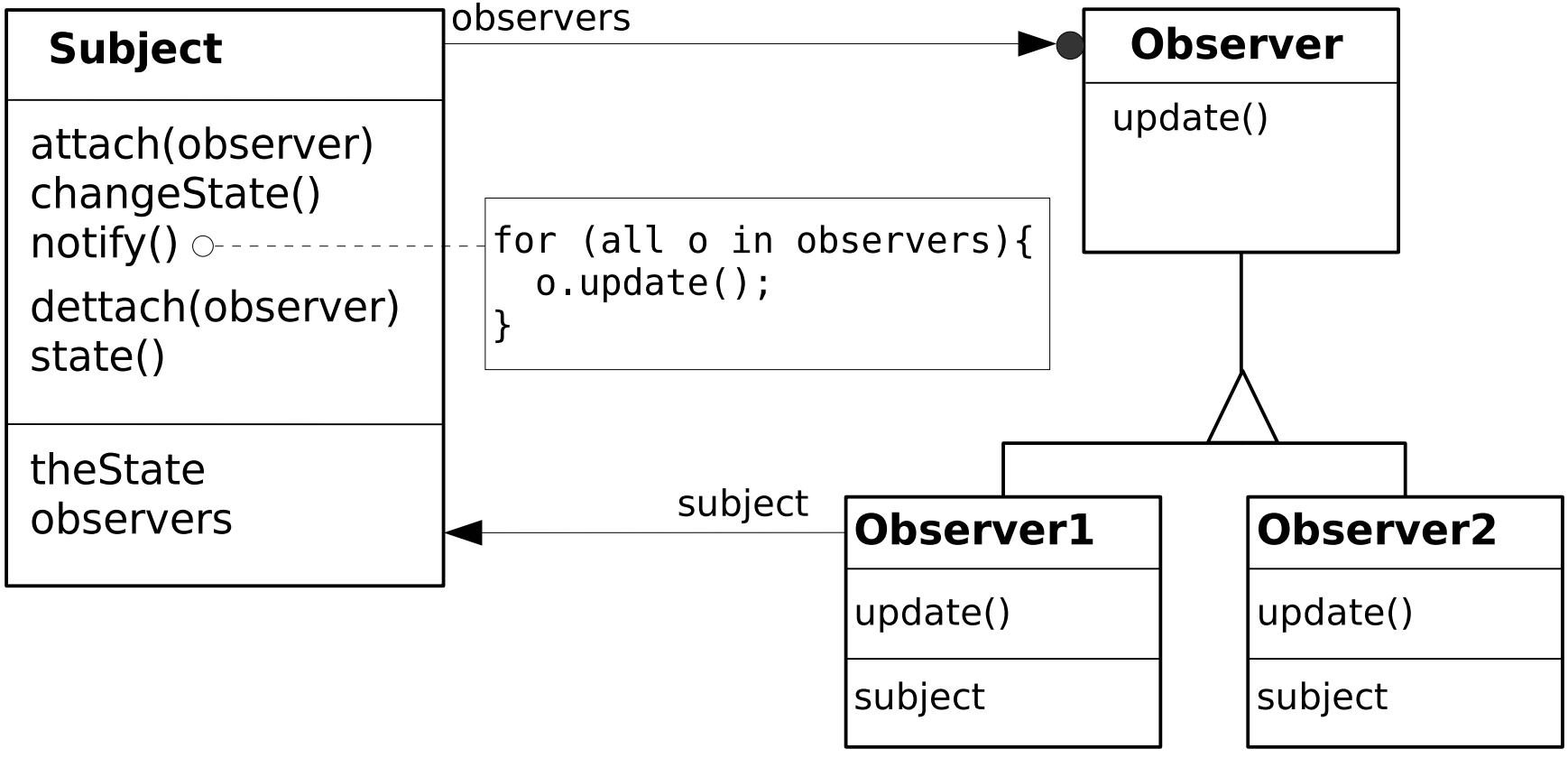
Promatrač: primjenljivost

- 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
- kad postoje dva programska entiteta, jedan ovisan o drugome, a zgodno ih je razdvojiti

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
  - obično ima kopiju podskupa stanja subjekta
  - implementira sučelje za obavještavanje

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 mogu biti inicirane i od strane Promatrača;
  Promatrač koji je inicirao promjenu također obnavlja stanje tek nakon službene obavijesti
- obavještavanje može biti inicirano ili od strane Subjekta ili od strane njegovih klijenata

Promatrač: posljedice

- ukidanje ovisnosti između Subjekta i konkretnih Promatrača! Subjekt ne ovisi o složenoj implementaciji Promatrača
- mogućnost dinamičke prijave i odjave Promatrača
- potreba za složenijim protokolom obavještavanja (sve promjene se ne tiču svih promatrača)
- pospješuje se **nadogradivost bez promjene** (dodavanje promatrača), **inverzija ovisnosti** (subjekt ovisi o apstraktnom sučelju), te **ortogonalnost** (razdvajanje promatrača i subjekta).

PROMATRAČ: IMPLEMENTACIJSKA PITANJA

odgovornost za obavještavanje odnosno obnavljanje

- 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 i u trenutku prijave:

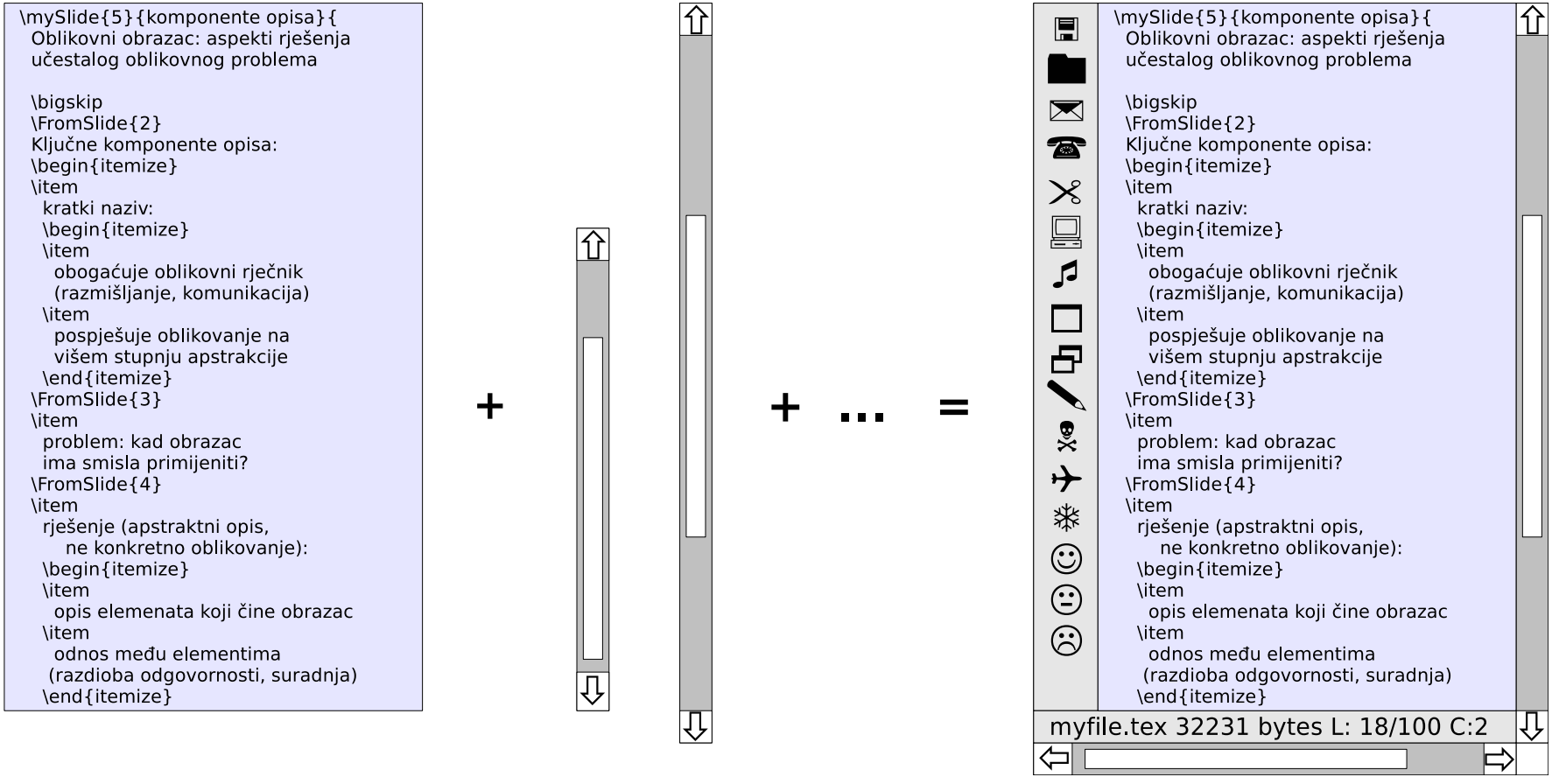
```
void Subject::attach(Observer*, Aspect& interest)
```

Promatrač: primjeri upotrebe

- obrazac Promatrač je sastavni dio arhitektonskog obrasca model - pogled - upravljač (MVC)
- obrazac promatrač je srodan oblikovnoj organizaciji dokument-pogled (MFC, Microsoft)
- elementi GUI biblioteka često se ponašaju kao Subjekti, dok se korisnički kod prijavljuje kao Promatrač (Java listeners)
- u C-u, poziv promatrača se izvodi tzv. call-back funkcijama čiji prvi argument obično identificira kontekst subjekta

DEKORATOR: NAMJERA

Dinamičko dodavanje odgovornosti pojedinom objektu



Alternativa nekontroliranom množenju izvedenih tipova, pretrpanim osnovnim razredima i miješanju odgovornosti

Dekorator: motivacija

Alternativni načini rješavanja:

```
class TextView { /* ...*/};

class TextViewScroll: public virtual TextView  { /* ...*/};

class TextViewStatus: public virtual TextView  { /* ...*/};

class TextViewScrollStatus:
  public TextViewScroll, public TextViewStatus  { /* ...*/};


class ImageView { /* ...*/};

class ImageViewScroll /* ??? */
```

- što ako želimo istu modifikaciju primijeniti na više osnovnih razreda?
- što ako **nezavisne** modifikacije želimo kombinirati **dinamički**?

Dekorator: rješenje

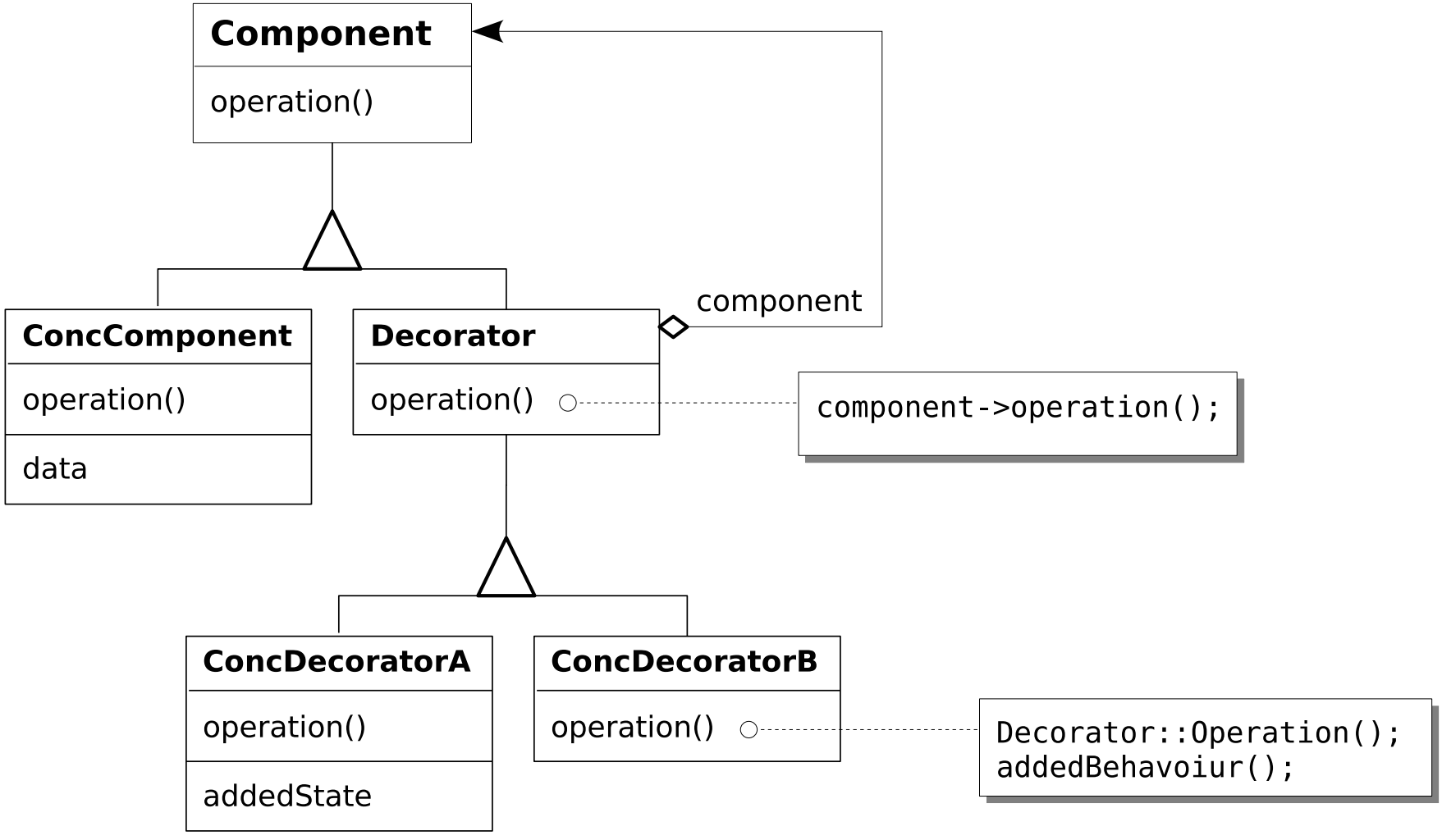
Obrazac dekorator (decorator, wrapper)

- dodjela nezavisnih odgovornosti pojedinim objektima (ne cijelim razredima!)
- temeljna ideja – rekurzivna kompozicija (handle-body idiom): umetnuti početni objekt (komponentu) u novi objekt (dekorator); dekorator implementira dodatnu odgovornost, a ostale odgovornosti prosljeđuje komponenti
- transparentnost dekoratora omogućava **rekurzivno** umetanje (pospješuje se ortogonalnost)

Dekorator: strukturni dijagram

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

Fleksibilnost se ostvaruje rekurzivnom kompozicijom (handle-body)



Dekorator: primjenljivost

- pojedinim objektima je potrebno pridodati odgovornosti **dinamički** i **transparentno** (bez utjecanja na druge komponente)
- odgovornosti je potrebno moći **povući**
- nasljeđivanje konkretne komponente nepraktično:
  - želimo dinamičku konfiguraciju
  - imamo više osnovnih razreda
  - nemamo (ne želimo) višestruko nasljeđivanje
- dodatke nije prikladno implementirati unutar osnovnog razreda jer:
  - nisu primjenljivi na sve objekte
    (što je s nepravokutnim prozorima?)
  - preglomazni su i često se mijenjaju
    (kršenje načela jedinstvene odgovornosti)

DEKORATOR: SUDIONICI

- **Komponenta** (apstraktna komponenta prikaza)
  - definira sučelje za objekte kojima se odgovornosti mogu dinamički konfigurirati
- **Konkretna komponenta** (komponenta za editiranje teksta)
  - definira izvedbu objekata s dinamičkim odgovornostima
- **Dekorator**
  - nasljeđuje sučelje Komponente
  - sadrži referencu na umetnutu komponentu
- **Konkretni dekorator** (skroliranje, statusna linija, ...)
  - 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:
  - klijenti ovise o apstraktnom razredu preko minimalnog sučelja (načelo jedinstvene dogovornosti!)
  - aplikacija ne mora održavati svojstva koja se ne koriste
- 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 (C++)

```
struct VisualComponent{
  virtual void draw() =0;
  virtual void resize() =0;
};

struct TextView:
  public VisualComponent
{
  //...
};

struct Decorator:
  public VisualComponent
{
  virtual void draw(){
    component_->draw();
  }
  virtual void resize(){
    component_->resize();
  }
public:
  VisualComponent* contents(){
    return component_;
  }
private:
  VisualComponent* component_;
};
```

```
struct DecoratorScroll: public Decorator{
  DecoratorScroll(VisualComponent* c);
  virtual void draw(){
    drawScrolls();
    Decorator::draw();
  }
  virtual void resize();
};
struct Window{
  VisualComponent* contents();
  void setContents(VisualComponent* c);
};
void client(){
  // experienced user on an ultra-portable
  VisualComponent* ptextview=new TextView
  window->setContents(ptextview);
  // show bells and whistles on a big screen
  DecoratorStatus* pdstat=
    new DecoratorStatus(ptextview);
  DecoratorScroll* pdscroll=
    new DecoratorScroll(pdstat);
  window->setContents(pdscroll);
  // back to spartan configuration
  window->setContents(ptextview);
  delete pdscroll; //shallow delete!
  delete pdstat; //shallow delete!
}
```

posebna prednost ako imamo i:

```
class ImageView: public VisualComponent { /*...*/};
```

Dekorator: izvorni kôd (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 objekti prvog razreda pa ih možemo i dekorirati:

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

```
hello()
# prints Hello world!

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

Posebna sintaksa omogućava definiranje dekoriranih funkcija

```
@makebold
def hellobold2():
    return "hello world"
hellobold2()
# prints <b>Hello world!</b>
# http://wiki.python.org/moin/PythonDecoratorLibrary
```

Dekorator: primjeri upotrebe

- transparentno dodavanje grafičkih efekata u bibliotekama GUI elemenata
- konfiguriranje dodatnih funkcionalnosti tokova podataka (streams)
- konfiguriranje pristupa bazi podataka:
  - sa ili bez logiranja
  - signalizacija pogrešaka povratnim kôdovima ili iznimkama
- Python:
  - modificirati funkciju iz eksterne biblioteke
  - debuggirati/logirati/profilirati više funkcija, ali bez ponovljenog kôda

TVORNICE: PRIMJER

Učitavanje vektorske grafike: primjer datoteke, kôd

```
# Format: ObjectType,
#   position, data, [style]
#
# Circle
C205,468 30
# Square
S300,288 20
# Rectangle
R320,128 40 20
# Polygon,
#   area style: red fill,
#   line width: .01cm
P311,222 ... ASF0xff0000 LW0.01cm
# Polygon, named style
P311,222 ... UMLClass
# Text, named style
T315,238 'ConcreteClass' UMLClass
```

```
void loadDrawing(istream& is,
  vector<Object*>& drawing)
{
  string str;
  while (getline(is,str)){
    Object* pObj(0);
    switch (str[0]){
    case 'C': pObj=new Circle; break;
    case 'S': //...
    case 'R': //...
    //...
    }

    int i=pObj->load(str.substr(1));
    pObj->applyStyle(Style(str.substr(i)));
    drawing.push_back(pObj);
  }
}
```

Očekujemo nove vrste objekata (krivulje, spojne linije, elipse, ...)

Neke vrste objekata mogu i ispasti (npr krug, nakon uvođenja elipse)

Funkcija `loadDrawing` nije zatvorena za promjene!

Tvornice: osnovna ideja

Prekrojiti `loadDrawing` u skladu s načelom ortogonalnosti

Izdvojiti dodatne odgovornosti (uzroke promjene) u zasebnu komponentu

Recept za ortogonalizaciju: enkapsuliraj i izdvoji ono što se mijenja

```
void loadDrawing(istream& is,
    vector<Object*>& drawing)
{
  string str;
  while (getline(is,str)){
    Object* pObj(createObject(str[0]));
    int i=pObj->load(str.substr(1));
    pObj->applyStyle(Style(str.substr(i)));
    drawing.push_back(pObj);
  }
}
```

Funkcija `createObject` predstavlja obrazac parametrizirane tvornice

Lokaliziranje odgovornosti za stvaranje objekata:

`loadDrawing` više ne ovisi **izravno** o konretnim razredima

TVORNICE: RAZRADA

Osnovna ideja enkapsulacije stvaranja može se dodatno razraditi:

- virtualni poziv kreacijske metode preko osnovnog sučelja (obrazac metode tvornice)
- apstraktni razred čije metode kreiraju objekte iz obitelj srodnih razreda (obrazac apstraktne tvornice)

U svakoj kombinaciji, krajnji cilj je ublažiti ovisnost klijenata o konkretnim razredima!

Metoda tvornica

Definira se sučelje za kreiranje objekta, ali se sama izvedba prepušta izvedenim razredima

Metoda tvornica omogućava statičko konfiguriranje instanciranja

Primjer – otvaranje novog dokumenta u okviru za MDI aplikaciju:

```
class Document{
  //...
};

class Application {
public:
  virtual Document* createDocument()=0;
  void newDocument();
  void openDocument();
private:
  list<Document*> docs_;
};

void Application::newDocument(){
  Document* pDoc=createDocument();
  docs_.push_back(pDoc);
  pDoc->open();
}
```

```
class MyDocument:
  public Document
{
  //...
};

class MyApplication:
  public Application
{
  virtual
  Document* createDocument(){
    return new MyDocument;
  }
}
```

Metoda tvornica: motivacija

Okvirni razredi (Document, Application) napisani prije korisničkih razreda

Korisnički razred MyApplication instanciran ručno

`Application::createDocument()` nazivamo metodom tvornicom jer je odgovorna za stvaranje novih korisničkih objekata

Metoda createDocument() stvara vezu između korisničke aplikacije i korisničkih dokumenata

METODA TVORNICA: PRIMJENLJIVOST

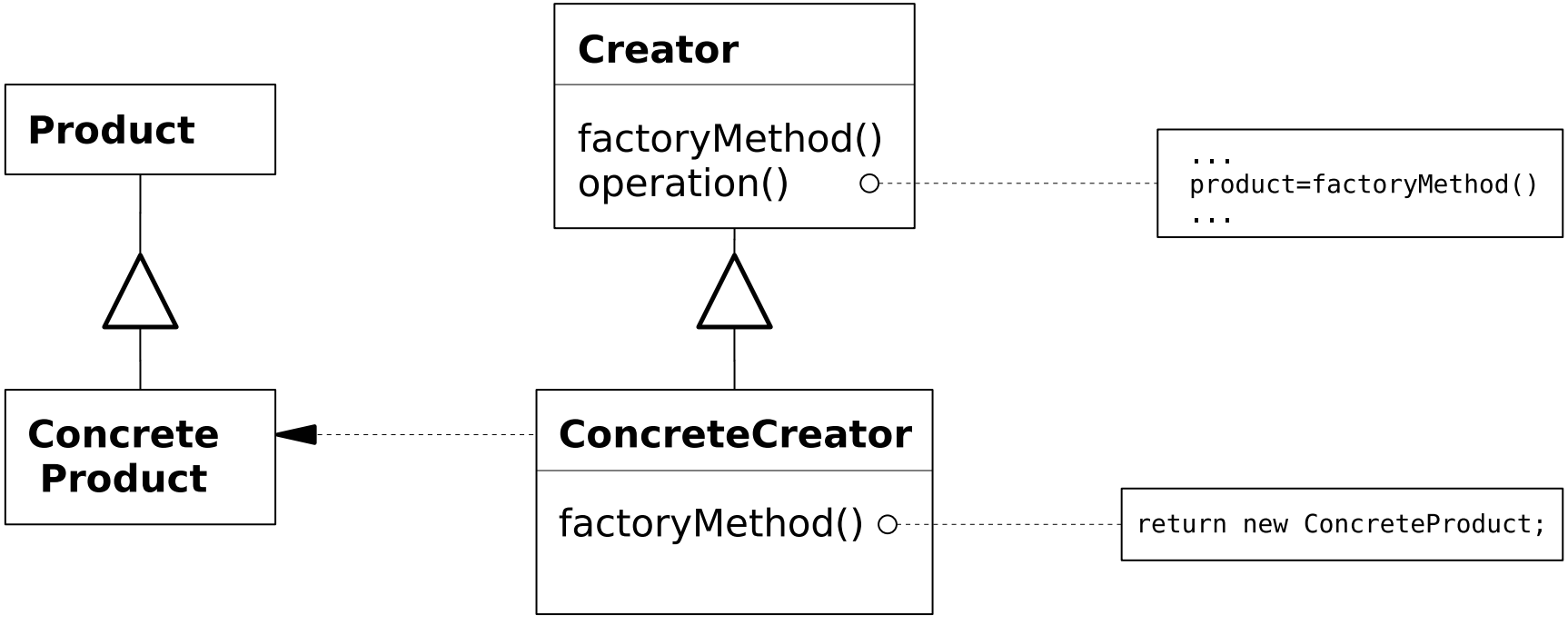
Obrazac metode tvornice (virtual constructor) koristimo kad:

- osnovni razred (Kreator) instancira objekte (Proizvode) čiji konkretni razred nije poznat
- izvedeni razred (Konkretni kreator) definira konkretni tip objekta (Konkretni proizvod) kojeg osnovni razred kreira
- klijenti povjeravaju odgovornost pomoćnom objektu (Proizvodu), a želimo lokalizirati znanje o njegovom konkretnom tipu
- želimo povezati stvaranje raznorodnih razreda (Konkretni kreator, Konkretni proizvod)

METODA TVORNICA: STRUKTURNI DIJAGRAM

Metoda tvornica: **kreacijski** obrazac u domeni **razreda**

Fleksibilnost se ostvaruje nasljeđivanjem



Metoda tvornica: sudionici i suradnja

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

Suradnja: Kreator izvedenim razredima prepušta izvedbu metode tvornice odnosno instanciranje odgovarajućeg Konkretnog proizvoda

Posljedice: pospješuje se inverzija ovisnosti u osnovnim razredima

Metoda tvornica: implementacijska pitanja

S obzirom na metodu tvornicu, kreator može biti ili apstraktan ili ponuditi podrazumijevanu implementaciju

Metoda tvornica može biti parametrizirana (kao u uvodnom primjeru)

Metode tvornice se ne mogu zvati iz konstruktora apstraktnog Kreatora (zašto?)

Metode tvornice se mogu koristiti za povezivanje paralelnih hijerarhija razreda (analogno apstraktnoj tvornici)

Područje primjene: programski okviri za razvoj aplikacija (framework, toolkit)

APSTRAKTNA TVORNICA: NAMJERA

Pruža sučelja za stvaranje obitelji srodnih objekata, bez navođenja njihovih konkretnih razreda.

Vrlo slično metodi tvornici, ali:

- apstraktna tvornica odgovorna za instanciranje **obitelji** objekata
- kod apstraktne tvornice, kreacijske metode pozivaju klijenti (kod metode tvornice, kreacijsku metodu poziva matični razred)
- kod apstraktne tvornice koristi se delegacija, dok metoda tvornica koristi nasljeđivanje

Konačni cilj: istovremeni odabir grupe izvedbenih detalja

APSTRAKTNA TVORNICA: MOTIVACIJA

Primjer korištenja unutar hipotetske GUI biblioteke koja podržava višestruke mogućnosti prikaza (gdk, wxWindows, MS Windows, ...)

odabir elemenata sučelja je (i) konzistentan i (ii) može se konfigurirati

```
class FactoryAbstract{
  virtual Window* createWindow() =0;
  virtual ScrollBar* createScrollBar() =0;
}

class FactoryGDK: public class FactoryAbstract{
  virtual Window* createWindow();
  virtual ScrollBar* createScrollBar();
}

class FactoryWin32: public class FactoryAbstract{
  virtual Window* createWindow();
  virtual ScrollBar* createScrollBar();
}

void client(FactoryAbstract& theFactory){
  Window wnd=theFactory.createWindow();
}
```

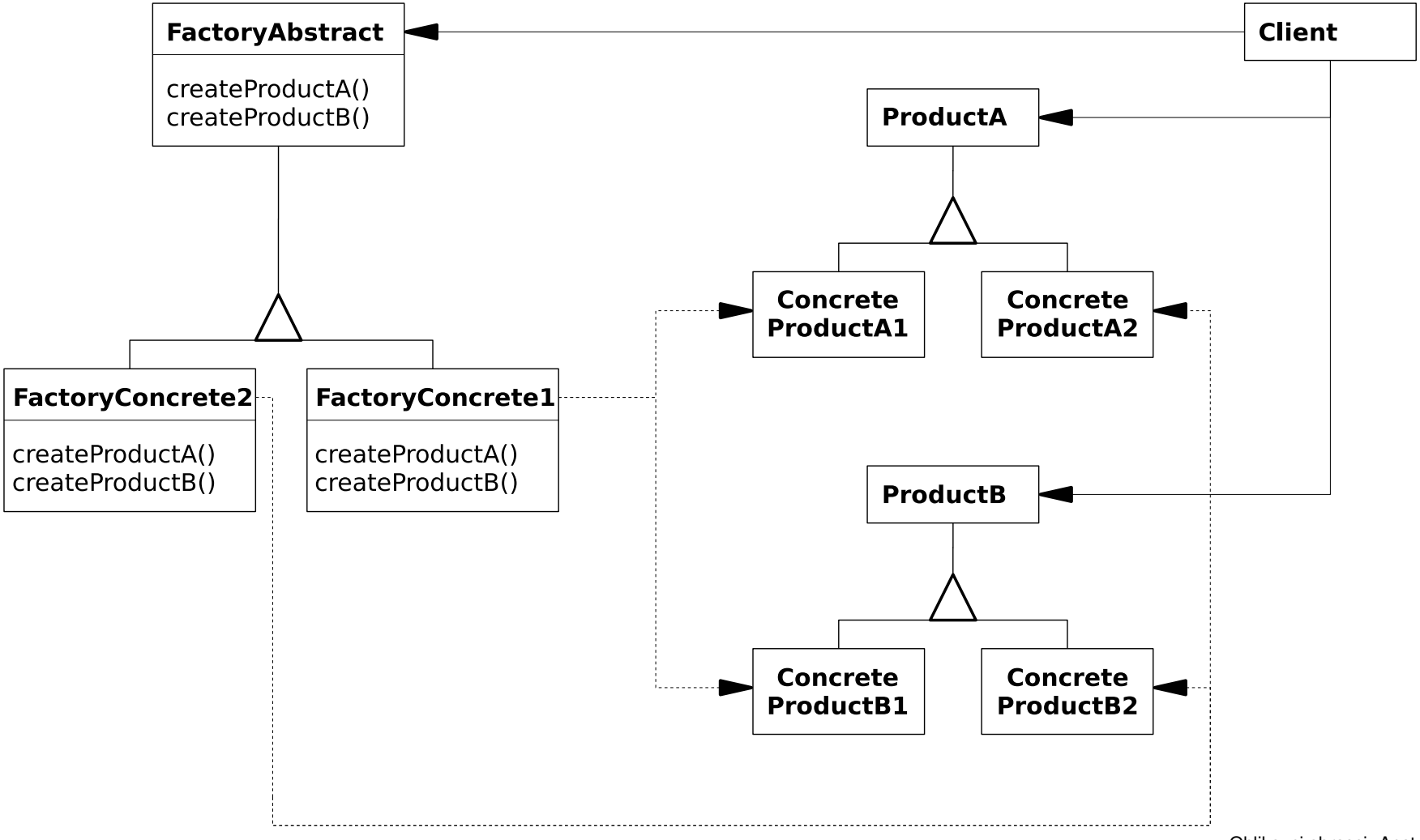
APSTRAKTNA TVORNICA: PRIMJENLJIVOST

- želimo da programski sustav bude neovisan o instanciranju objekata
- sustav treba konfigurirati s jednom ili više obitelji proizvoda
- obitelj srodnih proizvoda koncipirana je tako da se koristi zajedno ("princip sve ili ništa")
- želimo da klijenti budu **izolirani** od implementacija biblioteke razreda

APSTRAKTNA TVORNICA: STRUKTURNI DIJAGRAM

Apstraktna tvornica: **kreacijski** obrazac u domeni **objekata**,

Fleksibilnost se ostvaruje povjeravanjem i nasljeđivanjem



Apstraktna tvornica: sudionici

- **Apstraktna tvornica** (tvornica GUI elemenata)
  - deklarira sučelje za instanciranje apstraktnih proizvoda
- **Konkretna tvornica** (tvornice GDK, WxWindows, ...)
  - instancira konkretne objekte iz odgovarajuće obitelji
- **Apstraktni proizvod** (GUI element)
  - Zajedničko sučelje apstraktnih proizvoda dane vrste
- **Konkretni proizvod** (GDK prozor, ...)
  - proizvod iz obitelji definiranom odabranom tvornicom
- **Klijent** (aplikacija)
  - koristi apstraktna sučelja tvornice i proizvoda
  - konkretna tvornica se kreira na najvišoj razini programa

APSTRAKTNA TVORNICA: POSLJEDICE

- izoliranje klijenata od konkretnih razreda
- lako izmjenjivanje obitelji proizvoda
- pospješuje kozistentnost verzija proizvoda
- proširivanje obitelji proizvoda nije lako:
  implicira mijenjanje tvornice i svih izvedenih razreda

APSTRAKTNA TVORNICA: IMPLEMENTACIJA I PRIMJENE

Konkretne tvornice obično instancirane statički (Singleton)

Kreiranje pojedinih proizvoda često slijedi obrazac metode tvornice (ali, može se koristiti i kreacijski obrazac prototip)

Varijanta: sve objekte istog osnovnog tipa kreirati **jednom** parametriziranom metodom tvornicom

- lakše proširivanje tvornice novim proizvodima
- potreba za kasnijom eksplicitnom konverzijom (`dynamic_cast`)

Područje primjene: programski okviri za razvoj aplikacija (framework, toolkit)

JEDINSTVENI OBJEKT: NAMJERA

Osigurati da razred ima samo jednu instancu, omogućiti toj instanci globalni pristup

Važno je da neki razredi imaju točno jednu instancu:

- enkapsuliranje pristupa sklopovskim i programskim resursima (ekran, serijski port, biblioteke za pristup sklopovlju, bazama)
- parametrizirana tvornica s registrom razreda
- procedura s ispisnim stanjem (npr, pretvorbena tablica, svojstva sustava)

Jedinstveni objekt (singleton) je globalna varijabla s odgođenim instanciranjem i zabranom daljnjeg instanciranja

JEDINSTVENI OBJEKT: MOTIVACIJA

Rješenje je jednostavno, osim ako moramo razmatrati konkurentnost!

```
//MySingleton.hpp
class MySingleton{
private:
  MySingleton();
  ~MySingleton();
public:
  static MySingleton& instance();
public:
  //interface
private:
  static MySingleton* pInstance_;
};
```

```
//MySingleton.cpp
MySingleton* MySingleton::pInstance_=0;

MySingleton& MySingleton::instance(){
  //use synchronization in
  //multi-threaded applications!
  if (pInstance_==0){
    pInstance_=new MySingleton;
  }
  return *pInstance_;
}
```

Primjenljivost:

- točno jedna, globalno pristupačna instanca
- jedinstvenu instancu prikladno instancirati s odgodom (kad se javi potreba)

JEDINSTVENI OBJEKT: SUDIONICI I SURADNJA

Sudionici:

- **Jedinstveni objekt**:
  - definira statičku metodu za pristup objektu `instance()`
  - definira privatni (ili zaštićeni) konstruktor
  - pristupna metoda može implementirati odgođeno instanciranje
- **Klijenti**:
  - pristupaju Jedinstvenom objektu preko pristupne metode `instance()`

JEDINSTVENI OBJEKT: IMPLEMENTACIJA

Prednosti dinamičkog instanciranja nad statičkim:

- ubrzavanje pokretanja programa
- eksplicitni pristup objektu $\Rightarrow$ rješavanje problema međuovisnosti globalnih objekata
- dodatni kontekst (konfiguriracijska datoteka, komandna linija)
- međutim, u konkurentnom okruženju, statičko instanciranje može biti bolji odabir

JEDINSTVENI OBJEKT: IMPLEMENTACIJA (2)

Usporedno izvođenje u konkurentnom programu:

- potrebno uvesti međusobno isključivanje u metodi `instance()`
- to može biti skupo (vremenski):

```
// Singleton with straight-forward locking
MySingleton* MySingleton::pInstance_=0;
MySingleton& MySingleton::instance(){
  MutexLock l(mutex_);
  if (pInstance_==0){
    pInstance_=new MySingleton;
  }
  return *pInstance_;
}
```

Nažalost, nema lakih rješenja, opcije su:

- prihvatiti cijenu sinkronizacije
- pristupati preko cacheirane reference
- prihvatiti cijenu statičkog instanciranja

JEDINSTVENI OBJEKT: IMPLEMENTACIJA (3)

Isključivanje s dvostrukom provjerom (double-checked locking pattern) može rezultirati neželjenim nedeterminizmom:

```
// caveat: the double-checked locking pattern may subtly fail!
MySingleton& MySingleton::instance(){
  if (pInstance_==0){
    MutexLock l(mutex_);
    if (pInstance_==0){
      pInstance_=new MySingleton;
      // compiler generates:
      // (i)  pInstance_=operator new(sizeof(MySingleton));
      // (ii) new (pInstance_) MySingleton;
    }
  }
  return *pInstance_;
}
```

Navodno, poboljšani memorijski model Jave 1.5 rješava probleme (to je moguće jer Java "zna" za konkurentnost)

C++ ne želi znati ništa o konkurentnosti kako bi mogao bolje optimirati (C++0x je navodno bolji po tom pitanju)

`http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf`

JEDINSTVENI OBJEKT: IMPLEMENTACIJA (3)

Ako nam je pozivanje destruktora Jedinstvenog objekta važno:

- Instancu možemo deklarirati kao lokalnu statičku varijablu (ne radi u konkurentnom programu)

```
MySingleton& MySingleton::instance(){
  static MySingleton instance_;
  return instance_;
}
```

- Instancu možemo eksplicitno pobrisati iz destruktora statičkog člana

```
class MySingleton {
  //...
private:
  struct MyDestructor{
    ~MyDestructor(){
      delete MySingleton::pInstance_;
    }
  };
  friend struct MyDestructor;
  static MyDestructor dtor_;
};

// MySingleton.cpp
MySingleton* MySingleton::pInstance_=0;
MySingleton::MyDestructor MySingleton::dtor_;
```

JEDINSTVENI OBJEKT: PRIMJENE

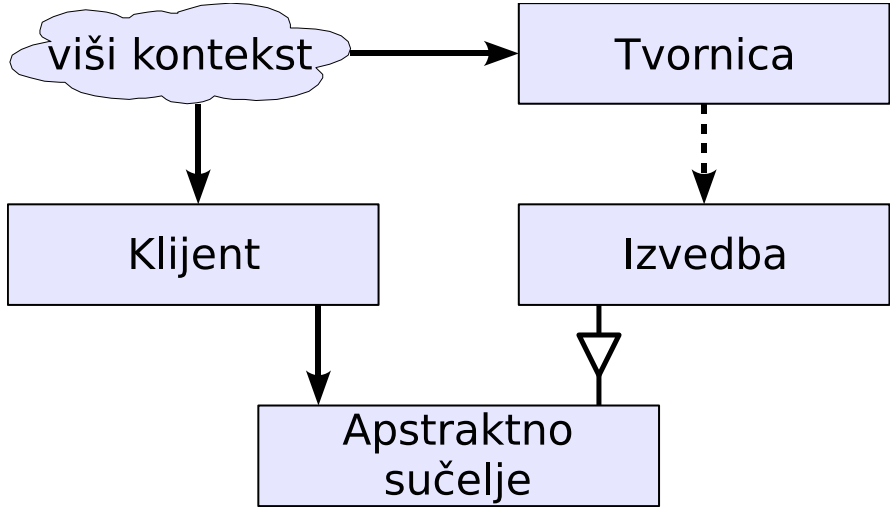
Razredi kreacijskih obrazaca se često izvode kao jedinstveni objekti:
Tvornica, Prototip, Graditelj

Enkapsuliranje pristupa sklopovskim i programskim resursima
(ekran, serijski port, biblioteke za pristup sklopovlju, bazama)

Procedura s ispisnim stanjem
(npr, pretvorbena tablica, svojstva sustava)

TVORNICE: PROBLEM OVISNOSTI

U svim dosadašnjim rješenjima, ipak postoji ovisnost vršne komponente programa o konkretnim izvedbama...



$\Rightarrow$ nove izvedbe ne možemo dodavati bez mijenjanja postojećeg koda!

To može biti problem, mi bismo htjeli:

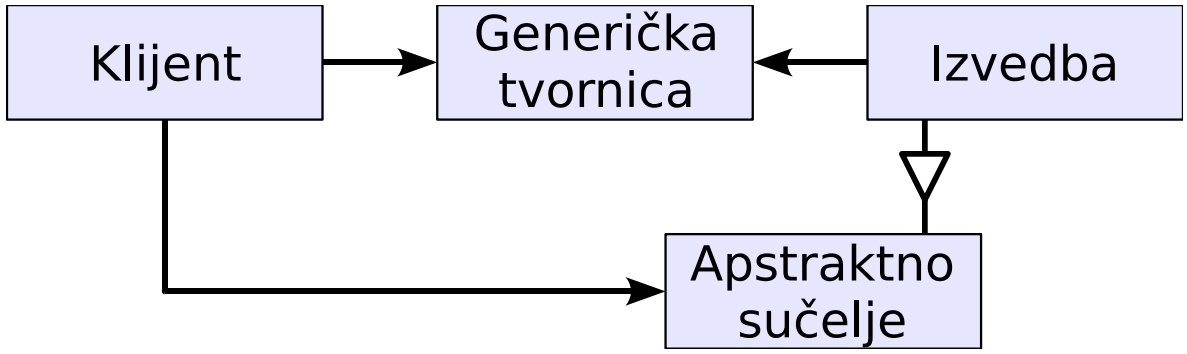
1. da dodavanje novih izvedbi zahtijeva što manje intervencija
2. da korisnici mogu pisati proširenja (npr, flash plugin za browsere)
3. ukratko, postići **nadogradnju bez promjene**

TVORNICE: ŽELJENO RJEŠENJE

Kako bismo postigli poboljšanje u tom smislu (tj, da nove tipove možemo dodati **bez mijenjanja** postojećeg koda)?

Konvencionalni prevođeni jezici tijekom izvođenja najčešće ne podržavaju introspektivna pitanja poput: "da li je u projekt ukompajliran razred za učitavanje slika u formatu jpeg?"

U konvencionalnim jezicima, konkretne komponente moraju se stoga nekako **registrirati** kod tvornice, idealno tijekom statičke inicijalizacije



TVORNICE: SKICA RJEŠENJA

```
//pkg1_base.hpp
class pkg1_base{
  // declarations of some
  // pure virtual methods
};
//pkg1_client.cpp
//no dependency on concrete classes!
#include "../util/util_factory.hpp"
#include "../pkg1/pkg1_base.hpp"
...
  std::string s;
  std::cout <<"Enter object type\n";
  std::cin >>s;
  pkg1_base* pobj= (pkg1_base*)
    util::Factory::instance().
      create(s);
...
```

```
//util_factory.hpp
namespace util{
class Factory{
private:      //we are singleton
  Factory();
public:
  static Factory& instance();
public: // audience: concrete products
  typedef void* (*MyPtrFun)();
  int registerCreator(
    const std::string& id, MyPtrFun pfn);
public: // audience: instantiating clients
  void* create(const std::string& id);
  // additional methods provide access to
  // identifiers of the registered classes
};
}
```

```
//pkg1_myclass.cpp
#include "pkg1_base.hpp"
#include "../util/util_factory.hpp"
class pkg1_myclass: public pkg1_base{
  // virtual methods declared
};
static void* myCreator(){
  return new pkg1_myclass;
}
static int hreg=util::Factory::instance().
    register("pkg1_myclass", myCreator);

// virtual methods implemented
```

TVORNICE: RJEŠENJE U PYTHONU

Svaki razred "novog stila" (`object`) sadrži polje izvedenih razreda!

To polje dobivamo pozivom metode `__subclasses__()` osnovnog tipa

Iteriranjem po tom polju možemo propitivati atribute izvedenih tipova.

Kad dođemo do odgovarajućeg izvedenog tipa - naprosto ga instanciramo!

```
class Base(object):
  pass
class Concrete1(Base):
  @staticmethod
  def name():
    return 'name1'
class Concrete2(Base):
  @staticmethod
  def name():
    return 'name2'
```

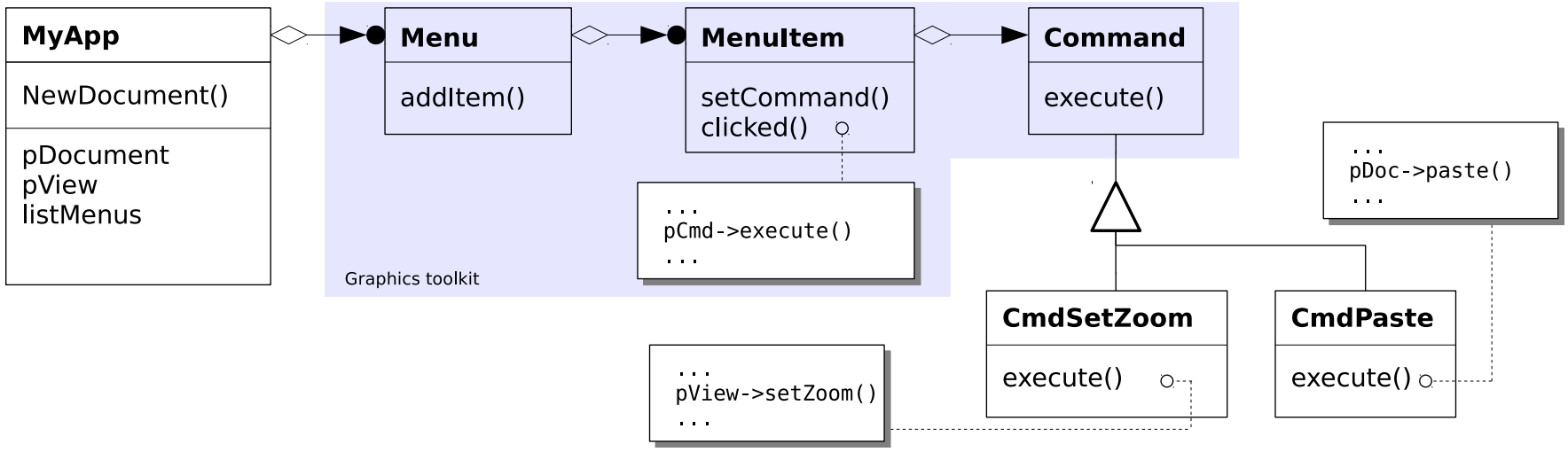
```
def factory(baseclass, name):
  for cls in baseclass.__subclasses__():
    if cls.name()==name:
      return cls()
  raise ValueError

x1=factory(Base, 'name1')
print(type(x1)) # <class '__main__.Concrete1'>
x2=factory(Base, 'name2')
print(type(x2)) # <class '__main__.Concrete2'>
```

Podsjetnik: razredi u Pythonu su također objekti!

NAREDBA: NAMJERA

Omogućiti unificirano baratanje raznorodnim zahtjevima

Motivacijski primjer: implementiranje konfigurabilnih elemenata korisničkog sučelja (meni, dugme, ...)



Ideja: razdvojiti zaprimanje i pokretanje korisničkog zatjeva od implementacijskih detalja konkretne obrade

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

NAREDBA: MOTIVACIJA

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

- otežano ponovno korištenje menija čija izvedba ne bi mogla biti zatvorena u biblioteci
- karikirano, svaka aplikacija bi pisala svoje menije

Kad bi svi `MenuItemi` pozivali `MyApp::event`, implementacija bi:

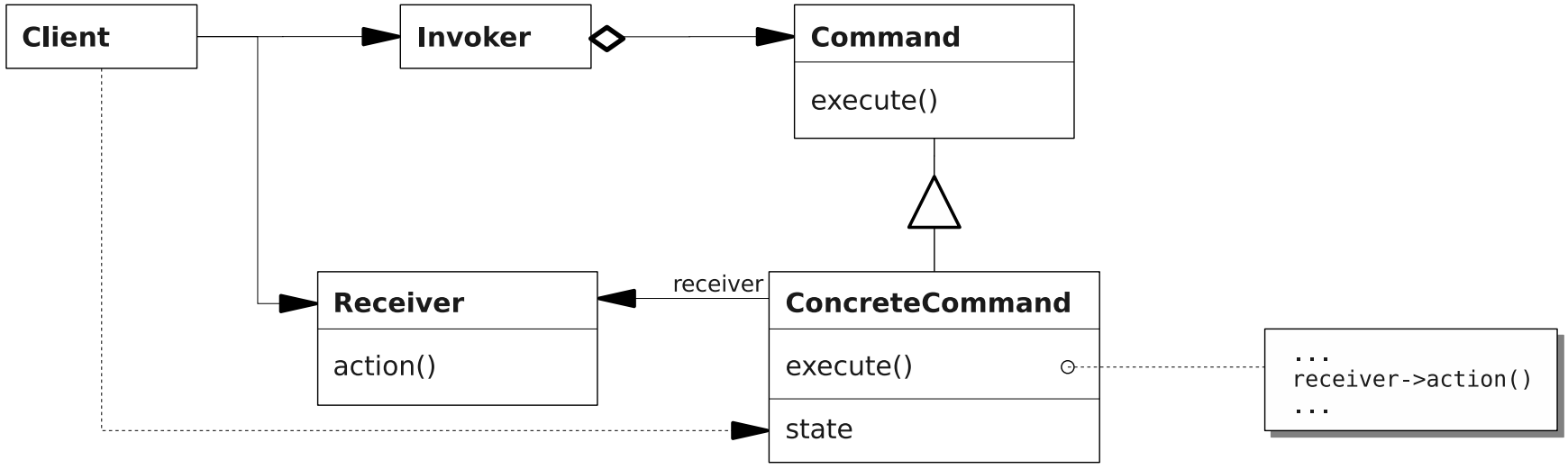
- imala puno **`if`**ova
- ne bi bila nadogradiva bez promjene

Naredbom postižemo distribuciju poslova bez da išta znamo o konkretnim operacijama koje je potrebno poduzeti

- dinamička konfigurabilnost umjesto ožičene funkcionalnosti

NAREDBA: STRUKTURNI DIJAGRAM

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



Analogija iz fizičkog svijeta (restoran brze prehrane):

- Klijent: gost (naručuje jelo)
- Naredba: narudžba (predefinirani izbor: ćevapi, lepinja, luk)
- Pozivatelj: konobar (nosi listić s narudžbom kuharu)
- Primatelj: kuhar (sprema narudžbu i izvršava je kad dođe na red)

NAREDBA: PRIMJENLJIVOST

- potrebno parametrizirati objekt s procedurom koju valja obaviti (specijalan slučaj strategije)
- zadavanje, raspoređivanje i izvođenje zadataka se zbiva u vremenski odvojenim intervalima
- potrebno podržati operaciju undo ili grupiranje operacija u makroe
- potrebno podržati inkrementalno bilježenje (logiranje) naredbi za slučaj oporavka nakon kvara sustava
- potrebno podržati složene atomarne operacije (transakcije)

Naredba: sudionici

Naredba (Command, listić s narudžbom):

- deklarira jednostavno sučelje za izvođenje operacije

Konkretna naredba (CmdPaste, ćevapi s lukom):

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

Pozivatelj (MenuItem, konobar):

- inicira zahtjev preko sučelja Naredbe

Primatelj (MyDocument, kuhar):

- 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 naknadni opoziv (*undo*)

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 Promatraču

Nedostatak: velik broj malih razreda (može se ublažiti predlošcima)

Naredba: primjene

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

Bilježenje operacija u cilju omogućavanja oporavka nakon kvara sustava

Podržavanje opozivih akcija, uključujući transakcijsko poslovanje

Korisnička sučelja, npr Java Swing:

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