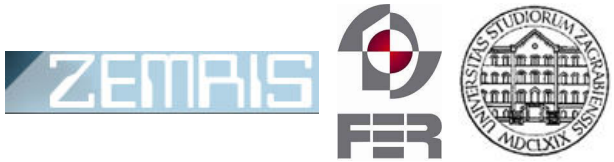


Oblikovni obrasci u programiranju

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, Posjetioc.
- **Strukturni** obrasci: Prilagodnik, Most, Kompozicija, Dekoracija, Pročelje, Dijeljeni objekt, Surogat.
- Obrasci **stvaranja**: Apstraktna tvornica, Metoda tvornica, Jedinstveni objekt, Prototip, Graditelj.

Uvod: primjer

Ponovo primjer programa za obradu slike, s podatnim pribavljanjem, obradom i spremanjem slika:

```
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(4);
  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));
  }
}
```

Ideja: **povjeriti** (delegate) dio posla kroz pokazivač na osnovni razred

Podržavaju se temeljna načela (jedinstvena odgovornost, otvorenost)

Uvod: prvi obrazac!

- I to je bio naš prvi **oblikovni obrazac** (naziv: strategija)!
  - "općenito rješenje učestalog oblikovnog problema"
    (problem: odvojiti izvedbu algoritma od klijenata)
  - "široko primjenljivi arhitektonski element"
    (ortogonalne osi promjenljivosti modula)
- **Prednosti** obrazaca:
  - oblikovanje je teško: iskusni arhitekti cijene **dokazana** rješenja
  - produktivnost oblikovanja, obogaćeni rječnik, laka komunikacija
- **Kako koristiti obrasce**:
  - obrasci komplementarni **bibliotekama** (izvedba vs. organizacija)
  - različite **razine** (idiomi, oblikovni obrasci, arhitektonski obrasci)
  - primjena: osmišljavanje novog, prekrajanje starog kôda

Uvod: razine

- Arhitektonski obrasci: struktura programa na najvišoj razini
  - najviša razina apstrakcije, organizacija podsustava (10 KLoC)
  - npr: model-pogled-upravljač, klijent-poslužitelj, ravnopravna mreža, troslojna arhitektura, oglasna ploča
- 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://jaynes.colorado.edu/PythonIdioms.html`
    - `http://en.wikibooks.org/wiki/More_C++_Idioms`
  - npr: `for(i=0; i<n; ++i)`, konstruktori, const korektnost, ...

Uvod: komponente opisa

Oblikovni obrazac: aspekti rješenja učestalog 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)
- posljedice:
  - prednosti, nedostatci, kompromisi
  - utjecaj na **nadogradivost**, **proširivost**, višestruko korištenje

STRATEGIJA: OPIS

- Problem:

  Obrazac strategije je koristan kad je potrebno dinamički mijenjati postupke koji se primijenjuju u aplikaciji. Obrazac omogućava izmjenu postupaka neovisno o kontekstu iz kojeg se koriste. To je posebno korisno ako kontekst zahtijeva više obitelji postupaka.

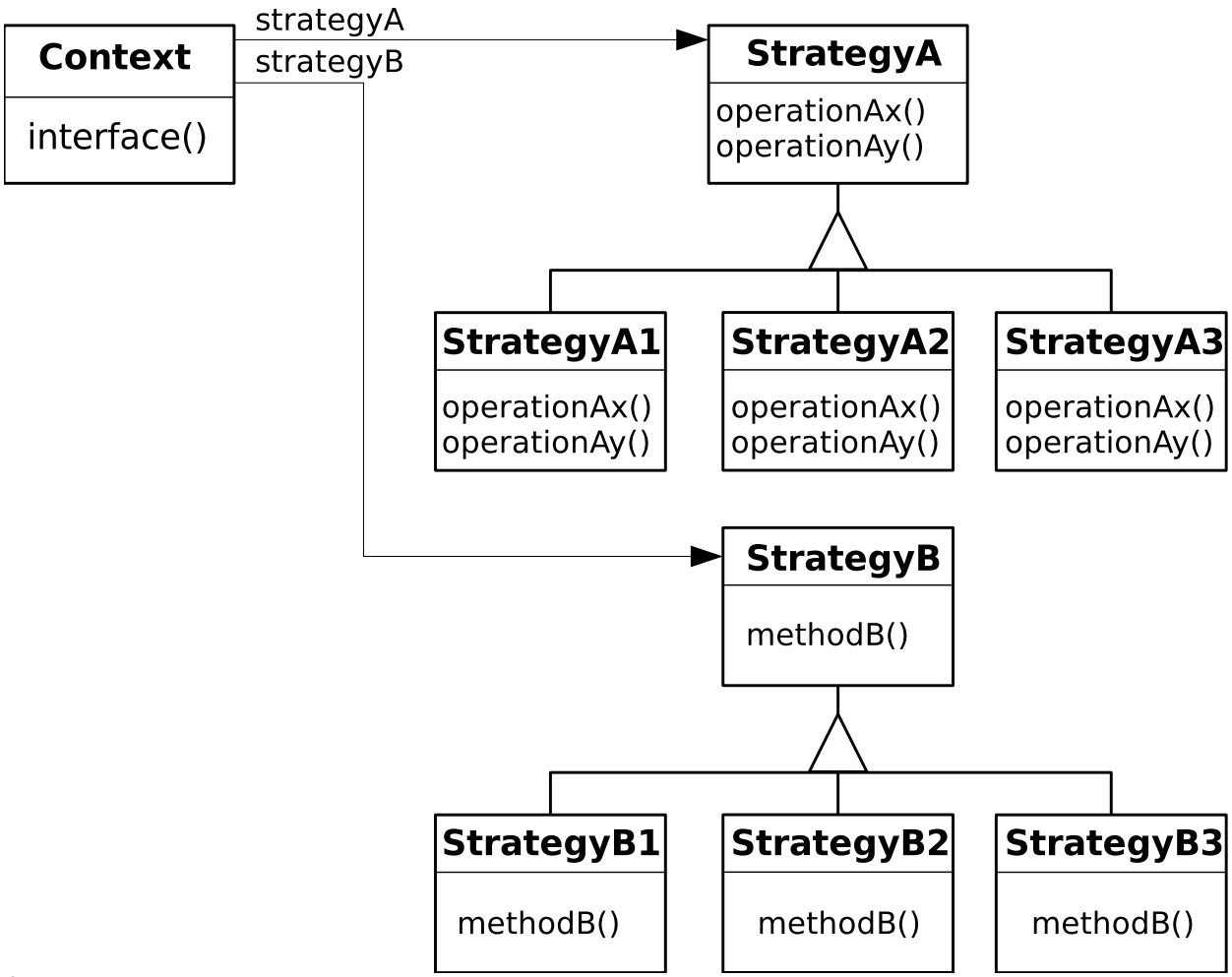
- Rješenje:

  Definirati hijerarhiju(e) postupaka, enkapsulirati konkretne postupke, omogućiti izmjenjivanje prema potrebi. Kontekst povjerava zadatke konkretnim postupcima preko sučelja osnovnog razreda.

- Posljedice:

  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: GoF opis

- **Naziv**, alternativni nazivi (strategy, policy)
- **Namjera**, motivacijski primjer, primjenljivost
- **Struktura**, sudionici, suradnja
- **Posljedice**, implementacijska pitanja, izvorni kod, primjeri upotrebe, srodni obrasci.

STRATEGIJA: PRIMJENLJIVOST

- dinamička konfiguracija konteksta sa željenim ponašanjem
  (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 unutar naredbe za grananje toka:
  dinamički polimorfizam umjesto krutog odabira grananjem
- dinamička konfiguracija nije presudna $\Rightarrow$ statički polimorfizam
  (spremnici STL-a, postupci su stvaranje, pridjeljivanje i oslobađanje)

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 reference 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 dostavljanje konkretne Strategije
  - kasnije, klijenti komuniciraju isključivo s Kontekstom
  - klijent često odabire iz čitave obitelji konkretnih Strategija
  - najveća korist obrasca: potreba za više obitelji postupaka (ortogonalna konfiguracija, nadlinearni rast funkcionalnosti)

STRATEGIJA: POSLJEDICE

- **strategija** implicira obitelji srodnih algoritama
  (zajednička funkcionalnost se može izdvojiti nasljeđivanjem)
- podatnija **alternativa** nasljeđivanju:
  - razdvajanje konteksta of ponašanja
  - mogućnost ortogonalnog skupa postupaka
- eliminacija uvjetnih izraza
- mogućnost primjene u slučaju srodnih implementacija
- veća složenost (ali i podatnost) 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: PRIMJERI UPOTREBE

- različiti načini prikaza grafičkih elemenata
- različite optimizacijske strategije za (i) dodjeljivanje registara i (ii) redosljed instrukcija
- različiti postupci provlačenja vodova između elektroničkih elemenata
- način alociranja memorije STL spremnika
- različiti postupci validacije korisničkog unosa u GUI dijalozima (podrazumijevani postupak je bez validacije)

STRATEGIJA: GOF KLASIFIKACIJA

Podjela prema **namjeni**:

- Kreacijski obrasci apstrahiraju **instanciranje** novih objekata
- Strukturalni: **sastavljanje** razreda i objekata u veće strukture
- Ponašajni: bave se **postupcima** i raspodjelom odgovornosti

Podjela prema **domeni primjene**:

- Obrasci razreda razmatraju odnose među razredima i podrazredima
- Obrasci objekata usredotočavaju se na odnose među objektima

Mehanizmi za ostvarivanje podatnosti:

- Nasljeđivanje: dinamičko povezivanje, statička konfiguracija (**Python!**)
- Povjeravanje: dinamička konfiguracija (obrasci objekata)!
- Generici: statička konfiguracija i povezivanje, efikasnost

STRATEGIJA: KLASIFIKACIJA

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

Podatnost se ostvaruje:

- kombinacijom povjeravanja i nasljeđivanja
- generičkim programiranjem

Povjeravanje omogućava **dinamičku** konfiguraciju

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

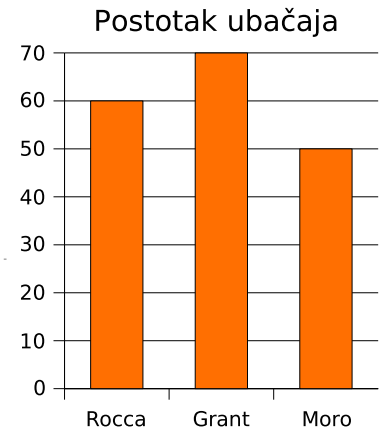
STRATEGIJA: PREDNOSTI OBRAZACA

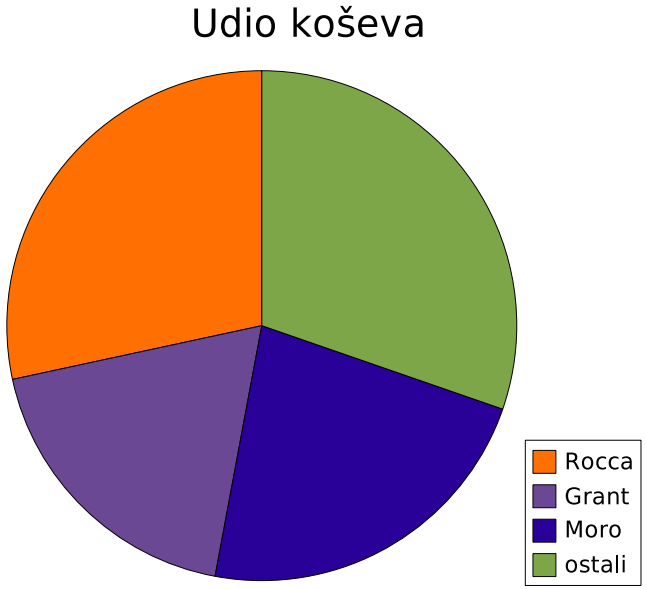
- Obrasci su iskušani recepti za primjenu programskih metodologija
- Rezultat: podatni, nadogradivi i ponovo iskoristivi kôd
  (u skladu s načelima oblikovanja)
- Problem je što je i najmoderniju metodologiju lako zloupotrijebiti ili suboptimalno koristiti; optimalna rješenja često nisu očita, zahtijevaju mnogo promašenih pokušaja
- Obrasci dokumentiraju načine kako optimalno zadovoljiti načela oblikovanja kod klasa učestalih problema
- Konačno, obrasci pospješuju razumljivost, komunikaciju, dokumentaciju

PROMATRAČ: NAMJERA

Potrebno je ostvariti **1:n ovisnost** među objektima: 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);
  barChartDisplay.update(double* data);
}
```

Što ako poželimo imati različite prikaze?

Što ako poželimo prikaze dinamički konfigurirati?

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;
  pretplatnici dolaze do željenih podataka preko reference na izdavača
- promatrači prekidaju pretplatu kad više ne trebaju obavještavanje

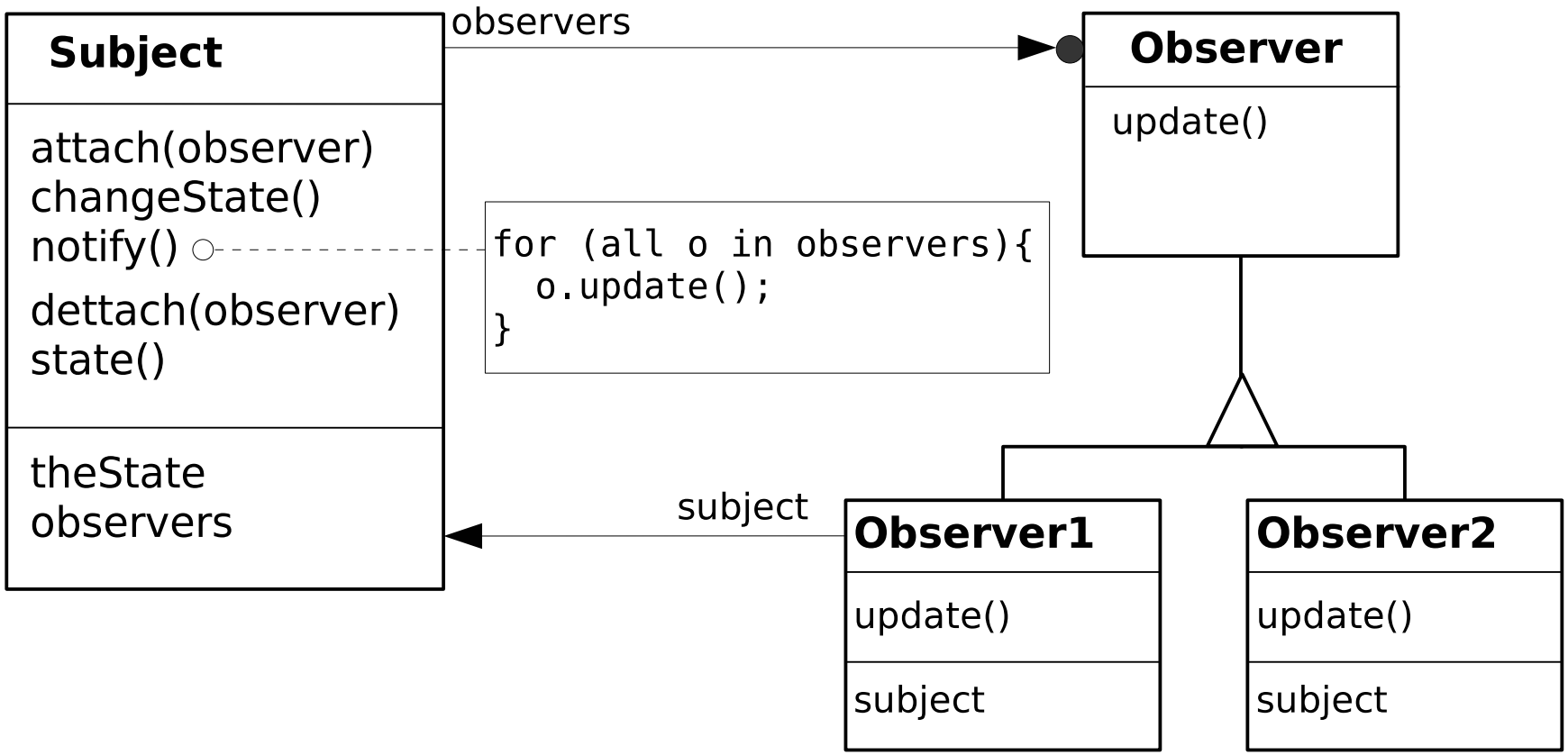
Promatrač: primjenljivost

- kad postoje dva aspekta, jedan ovisan o drugome, a zgodno ih je razdvojiti
- 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**

Podatnost 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
- **konkrektan 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

- dokidanje 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 obavješavanje, nakon unosa niza promjena (mogućnost grešaka)

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

promatrači mogu definirati svoje interese i u trenutku prijave:

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

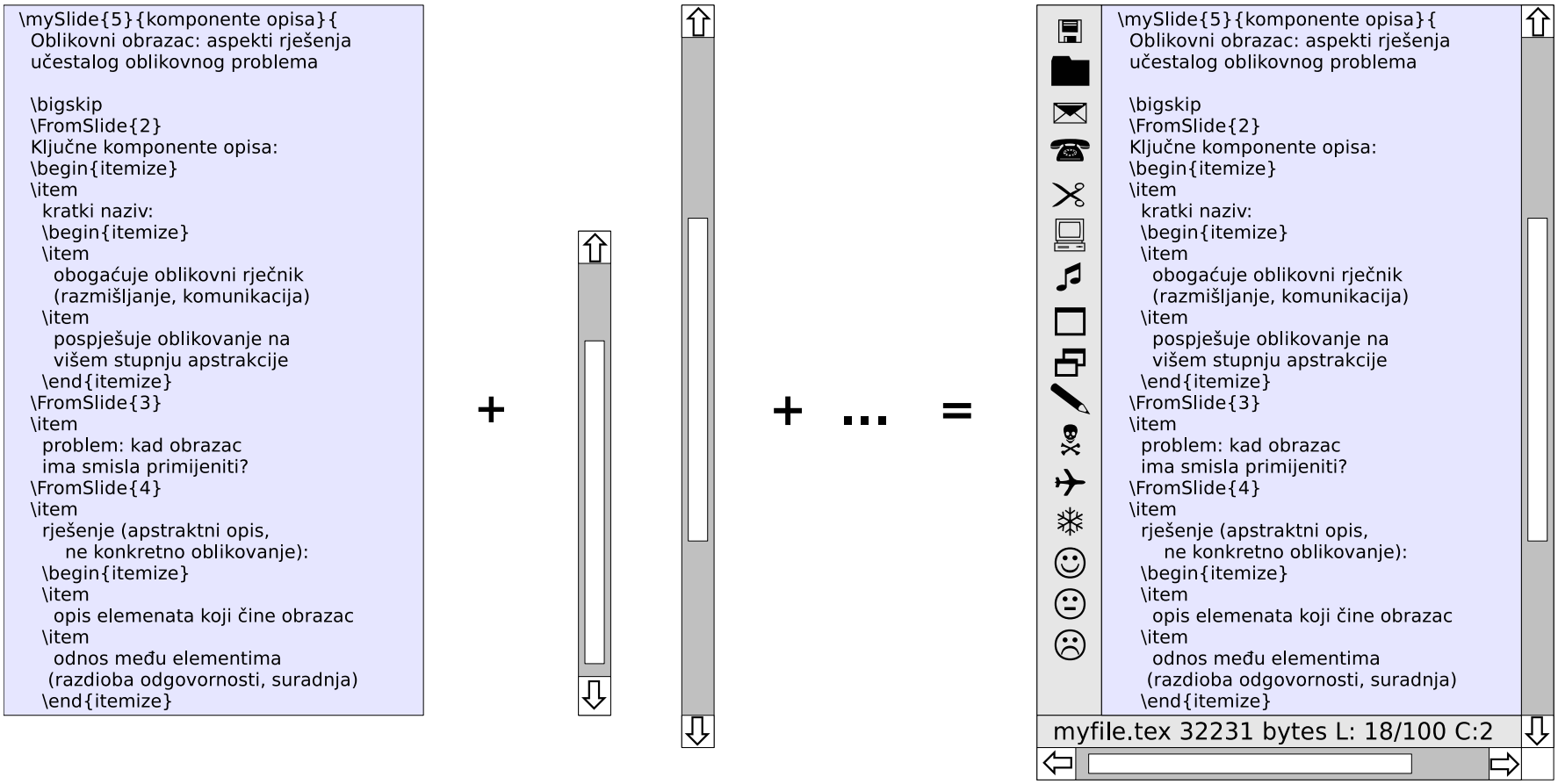
Promatrač: primjeri upotrebe

- obrazac Promatrač je sastavni dio arhitektonskog obrasca model - pogled - upravljač (MVC)
- obrazac promatrač je srodan organizaciji dokument-pogled (MFC, Microsoft)
- elementi GUI biblioteka često se ponašaju kao Subjekti, dok se korisnički kod prijavljuje kao promatrač
- 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 nasljeđivanju, pretrpanim osnovnim razredima i miješanju odgovornosti



Dekorator: motivacija

Alternativni načini rješavanja:

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

class WindowScroll: public virtual Window  { /* ...*/};

class WindowStatus: public virtual Window  { /* ...*/};

class WindowScrollStatus:
  public WindowScroll, public WindowStatus  { /* ...*/};


class Box: public Window { /* ...*/};

class BoxScroll /* ??? */
```

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

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) koji implementira dodatnu odgovornost, a ostale odgovornosti prosljeđuje komponenti
- transparentnost dekoratora omogućava **rekurzivno** umetanje (pospješuje se ortogonalnost)

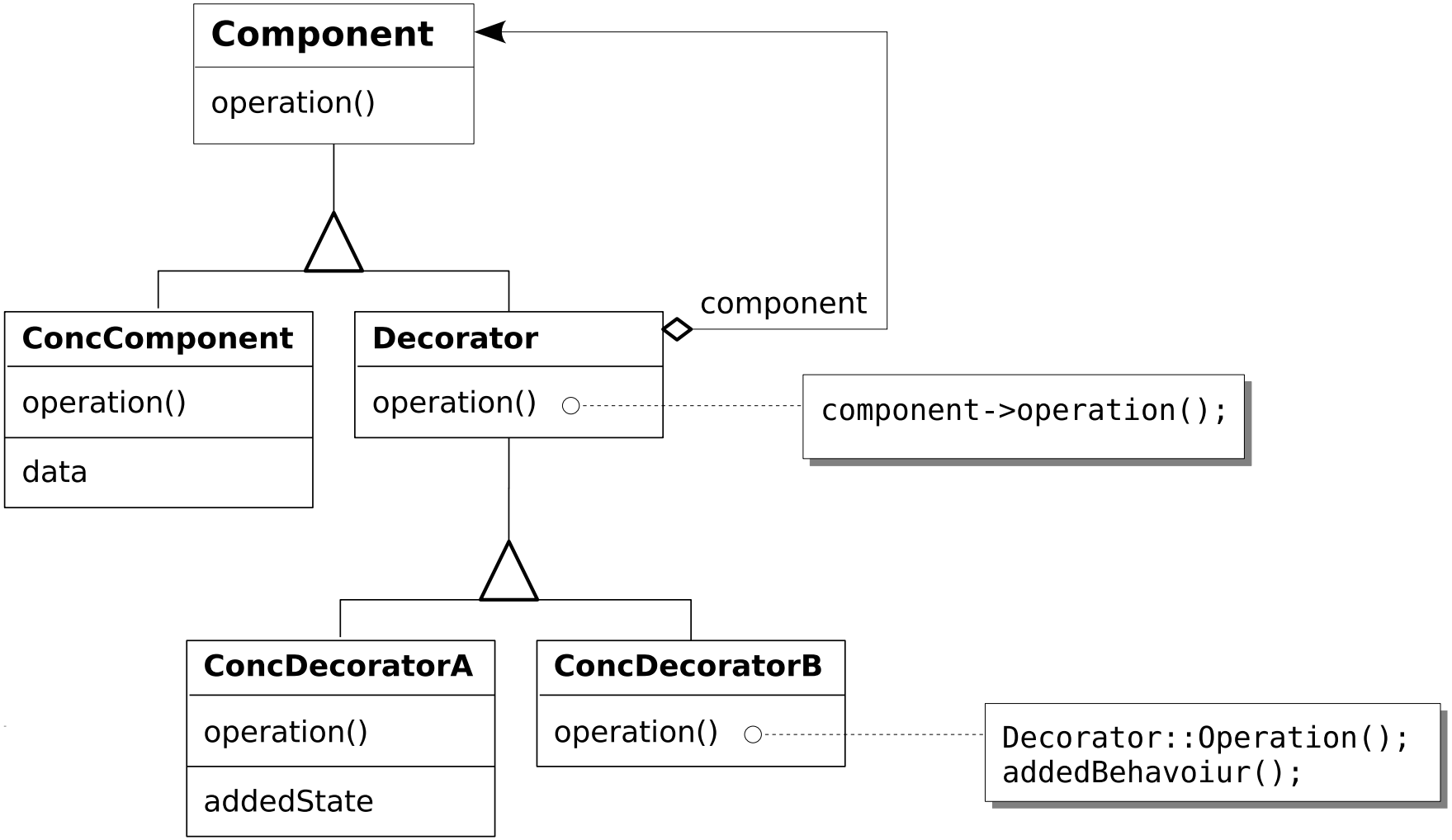
Dekorator: primjenljivost

- pojedinim objektima je potrebno pridodati odgovornosti **dinamički** i **transparentno** (bez utjecanja na druge komponente)
- odgovornosti je potrebno moći **povući**
- proširenje nasljeđivanjem 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: STRUKTURNI DIJAGRAM

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

Podatnost se ostvaruje rekurzivnom kompozicijom (handle-body)



Dekorator: sudionici

- **Komponenta** (sadržaj prozora)
  - definira sučelje za objekte kojima se odgovornosti mogu dinamički konfigurirati
- **Konkretna komponenta** (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

Dekoratori mogu pružati i dodatne usluge, neovisno o sučelju Komponente

Dekorator: posljedice

- u odnosu na nasljeđivanje:
  dinamička ortogonalna konfiguracija
- u odnosu na konfigurabilni osnovni razred:
  - **ortogonalnost** temeljne funkcionalnosti i dodatnih odgovornosti
  - aplikacija ne mora održavati svojstva koja se ne koriste
  - lako dodavanje novih dekoratora
- oprez: dekorirana komponenta nije identična originalu!
- povećana složenost, puno malih objekata

Dekorator: implementacijska pitanja

U strogo tipiziranim jezicima Dekorator mora naslijediti Komponentu

Ako imamo samo jednog konkretnog Dekoratora, apstraktno sučelje možemo izostaviti

Kako bi se izbjegle nepotrebne međuovisnosti, Komponenta mora biti što apstraktnija

Dekorator vs Strategija:

- mijenjanje vanjskog izgleda odnosno 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

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

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(c);
  }
  virtual void resize();
};

struct Window{
  VisualComponent* contents();
  void setContents(VisualComponent* c);
};

void client(){
  // experienced user on an ultra-portable
  window->setContents(new TextView);
  // show bells and whistles on a big screen
  window->setContents(new DecoratorScroll(
      new DecoratorStatus(
        window->contents())));
  // back to spartan configuration
  VisualComponent* old=window->contents();
  window->setContents(
      old->contents()->contents());
  delete old; //shallow delete!
}
```

posebna prednost ako imamo i: `class Box: public Window { /*...*/};`

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

TVORNICE: PRIMJER

Učitavanje vektorske grafike: primjer datoteke, kôd

```
# Format: ObjectType,
#   position, data, [style]
#
# Circle
C205,468 30
# Quadrangle
Q300,288 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 loadFile(istream& is,
  vector<Object*>& drawing)
{
  string str;
  while (getline(is,str)){
    Object* pObj(0);
    switch (str[0]){
    case 'C': pObj=new Circle; break;
    case 'Q': //...
    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 `loadFile` nije zatvorena za promjene!

TVORNICE: OSNOVNA IDEJA

Prekrojiti `loadFile` u skladu s načelom ortogonalnosti

Izdvojiti dodatne odgovornosti (uzroke promjene) u zasebnu komponentu

Recept za ortogonalizaciju: enkapsuliraj i izdvoji ono što se mijenja

```
void loadFile(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:

`loadFile` više **ne ovisi** o konretnim razredima (inverzija ovisnosti)

TVORNICE: RAZRADA

Osnovna ideja enkapsulacije stvaranja može se dodatno razraditi:

- virtualni poziv tvornice iz osnovnog razreda (obrazac metode tvornice)
- definirati apstraktni razred čije metode sadrže tvornice za obitelj srodnih razreda (obrazac apstraktne tvornice)

U svakoj kombinaciji, krajnji cilj je osloboditi klijente od ovisnost o konkretnim razredima!

Metoda tvornica

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

Metoda tvornica omogućava statičku konfiguraciju instancijacije

Primjer – otvaranje novog dokumenta u okviru za MDI aplikaciju:

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

class Application {
public:
  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
{
  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 tvornice jer je odgovorna za stvaranje novih korisničkih objekata

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

Metoda tvornica: primjenljivost

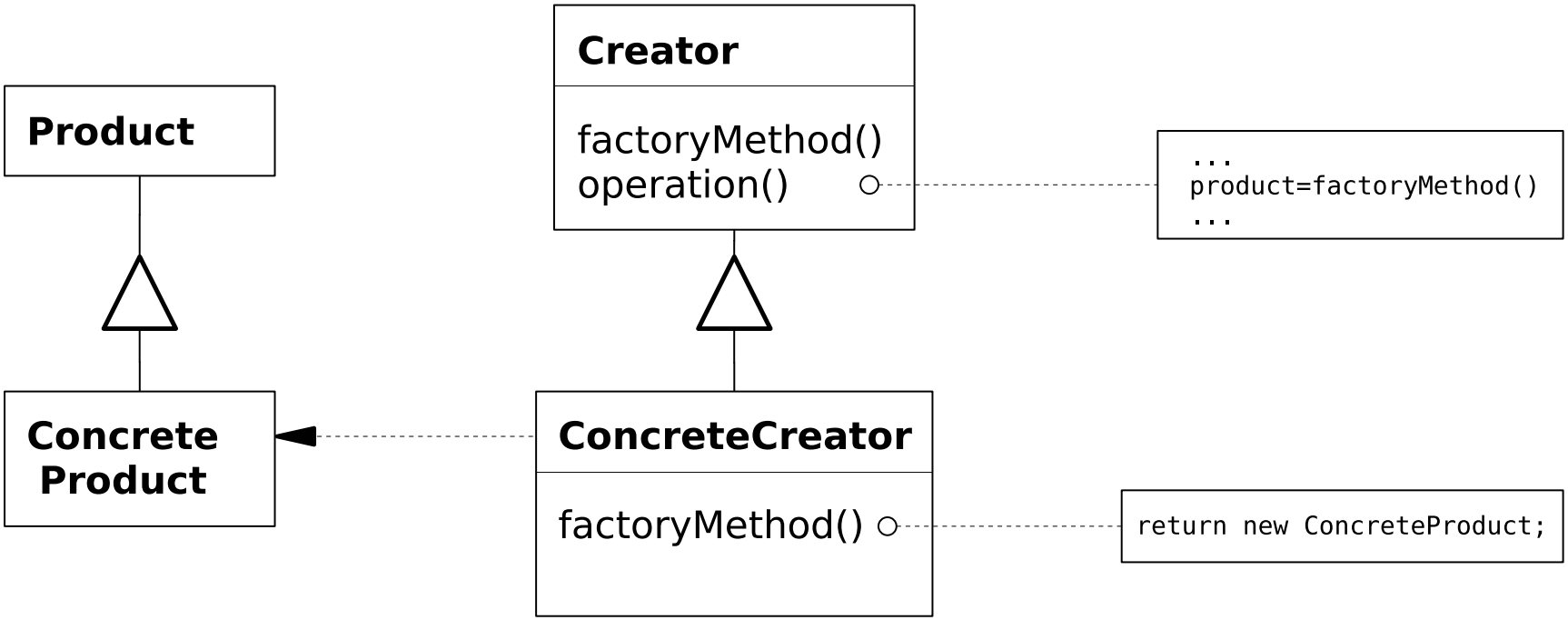
Obrazac metode tvornice (virtual constructor):

- osnovni razred instancira objekte čiji konkretni razred nije poznat
- izvedeni razred specificira konkretni tip objekta kreiranog od strane osnovnog razreda
- razred povjerava odgovornost pomoćnom objektu, a potrebno je lokalizirati znanje o konkretnom tipu pomoćnog objekta

METODA TVORNICA: STRUKTURNI DIJAGRAM

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

Podatnost 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 gdje se instancira Konkretni proizvod

Suradnja: Kreator izvedenim klasama povjerava izvedbu metode tvornice odnosno instanciranje odgovarajućeg Konkretnog proizvoda

Posljedice: pospješuje se inverzija ovisnosti u osnovnim razredima

Metoda tvornica: implementacijska pitanja

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

Metoda tvornice 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 tvornice, ali:

- apstraktna tvornica koristi delegaciju, dok metoda tvornica koristi nasljeđivanje (iako, metodu tvornicu mogu zvati i klijenti!)
- apstraktna tvornica odgovorna za instanciranje **obitelji** objekata

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{
  virtual Window* createWindow();
  virtual ScrollBar* createScrollBar();
}

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

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

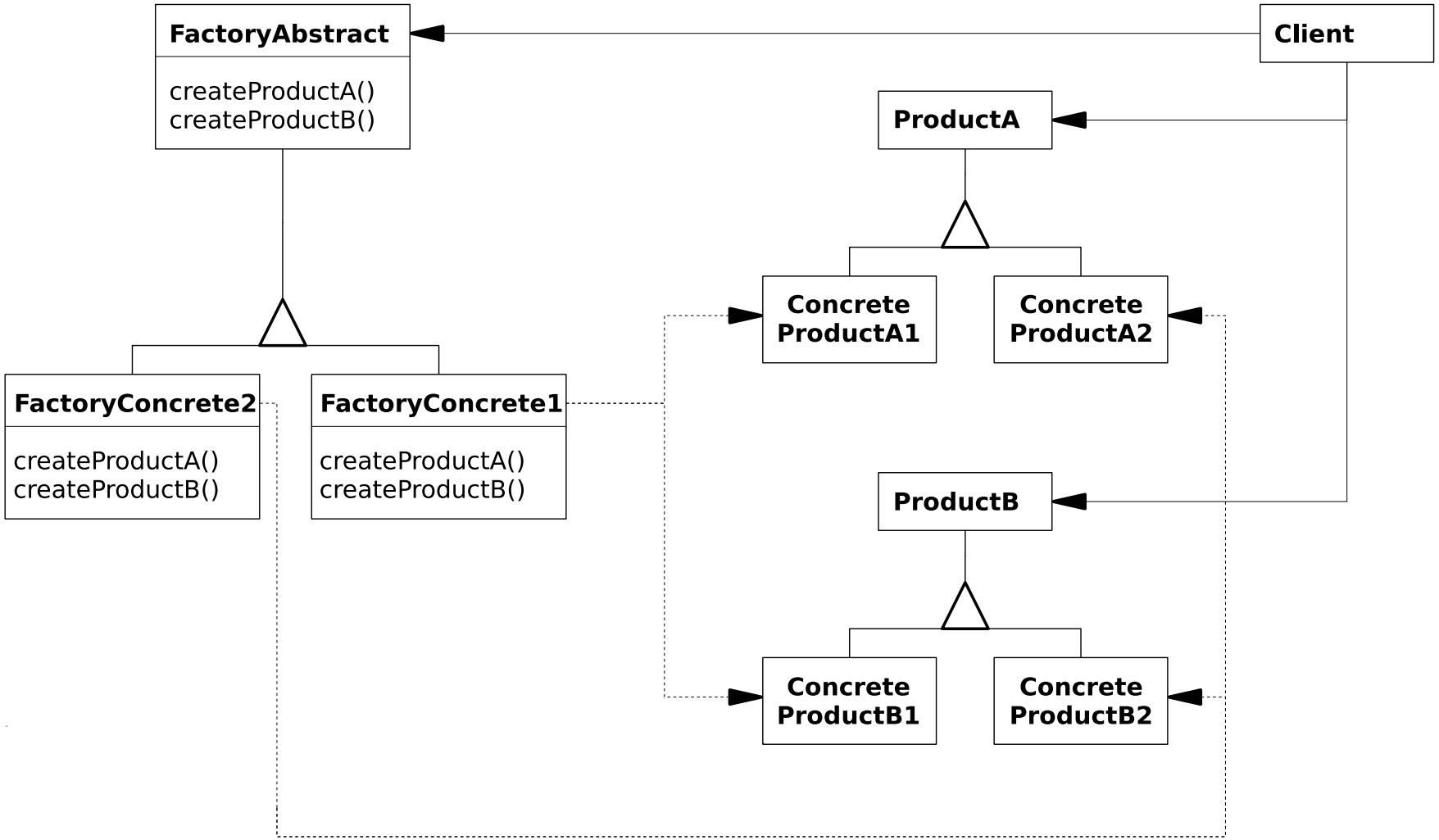
Apstraktna tvornica: primjenljivost

- 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")
- pogodno omogućiti dostup samo do sučelja (ne i implementacije) biblioteke razreda

APSTRAKTNA TVORNICA: STRUKTURNI DIJAGRAM

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

Podatnost 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 (netko ipak treba kreirati konkretnu tvornicu!)

Apstraktna tvornica: posljedice

- izoliranje klijenata od konkretnih razreda
- lako izmjenjivanje obitelji proizvoda
- pospješuje kozistenciju unutar 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 obično prema obrascu metode tvornice

Parametriziranjem tvornica postižemo lakše nadograđivanje

Opasnost od pretjeranog oblikovanja (treba li nam sva ta konfigurabilnost?)

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)
- upravljanje registrom za instanciranje razreda
- procedura s ispisnim stanjem (npr, pretvorbena tablica, svojstva sustava)

Jedinstveni objekt 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{
protected:
  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
- statičko konfiguriranje globalnog objekta bez utjecaja na klijente
- globalnu instancu je 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: POSLJEDICE

- kontrolirani pristup jedinoj instanci razreda
- nema opterećenja globalnog imenika (*namespace*)
- omogućava statičku konfiguraciju nasljeđivanjem ili povjeravanjem
- sličan pristup primjenljiv kad želimo općenitiju kontrolu instanciranja

JEDINSTVENI OBJEKT: IMPLEMENTACIJA

Prednosti dinamičkog instanciranja nad statičkim:

- eksplicitno iskazani zahtjev za jedinstvenim objektom
- ubrzavanje pokretanja programa
- dodatni kontekst (konfiguriracijska datoteka, komandna linija)
- rješavanje problema međuovisnosti jedinstvenih objekata

Usporedno izvođenje u višenitnom okruženju

- potrebno uvesti međusobno isključivanje u metodi `instance()`
- to može biti skupo (vremenski), ali nema lakih rješenja: isključivanje s dvostrukom provjerom može ne raditi!
- opcije: prihvatiti cijenu sinkronizacije, pristup preko cacheirane reference, statičko instanciranje

JEDINSTVENI OBJEKT: IMPLEMENTACIJA (2)

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

```
// MySingleton.cpp
// caveat: this code may fail subtly and miserably!
volatile MySingleton* MySingleton::pInstance_=0;

MySingleton& MySingleton::instance(){
  if (pInstance_==0){
    MutexLock l(mutex_);
    if (pInstance_==0){
      pInstance_=new 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 zbog važnih optimizacija

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

JEDINSTVENI OBJEKT: PRIMJENE

Kreacijski obrasci se često izvode kao jedinstveni objekti:
Tvornica, Prototip, Graditelj

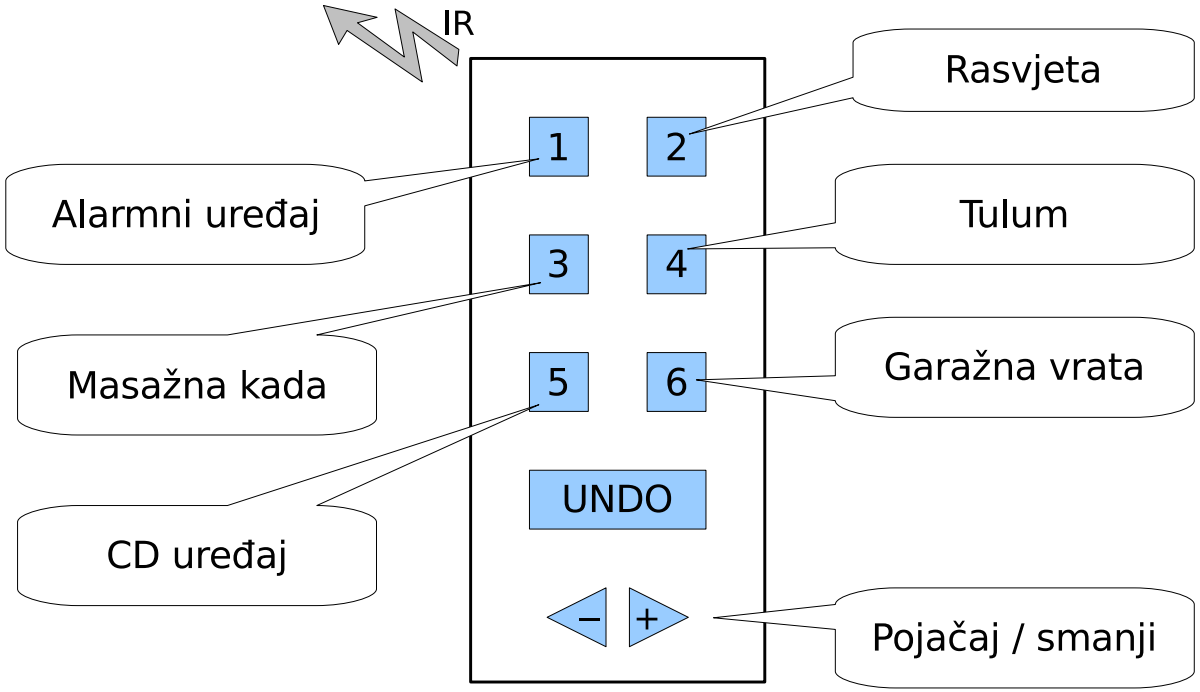
Enkapsuliranje pristupa sklopovskim i programskim resursima
(ekran, serijski port, biblioteke za pristup sklopovlju, bazama)

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

NAREDBA: NAMJERA

Omogućiti unificirano baratanje raznorodnim zahtjevima enkapsulacijom zahtjevâ u okviru polimorfnog objekta

Zgodan primjer: programska podrška za otvoreni daljinski upravljač



API-jevi za uređaje

Problem: ostvariti konfigurabilnost upravljača, tj. razdvojiti zaprimanje korisničkog zatjeva od implementacijskih detalja konkretne obrade

NAREDBA: MOTIVACIJA

Ponekad je potrebno osigurati distribuciju poslova bez da išta znamo o konkretnim operacijama koje je potreno poduzeti

```
//vendor APIs
struct ACMEHotTub{
  void circulate();
  void jetsOn();
  void jetsOff();
  void setTemp(int);
};
struct BulldogAlarm{
  void arm();
  void disarm();
};
struct BaywatchStereo{
  void on_off(bool on);
  void setCD();
  void setRadio();
  void setVolume();
};
```

```
//the remote control UI
void IRCtrl::onClick(int slot){
  curSlot_=slot;
}
void IRCtrl::onDoubleClick(int slot){
  //if (slot==1){
  //  if (! mySonyStereoOn_){
  //    mySonyStereo.onOff(true);
  //    mySonyStereo.setCD();
  //    mySonyStereo.setVolume(42);
  //  } else{
  //    mySonyStereo.onOff(false);
  //  }
  //  ...
}
void IRCtrl::onIncrease(){/* use curSlot_ */}
void IRCtrl::onDecrease(){/* use curSlot_ */}
```

Umjesto ožičene funkcionalnosti, htjeli bismo da `onDoubleClick` bude dinamički konfigurabilan

NAREDBA: RJEŠENJE

```
//configuration
void IRctrl::setCmdMgr(int slot,
    CommandManager* mgr)
{
  pCmdMgr[slot]=mgr;
}

//user interface
void IRCtrl::onClick(int slot){
  curSlot_=slot;
}

void IRCtrl::onDoubleClick(int slot){
  lastCommand_=pCmdMgr[slot]->pCmdAction();
  lastCommand_->execute();
  pCmdMgr[slot]->toggle();
}
void IRCtrl::onIncrease(){
  lastCommand_=pCmdMgr[curSlot_]->pCmdInc();
  lastCommand_->execute();
}
void IRCtrl::onDecrease(){
  lastCommand_=pCmdMgr[curSlot_]->pCmdDec();
  lastCommand_->execute();
}

void IRCtrl::onUndo(){
  lastCommand_->undo();
}
```

NAREDBA: RJEŠENJE

```
struct Command{
  virtual void execute()=0;
  virtual void undo()=0;
};

struct CommandAlarmOn{
  BulldogAlarm& myAlarm_;
  // ...
  virtual void execute(){
    myAlarm_.arm();
  }
  virtual void undo(){
    myAlarm_.disarm();
  }
};

struct CommandMacro{
  std::list<Command*> cmds_;
  virtual void execute(){
    it=cmds_.begin();
    while (it<cmds_->end()){
      it->execute(); ++it;
    }
  }
  virtual void undo(){
    it=cmds_.end();
    while (it>=cmds_->begin()){
      --it; it->undo();
    }
  }
};
```

```
struct CommandManager{
  virtual Command* pCmdAction()=0;
  virtual void toggle()=0;
  virtual Command* pCmdInc(){
    return &CommandNull::instance();}
  virtual Command* pCmdDec(){
    return &CommandNull::instance();}
};

struct CommandManagerAlarm{
  static CommandAlarmOn on_;
  static CommandAlarmOff off_;
  Command* pCmdAction_;

  virtual Command* pCmdAction(){
    return pCmdAction_;
  }
  virtual void pCmdToggle(){
    pCmdAction_= (pCmdAction_=&on_)?
      &off_: &on_;
  }
};

struct CommandManagerParty{
  static CommandMacro on_;
  static CommandMacro off_;
  // the rest same as above

  // a good idea: factoring the common
  // parts into a class template
};
```

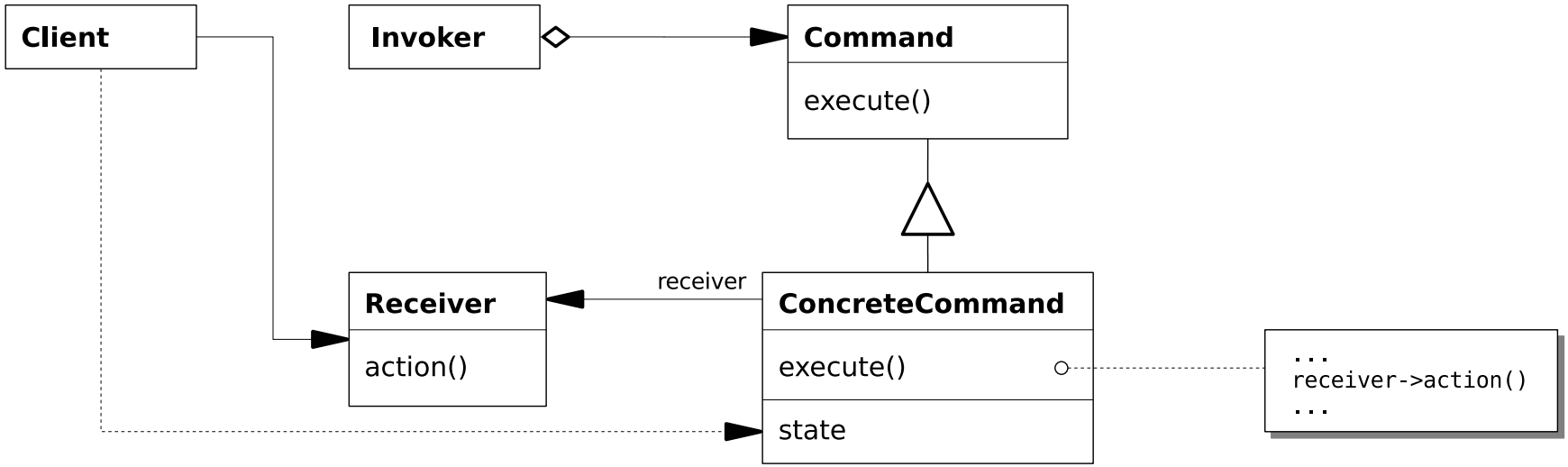
NAREDBA: PRIMJENLJIVOST

- potrebno parametrizirati objekt s procedurom koju valja obaviti (donekle slično strategiji!)
- zadavanje, raspoređivanje i izvođenje zadataka se zbiva u vremenski odvojenim intervalima
- potrebno podržati operaciju *undo*
- potrebno podržati inkrementalno bilježenje promjena za slučaj oporavka nakon kvara sustava
- potrebno podržati složene atomarne operacije (transakcije)

NAREDBA: STRUKTURNI DIJAGRAM

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

Podatnost se ostvaruje kombinacijom povjeravanja i nasljeđivanja



NAREDBA: SUDIONICI

Sudionici:

- **Naredba**:
  - deklarira jednostavno sučelje za izvođenje operacije
- **Konkretna Naredba** (CommandAlarmOn, narudžba):
  - definira vezu između Pozivatelja i Primatelja
  - implementira sučelje Naredbe pozivajući Primatelja
- **Klijent**:
  - kreira konkretnu Naredbu i postavlja njenog Primatelja
- **Pozivatelj** (IRCtrl::onDoubleClick, konobar):
  - inicira zahtjev preko sučelja Naredbe
- **Primatelj** (BulldogAlarm, kuhar):
  - zna koje operacije mora obaviti kako bi se zadovoljio zahtjev

Naredba: suradnja

- Klijent kreira konkretnu Naredbu i navodi njenog Primatelja
- 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

Naredba: posljedice

- Naredba odvaja inicijatora zahtjeva (Pozivatelja) od izvršioca (Primatelja)
- Zahtjevnije Naredbe mogu se složiti od jednostavnijih, kao u slučaju Makro-naredbe. Makro-naredbe odgovaraju obrascu Kompozicije.
- Dodavanje novih Naredbi ne zahtijeva mijenjanje postojećih razreda

Naredba: primjene

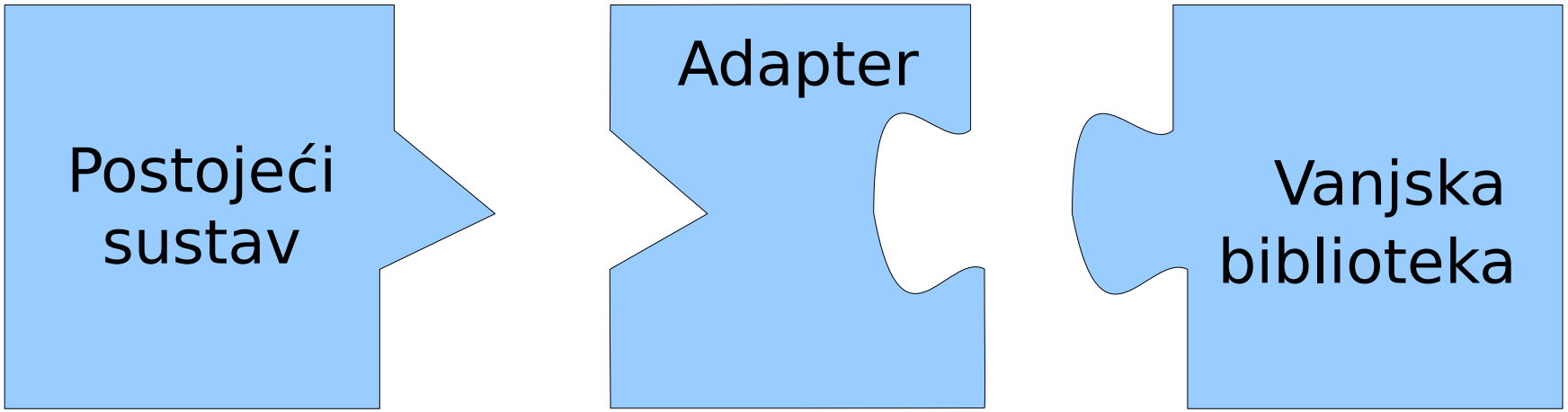
Raspoređivanje poslova u stvarnom vremenu
(npr, thread pool unutar web servera)

Bilježenje promjena u cilju omogućavanja oporavka nakon kvara sustava

Podržavanje opozivih akcija, uključujući transakcijsko poslovanje

Prilagodnik: namjera

Prilagoditi postojeći razred sučelju kojeg očekuju klijenti.



Razvoj i održavanje programa iznimno teška djelatnost:
često se više isplati pisati novi prilagodni kod, nego modificirati postojeći

Prilagodnik omogućava interoperabilnost inače nekompatibilnog kôda
$\Rightarrow$ najčešće korišteni obrazac!

Prilagodnik: motivacija

Želimo transparentno raditi s više tipova kamera (analogne, DCAM 1394, DV 1394, USB 2.0, ...)

Ideja: dizajnirati apstraktno sučelje koje odražava inherentne potrebe naše aplikacije

Za svaku pojedinu biblioteku razviti prilagodnik prema apstraktnom sučelju (čak i ako to formalno nije moguće, ponekad se isplati snaći!)

Konkretnim implementacijama pristupati polimorfno, kroz referencu na osnovni razred

Dobitci: operabilnost, bolja arhitektura glavnog programa (oblikovanje u okviru domene, ne implementacije)

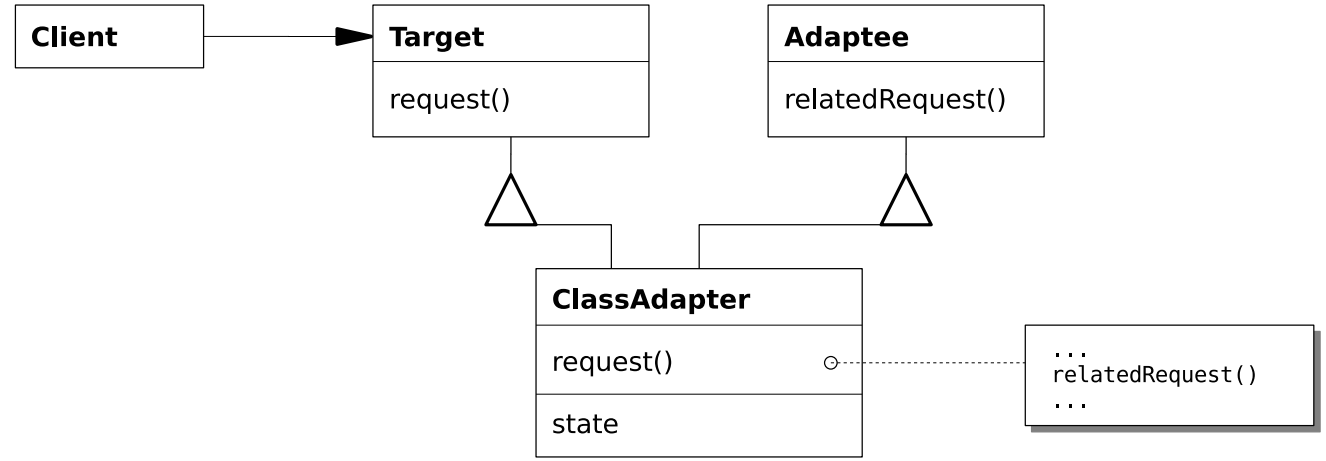
Prilagodnik: primjenljivost

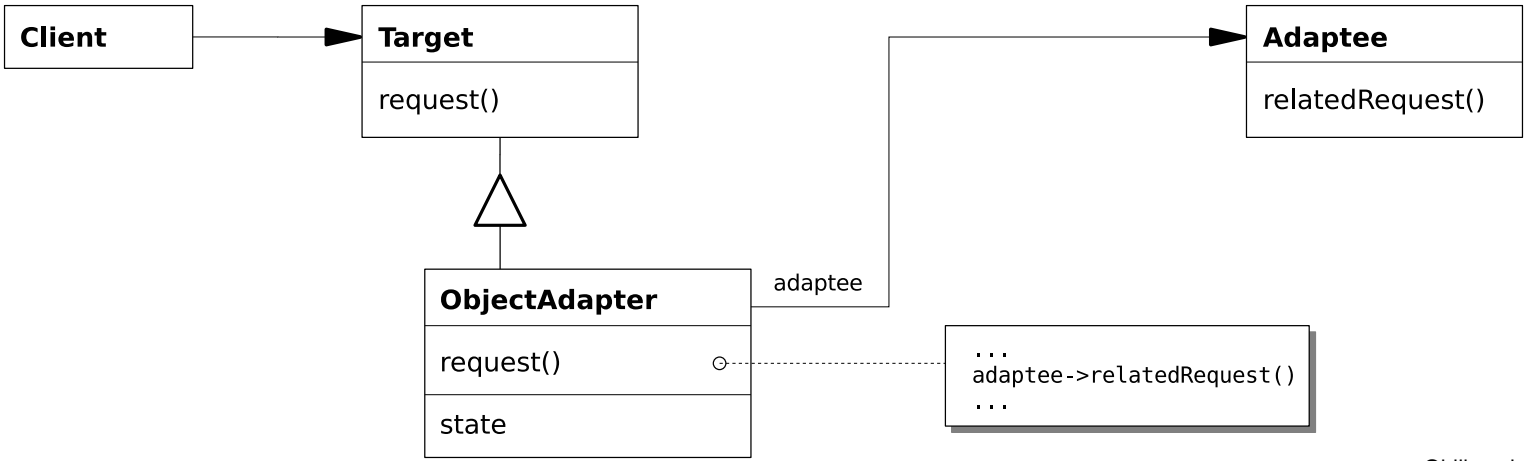
- Nekompatibilno sučelje nas sprječava u korištenju postojećeg koda
- potrebno je uniformno i transparentno pristupati raznorodnim resursima
- potrebno istovremeno prilagoditi više izvedenih razreda, a dio koji je potrebno prilagoditi se nalazi u osnovnom razredu (delegacija!)

PRILAGODNIK: STRUKTURNI DIJAGRAM

Prilagodnik (wrapper): **strukturni** obrazac; domena **objekata**, **razreda**

Podatnost se ostvaruje povjeravanjem ili nasljeđivanjem





Prilagodnik: sudionici i suradnja

Sudionici:

- **Cilj**:
  - definira sučelje specifično za domenu klijenta
- **Klijent**
  - surađuje s objektima koji implementiraju sučelje Cilja
- **Vanjski razred**:
  - implementira korisnu funkcionalnost kroz nekompatibilno sučelje
- **Prilagodnik**:
  - prilagođava sučelje Vanjskog razreda sučelju Cilja

Suradnja:

- Klijenti pozivaju sučelje Cilja, Prilagodnik pozive implementira preko poziva sučelja Vanjskog razreda

Prilagodnik: posljedice

Potpora korištenju apstraktnih sučelja (tj, inverziji ovisnosti) te boljoj razdiobi odgovornosti

Prednosti objektnog prilagodnika:

- može se primijeniti na sve razrede izvedene iz Vanjskog razreda

Prednosti razrednog prilagodnika:

- mogućnost utjecanja na Vanjski razred preko polimorfnih funkcija
- u igri je samo jedan objekt, nisu potrebne dodatne indirekcije

Prilagodnik: implementacija

Implementacija:

- Izvedba razrednog Prilagodnika:
  - javno nasljeđivanje Cilja
  - privatno nasljeđivanje Vanjskog razreda
- Izvedba objektnog Prilagodnika:
  - javno nasljeđivanje Cilja
  - povjeravanje objektu Vanjskog razreda

PRILAGODNIK: USPOREDBA

- Dekorator: poboljšana funkcionalnost uz zadržavanje sučelja
- Fasada: pojednostavljeno umjesto promijenjeno sučelje
  - struktura vrlo slična Prilagodniku, samo što se iza novog sučelja najčešće krije cijeli podsustav
  - namjera: odvajanje klijenata od izvedbenih detalja enkapsuliranog podsustava