# Oblikovni obrasci u programiranju

## Još rješenja učestalih oblikovnih problema

Siniša Šegvić

Sveučilište u Zagrebu

Fakultet elektrotehnike i računarstva

Zavod za elektroniku, mikroelektroniku
računalne i inteligentne sustave

# Tvornice: uvod

Do sada smo naučili da apstraktna sučelja mogu znatno pospješiti nadogradivost programskog sustava

Međutim, konkretne tipove ponekad ne možemo izbjeći:

- npr. stvaranje objekata podrazumijeva ovisnost o konkretnom tipu

Zbog toga najzanimljiviji trenutci u životu objektno orijentiranog programa često odgovaraju upravo stvaranju objekata

Kreacijski obrasci: organizacija komponenata koje stvaraju objekte

```
class Concrete : public AbstractBase{
  // ...
};

int MyComponent::method(){
  // creation implies dependence on a concrete class
  Concrete* pc= new Concrete;
}
```

Važan specijalni slučaj: učitavanje dokumenta s polimorfnim objektima

# 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

# Circle,
#   area style: red fill,
#   line width: .01cm
C311,222 10 : ASF0xff0000 LW0.01cm
# Polygon, named style
P311,222 ... : UMLClass
# Text, named style
T315,238 'MyClass' : UMLClass
```

```
void loadDrawing(std::istream& is,
  std::vector<Object*>& drawing)
{
  std::string str;
  while (std::getline(is,str)){
    // skip comments and empty lines
    // ...

    Object* pObj(0);
    switch (str[0]){
    case 'C': pObj=new Circle; break;
    case 'S': //...
    //...
    }
    pObj->load(str.substr(1));
    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]));
    pObj->load(str.substr(1));
    drawing.push_back(pObj);
  }
}
```

Funkciju `createObject` nazivamo parametriziranom tvornicom

Parametrizirana tvornica: osnovni kreacijski obrazac

- **namjera**: lokalizirati odgovornosti za stvaranje objekata
- `loadDrawing` više ne ovisi **izravno** o konkretnim razredima

# Tvornice: razrada

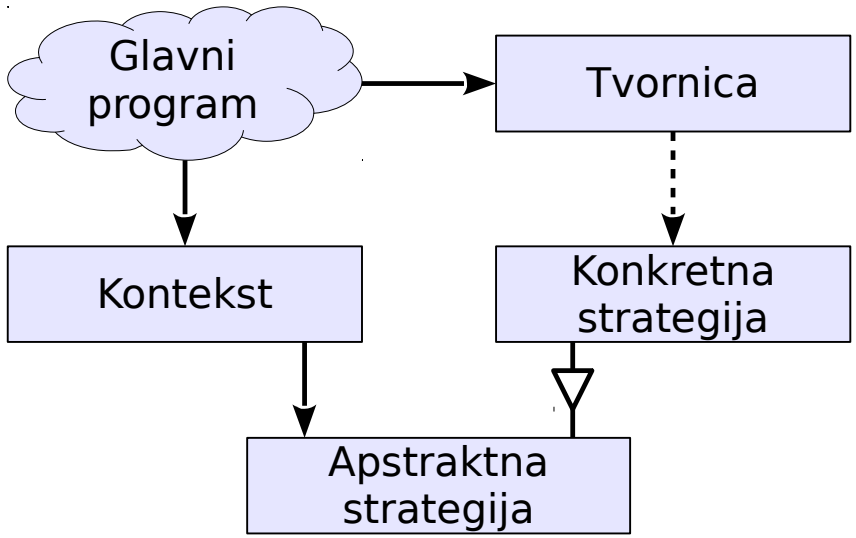
Osnovnu ideju parametrizirane tvornice razrađuju kreacijski obrasci:

- metoda tvornica: prepustiti implementaciju kreacijske metode klijentima
- apstraktna tvornica: povjeriti stvaranje objekata većeg broja povezanih razreda polimorfnom objektu
- jedinstveni objekt: onemogućiti stvaranje više od jednog primjerka
- prototip: stvarati objekte iz unaprijed pripremljenih prototipova
- ...

Ti obrasci na različite načine **umanjuju** ovisnost o konkretnim razredima, ali niti jedan od njih tu ovisnost ne otklanja u potpunosti

# Tvornice: problem ovisnosti

Npr. u Strategiji vršna komponenta ovisi o konkretnim objektima:




U takvoj organizaciji, nove strategije ne možemo dodavati bez mijenjanja postojećeg koda!

To može biti problem, mi bismo htjeli **nadogradnju bez promjene**:

1. da dodavanje novih proizvoda zahtijeva što manje intervencija
2. da korisnici mogu pisati proširenja (npr. flash plug-in za browsere)

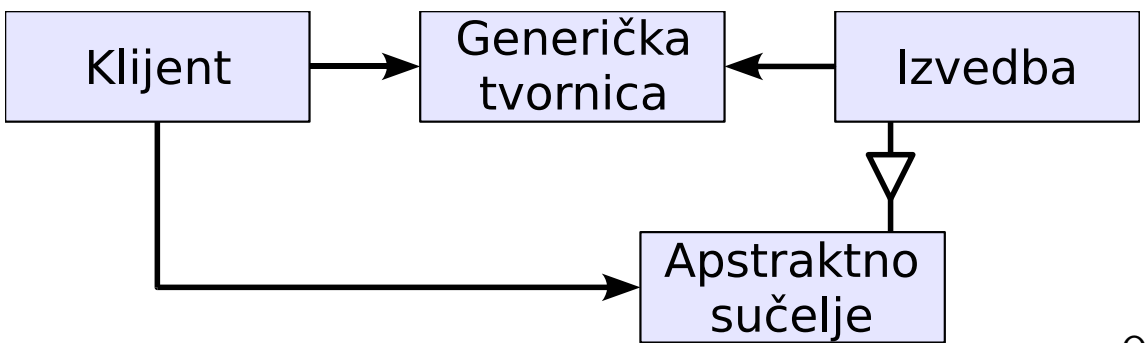
Možemo li ukloniti ovisnost tvornice o konkretnim razredima?

# Tvornice: generička tvornica

Kako izvesti generičku tvornicu (tvornicu koja ne pozna proizvode)?

1. učitati izvedbe metoda iz zadane dinamičke **biblioteke** (`.dll`, `.so`)
   - UNIX: `dlopen`, `dlsym`
   - Windows: `LoadLibrary`, `GetProcAddress`
2. koristiti **introspekciju** preko simboličkih imena razreda i paketa:
   - Java: `Class.forName(strClasName)`
   - Python: `imp.load_module(strModuleName)`, `getattr(module, strName)`
3. registrirati konstrukcijsku funkciju iz statičkog incijalizatora (C++)
   - ideja: statički incijalizator poziva generičku tvornicu
   - argumenti: konstrukcijska funkcija, simboličko ime (string)

Dijagram ovisnosti generičke tvornice (detalji u **lab. vježbama**):



# 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)
- različite vrste tvornica
- procedura s ispisnim stanjem (npr. pretvorbena tablica, svojstva sustava)

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

# Jedinstveni objekt: motivacija

Rješenje je jednostavno, osim ako moramo razmatrati konkurentnost!

```
//MySingleton.hpp
class MySingleton{
private:
  MySingleton();
  // make destructor, copy ctor
  // and assignment also private
public:
  static MySingleton& instance();
public:
  //interface
private:
  static MySingleton* pInst_;
};
```

```
//MySingleton.cpp

// initialized at the time of loading
MySingleton* MySingleton::pInst_=0;

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

**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: posljedice

- kontrolirani pristup jedinoj instanci razreda
- nema opterećenja globalnog prostora imena (*namespace*)
- ako je potrebno, konkretni tip jedinstvenog objekta može se odabrati tvornicom
- sličan pristup primjenljiv kad želimo općenitiju kontrolu instanciranja
- jedinstveni objekt je globalna varijabla:
  - povećava međuovisnost i smanjuje modularnost
  - stoga ga treba koristiti s mjerom

# Jedinstveni objekt: implementacija

Prednosti eksplicitnog pristupa globalnom objektu:

- rješavanje problema međuovisnosti globalnih objekata
- mogućnost dinamičkog instanciranja

Prednosti dinamičkog instanciranja nad statičkim:

- ubrzavanje pokretanja programa
- dodatni kontekst (konfiguracijska 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::pInst_=0;
MySingleton& MySingleton::instance(){
  MutexLock l(mutex_);
  if (pInst_==0){
    pInst_=new MySingleton;
  }
  return *pInst_;
}
```

Sve donedavno nije bilo lakih rješenja, a opcije su bile:

- prihvatiti cijenu sinkronizacije
- pristupati preko cacheirane reference
- prihvatiti cijenu (i probleme) 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 (pInst_==0){
    MutexLock l(mutex_);
    if (pInst_==0){
      pInst_=new MySingleton;
      // compiler generates:
      // (i)  pInst_=operator new(sizeof(MySingleton));
      // (ii) new (pInst_) MySingleton;
    }
  }
  return *pInst_;
}
```

Problemi riješeni u novijim verzijama jezika:

- Java 1.5 nudi poboljšani memorijski model
- C++0x uvodi predložak `std::atomic`, te atomarnu inicijalizaciju lokalnih statičkih varijabli (C++0x, 6.7.4)

http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
http://stackoverflow.com/questions/6008715/double-checked-locking-pattern-solved-in-c0x
http://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/

# Jedinstveni objekt: implementacija (4)

Ako nam je pozivanje destruktora Jedinstvenog objekta važno:

- instancu možemo deklarirati kao lokalnu statičku varijablu (prije C++0x ovo nije bilo thread-safe)

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

- instancu možemo pobrisati iz destruktora statičkog člana

```
class MySingleton {
  //...
private:
  struct MyDestructor{
    ~MyDestructor(){
      delete MySingleton::pInst_;
    }
  };
  friend struct MyDestructor;
  static MyDestructor dtor_;
};
// MySingleton.cpp
MySingleton* MySingleton::pInst_=0;
MySingleton::MyDestructor MySingleton::dtor_;
```

# Jedinstveni objekt: implementacija (5)

Varijanta Jedinstvenog objekta: Objekt s dijeljenim stanjem

- možemo imati više instanci, ali sve one dijele jedno stanje
- stanje izvedeno kao podatkovni član razreda (engl. Monostate)

```
public class Monostate{
  private static int state = 0;
  public void setX(int x){ state = x; }
  public int getX(){ return state; }
}
```

Ovo je posebno popularno u Pythonu (koristi se naziv Borg):

- rječnik atributa instance preusmjeravamo na atribut razreda!
- pristupi atributima instance svode se na pristup atributu razreda

```
class Config:
  __state = {}

  def __init__(self):
    self.__dict__ = self.__state
```

```
conf = Config()
conf.myval="Hello!"

conf2 = Config()
print(conf2.myval) # Hello!
```

# Jedinstveni objekt: primjene

Razredi kreacijskih obrazaca često se 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
(pretvorbena tablica, svojstva sustava)

# Iterator: namjera

Omogućiti uniformni slijedni obilazak kolekcije bez otkrivanja njene interne strukture

Nešto poput indeksiranja **brojačem**, ali općenitije:

- podržane proizvoljne kolekcije, ne samo polja

Obilazak iteratorom je sličan obilasku **pokazivačem**:

- iterator nudi slijedni obilazak svih kolekcija,
- iterator nudi pristup tekućem elementu

Prednost u odnosu na pokazivač:

- uniforman obilazak različitih kolekcija (stabla, rječnici, gomile, ...)
- enkapsuliran pristup elementu (sigurnije zbog ugrađenih testova)
- modularna organizacija bez potrebe za kopiranjem elemenata
- iterator je teže koristiti krivo.

# Iterator: motivacijski primjer (Java)

Polimorfni iteratori u Javi:

```
Map<String, Integer> mymap = new HashMap<String, Integer>();
mymap.put("burek", 11);

// explicit form:
Iterator<Map.Entry<String,Integer>> it = mymap.entrySet().iterator();
while (it.hasNext()) {
  Map.Entry<String,Integer> item = it.next();
  System.out.println(item.getKey() + " " + item.getValue());
}

// implicit form:
for (Map.Entry<String, Integer> item : mymap.entrySet()) {
  System.out.println(item.getKey() + " " + item.getValue());
}
```

# Iterator: motivacijski primjer (C++)

Generički iteratori jedan od tri glavna koncepta STL-a:
**korektnost** u odnosu na **tipove** i **konstantnost** objekata

```
typedef std::map<std::string, int> MyMap;
MyMap mymap;
mymap.insert(std::make_pair("burek",11));

// explicit form:
for (MyMap::const_iterator it = mymap.begin();
     it != mymap.end(); ++it)
{
  std::cout <<it->first <<" " <<it->second <<"\n";
}

// implicit form (g++ -std=c++11)
for (auto kvp : mymap){
  std::cout <<kvp.first <<" " <<kvp.second <<"\n";
}
```

# Iterator: motivacijski primjer (Python)

Implicitni iteratori u Pythonu:

```
mymap={'burek':11,'ćevapi':25}

# mi kažemo:
for key in mymap:
  print(key, mymap[key])
```

Iteratore možemo eksplicitno stvarati i pozivati:

```
# zapravo, izvodi se sljedeće:
it = iter(mymap)              # Python 3: poziva se mymap.__iter__()
try:
  while True:
    key=next(it)              # Python 3: poziva se it.__next__()
    print(key, mymap[key])
# it is easier to ask forgiveness than to ask permission...
except StopIteration:
  pass
```

Nešto efikasniji način:

```
# samo jedan pristup rječniku u svakom prolazu:
for key,val in mymap.items():
  print(key, val)
```

# Iterator: primjenljivost

Osnovne namjene iteratora:

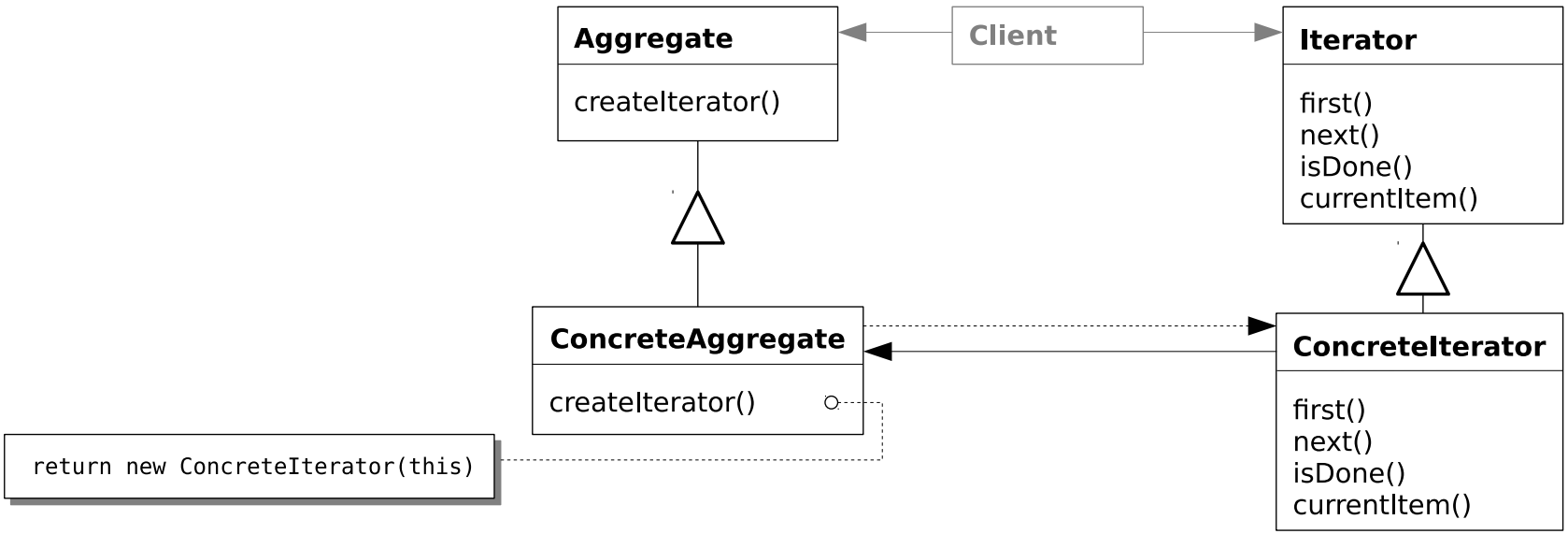
- enkapsuliran i uniforman pristup elementima kolekcije
- polimorfni obilazak kolekcije
  - klijent apstrahira vrstu kolekcije i način obilaska
- usporedni obilasci kolekcije

Detalji:

- prednosti se očituju i ako nema potrebe za polimorfnim pristupom (jasnija upotreba)
- u različitim jezicima koriste se različite varijante polimorfizma (C++: statički, Java: dinamički)

# Iterator: strukturni dijagram

Iterator (Cursor): **ponašajni** obrazac (iterator referencira kolekciju)



# Iterator: sudionici i suradnja

**Sudionici:**

- **Iterator**
  - deklarira sučelje za pristupanje i slijedni pristup elementima
- **Konkretni iterator**
  - implementira apstraktno sučelje, evidentira tekući položaj
- **Apstraktni skupni objekt**
  - deklarira sučelje za kreiranje iteratora
- **Konkretni skupni objekt**
  - izvodi apstraktno sučelje, stvara odgovarajući konkretni iterator

**Suradnja:**

- Konkretni iterator omogućava pristup tekućem elementu, te prijelaz na sljedeći element Skupnog objekta

# Iterator: posljedice

- Podržavaju se varijacije prolaska kroz skupni objekt
  (npr, prolaz kroz stablo može biti ``u dubinu'', ``u širinu'', ...)
- pospješuje se jedinstvena odgovornost:
  skupni objekt ne definira sučelje za slijedni pristup elementima
- omogućavaju se višestruki usporedni prolazi kroz skupni objekt
- uska povezanost Konkretnog iteratora i Konkretnog skupnog objekta
  - iznimno, cirkularna ovisnost je prihvatljivo rješenje
  - kôd konkretnog iteratora fizički možemo smjestiti u komponentu skupnog objekta

# Iterator: Implementacija

Varijanta iteratora: Kursor (engl. Cursor)

- prolaz definiran u skupnom objektu, iterator enkapsulira poziciju
- bolja enkapsulacija skupnog objekta
- teško definirati višestruke algoritme prolaza (u širinu, u dubinu)

Operacije nad spremnikom mogu poništiti valjanost iteratora:
npr, `erase`, `push_back` (može uzrokovati realokaciju spremnika!)

Zbog toga ponekad radimo čudne stvari: kopiramo spremnik, iteriramo unatrag itd.

# Iterator: Implementacija (2)

Iteratori s dinamičkim polimorfizmom: veći otisak, sporiji dohvat

- najčešće nije bitno, ali može biti
- C++ ne koristi polimorfne iteratore

U C++-u, Iteratori se koriste za parametrizaciju algoritama (!)

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class StrictWeakOrdering>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering comp);
```

Iteratori s različitim mogućnostima, npr za C++:

- `Forward` (npr. `std::ostream_iterator`, za `std::copy`),
- `Bidirectional` (npr. `std::list`, za `std::list<T>::sort`) i
- `Random Access` (npr. `std::vector`, za `std::binary_search`)

# Iterator: kraj obilaska

Što se događa ako iteriramo nakon kraja kolekcije?

- C++ nema ni provjere ni iznimke: nedefinirano ponašanje
- Java i Python bacaju `NoSuchElementException` i `StopIteration`

Java iznimke koristi samo iznimno (preporuča se testirati `hasNext`)

U Pythonu, kraj kolekcije **uvijek** se signalizira iznimkom `StopIteration`

- iteratori Pythona nemaju metodu `hasNext`
  - jednostavnost i efikasnost (jedan poziv umjesto dva)
  - mogućnost oblikovanja iteratora koji ne znaju jesu li došli do kraja prije nego što klijent zatraži element iza posljednjeg.
- ovakvi iteratori mogu modelirati procese koji **generiraju** podatke (npr. čitaju ih iz socketa) pa ih u Pythonu nazivamo generatorima

Tri pristupa, svaki ima prednosti i nedostatke

# Iterator: modularnost bez kopiranja (1)

Razmotrimo sljedeću implementaciju varijance:

```
#!/usr/bin/python3
def variance(L):
  # sum uses the iterator protocol
  mean=sum(L)/len(L)
  return sum(sqrdev(L,mean))/len(L)

def sqrdev(L, mean):
  L2=[]
  for x in L:
    L2.append((x-mean)**2)
  return L2
```

Nedostatak: `sqrdev` stvara novu listu koja nepotrebno troši memoriju

**Ideja**: umjesto stvaranja privremene liste --- iterirati po originalnoj listi!

- iz funkcije `sqrdev` vratiti iterator `g` koji **dekorira** iterator liste `L`
- `g` podržava standardni obilazak $\Rightarrow$ možemo ga dati funkciji **`sum`**
- funkcija **`sum`** obilazi `g`, elementi liste `L` dohvaćaju se na zahtjev

# Iterator: modularnost bez kopiranja (2)

Bolju implementaciju dobivamo dekoratorom `SqrdevDecorator`:

- metoda `__next__` računa `(x_i - mean )**2` **na zahtjev**
  - ovo je primjer tehnike lijenog računanja (lazy evaluation)
- metoda `__iter__` omogućava standardni obilazak

```
class SqrdevDecorator:
  def __init__(self, L, mean):
    self.it = L.__iter__()
    self.mean = mean
  def __next__(self):
    return (next(self.it)-self.mean)**2
  def __iter__(self):
    return self
```

```
# variance2 decorates the iterator over L
# to avoid creating a temporary list
def variance2(L):
  mean=sum(L)/len(L)
  g=SqrdevDecorator(L,mean)
  return sum(g)/len(L)
# can we make SqrdevDecorator less verbose?
```

# Iterator: generatori (1)

Python predviđa dva načina za jasnije i konciznije dekoriranje iteratora: generatorske funkcije i generatorske izraze

Generatorske funkcije temelje se na naredbi `yield`
(`SqrdevGenFun` i `SqrdevDecorator` stvaraju ekvivalentne objekte)

```
def SqrdevGenFun(L, mean):
  for x in L:
    yield (x-mean)**2

def variance3(L):
  mean=sum(L)/len(L)
  g=SqrdevGenFun(L,mean)
  return sum(g)/len(L)
```

Generatorske funkcije mogu i *generirati* podatke:

```
def fibonacci(n):
  a,b,count = 0,1,0
  while count < n:
    a,b,count = b,a+b,count+1
    yield a
for x in fibonacci(10):
  print(x)
```

# Iterator: generatori (2)

Nekad se možemo još sažetije izraziti generatorskim izrazom:

```
def variance4(L):
  mean=sum(L)/len(L)
  return sum((x-mean)**2 for x in L)/len(L)
```

Generatorski izrazi i funkcije vraćaju generatore (koji su i iteratori):

```
>>> L=[1,2,1]; g=(2*x for x in L)
>>> type(g)
<class 'generator'>
>>> next(g), next(g), next(g)
(2, 4, 2)
>>> next(g)
... StopIteration
```

Generatori su specijalna vrsta iteratora koji:

- modeliraju virtualne kolekcije koje ne zauzimaju memoriju i mogu biti beskonačne
- najčešće izražavaju modificiranje elemenata dekoriranog slijeda
- mogu "stvarati" podatke (npr. očitavanjem senzora)

# Iterator: primjene i srodni obrasci

**Primjene:**

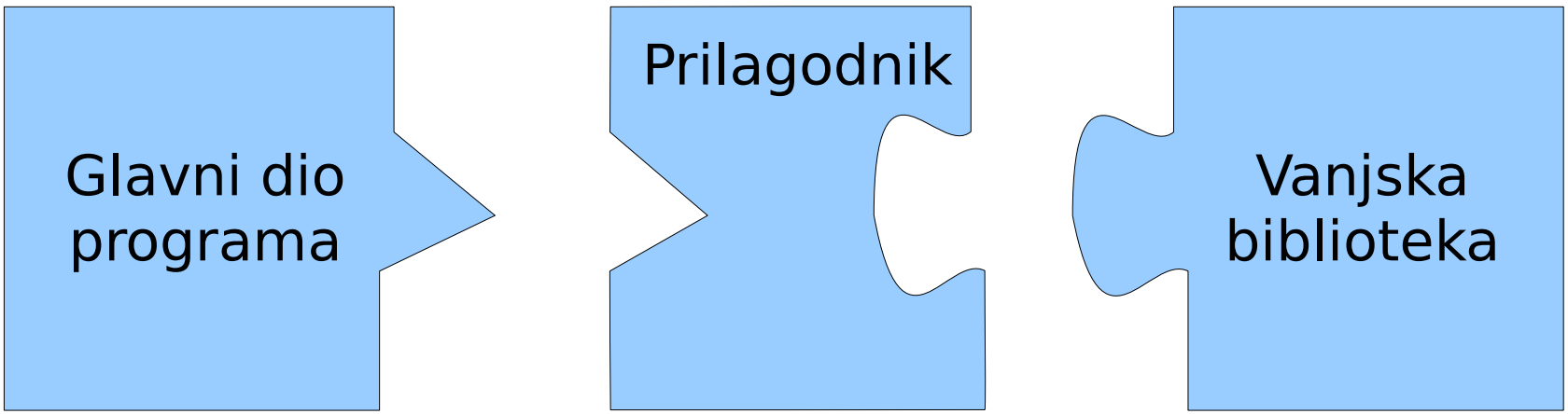
- Eksplicitni iteratori raširени u modernim bibliotekama (STL, Java, Python)
- Mnogi jezici podržavaju implicitnu iteraciju (Smalltalk, Python, Java, C++, Perl)

**Srodni obrasci:**

- Polimorfni iteratori se obično kreiraju u metodi tvornici konkretnog skupnog objekta
- Kompozit može omogućiti obilazak elemenata prikladnim iteratorom

# Prilagodnik: namjera

Prilagoditi postojeći razred sučelju kojeg očekuju klijenti.



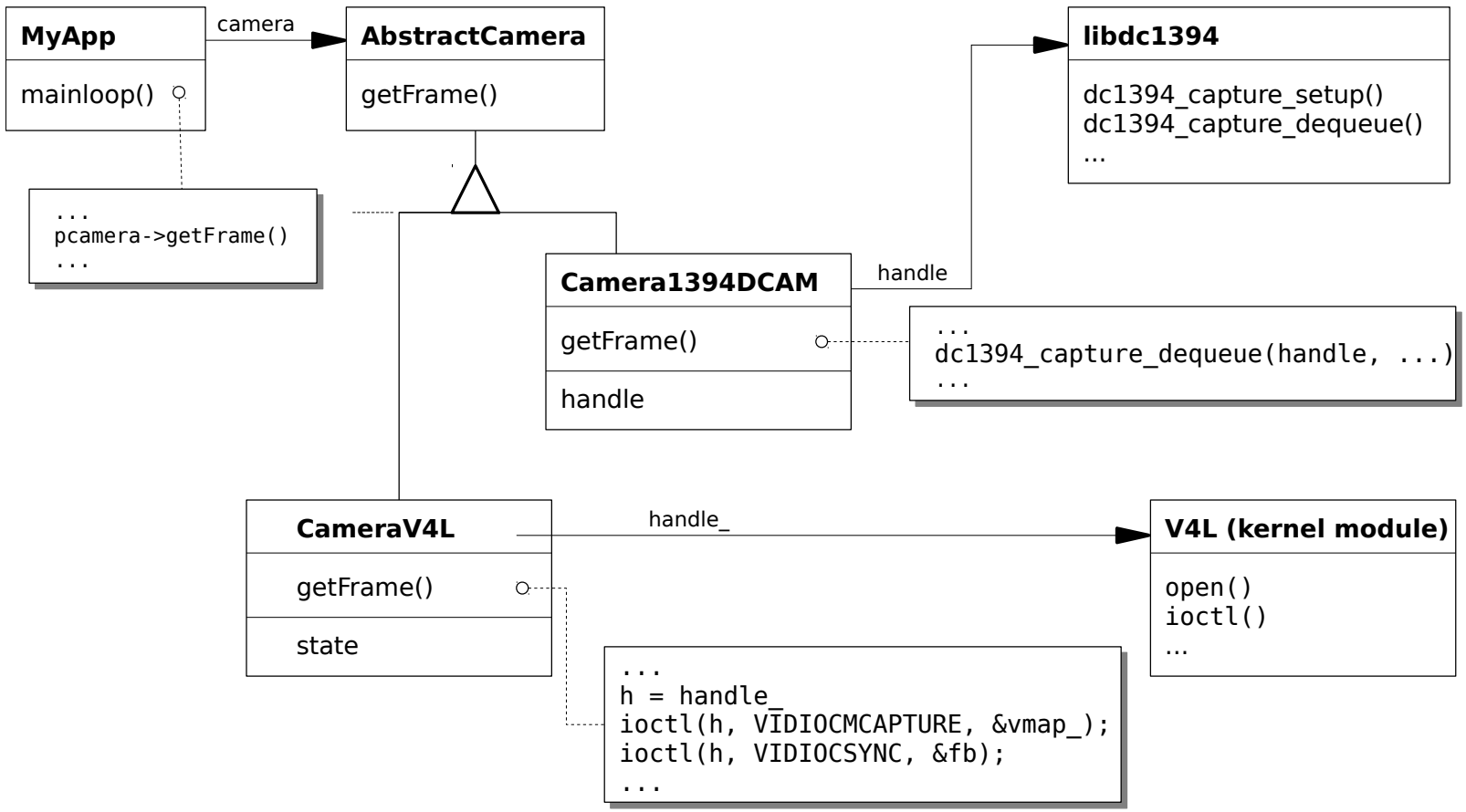

Razvoj i održavanje programa zahtijeva vremena:
često se više isplati pisati novi prilagodni kod, nego modificirati postojeći

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

# Prilagodnik: motivacija

Pretp: naš program treba **transparentno** raditi s više tipova kamera

- analogne, DCAM 1394, DV 1394, USB 2.0, ...
- svakom tipu kamere pristupamo preko odgovarajuće biblioteke



# Prilagodnik: namjera

Glavna **ideja** obrasca:

- oblikovati (apstraktno) sučelje S koje odražava inherentne potrebe naše aplikacije
- za svaku pojedinu biblioteku razviti prilagodnik prema S
- konkretnim implementacijama sad možemo pristupati polimorfno, preko reference na S (obrazac strategija!)

**Dobitci**:

- transparentna interoperabilnost
- prilagođeno sučelje otežava neželjene obrasce korištenja vanjskih komponenata
- klijenti oblikuju u okviru domene, ne implementacije

# Prilagodnik: primjenljivost

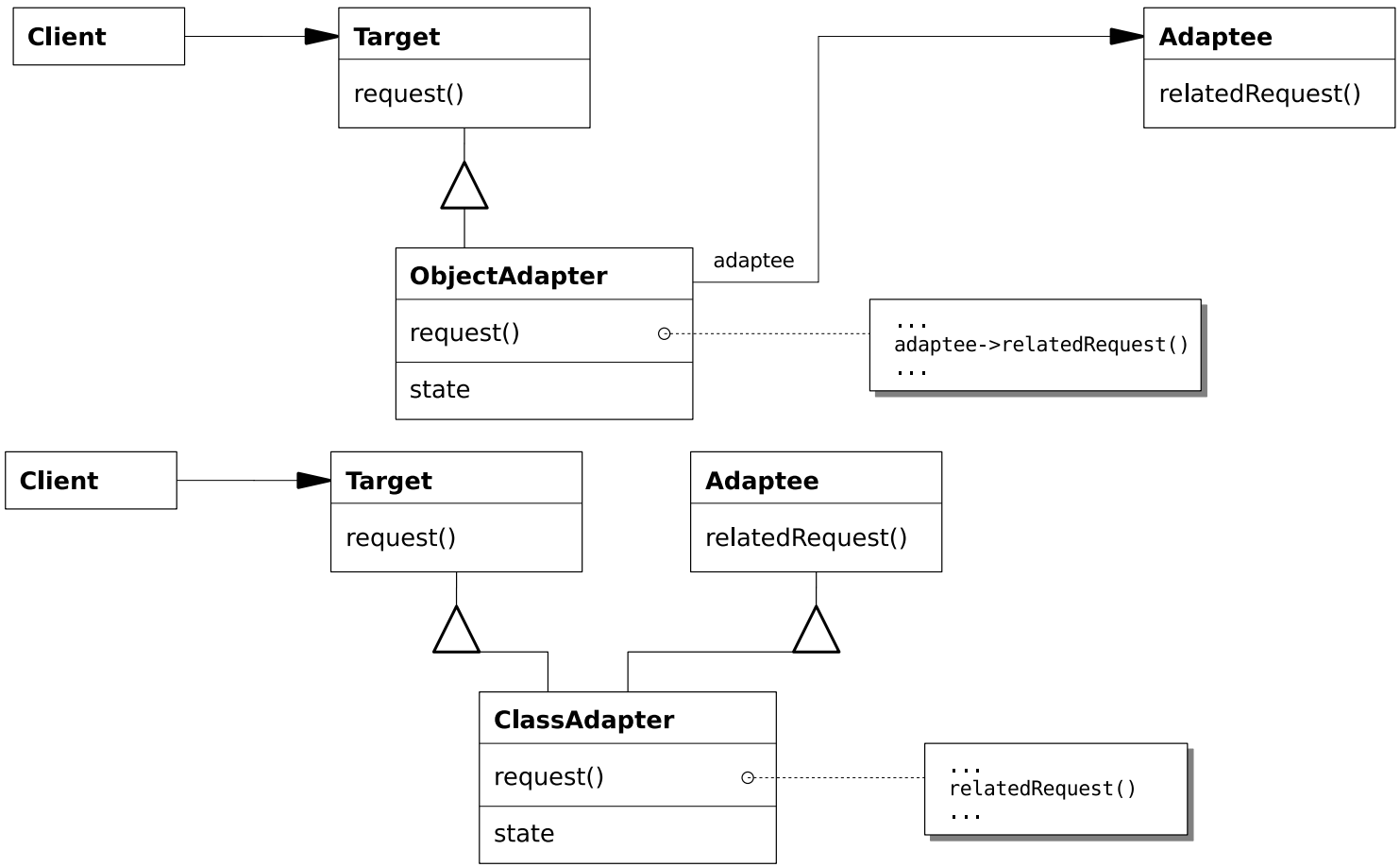
Prilagodnike koristimo ako vrijedi bilo što od sljedećeg:

- nekompatibilno sučelje nas sprječava u korištenju postojećeg koda
- potrebno je uniformno i transparentno pristupati raznorodnim resursima
- potrebno istovremeno prilagoditi više izvedenih razreda, a sučelje koje treba prilagoditi se nalazi u osnovnom razredu

# Prilagodnik: strukturni dijagram

Prilagodnik (adapter, wrapper):

- prilagodnik **umata** vanjsku komponentu → **strukturni** obrazac
- prilagodljivost se ostvaruje povjeravanjem (ili nasljeđivanjem)



# Prilagodnik: sudionici i suradnja

**Sudionici:**

- **Ciljno sučelje**
  - definira sučelje specifično za domenu klijenta
- **Klijent**
  - surađuje s objektima koji implementiraju Ciljno sučelje
- **Vanjska komponenta** (ili razred):
  - implementira korisnu funkcionalnost kroz nekompatibilno sučelje
- **Prilagodnik**:
  - prilagođava sučelje Vanjske komponente Ciljnom sučelju

**Suradnja**:

- Klijenti pozivaju metode Ciljnog sučelja, Prilagodnik pozive implementira preko poziva sučelja Vanjske komponente

# Prilagodnik: posljedice

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

Prednosti objektnog prilagodnika:

- jedino rješenje kad je vanjska funkcionalnost pisana u C-u
- može se primijeniti i na objekte razreda izvedenih iz Vanjskog razreda

Prednosti razrednog prilagodnika:

- mogućnost utjecanja na Vanjski razred nadomjesnim polimorfnim funkcijama
- rukujemo samo jednim objektom, nisu potrebne dodatne indirekcije

# Prilagodnik: implementacija

**Implementacija**:

- Izvedba razrednog Prilagodnika:
  - javno nasljeđivanje Ciljnog sučelja
  - privatno nasljeđivanje Vanjskog razreda
- Izvedba objektnog Prilagodnika:
  - javno nasljeđivanje Ciljnog sučelja
  - povjeravanje objektu Vanjskog razreda
- Dodatne mogućnosti objektnog prilagodnika u dinamičkim jezicima:
  - umatanje proizvoljnih vanjskih komponenata
  - dinamički odabir metoda vanjske komponente
  - automatsko prosljeđivanje poziva metodama vanjskog razreda

# Prilagodnik: izvorni kôd (1)

Razredni prilagodnik u Pythonu izveden prema osnovnom receptu:

- prilagodnik `PoliceDog` prilagođava vanjski razred `Dog` ciljnom razredu `PoliceOfficer`;
- prilagođeni objekti mogu biti slani i klijentu `report` ciljnog razreda `PoliceOfficer` i klijentima vanjskog razreda `Dog`

```
#!/usr/bin/python3

class PoliceOfficer:
  def __init__(self, name):
    self.name=name
  def greet(self):
    return "Zdravo!"

def report(officer):
  print("Policajac", officer.name,
    "kaže:", officer.greet())



e1=PoliceOfficer('Nikica')
report(e1) # works
```

```
class Dog:
  def __init__(self, name):
    self.name=name
  def bark(self):
    return "Vau!"

# report(Dog('Khan')) does not work!

class PoliceDog(PoliceOfficer, Dog):
  def __init__(self, name):
    self.name=name
  def greet(self):
    return self.bark()

d1=PoliceDog('Khan')
report(d1) # works!
```

# Prilagodnik: izvorni kôd (2)

Objektni prilagodnik koji koristi mogućnosti dinamičkog jezika:

- prilagodnik `PoliceAdapter` radi s različitim vanjskim razredima
  - u primjeru su to razredi `Horse` i `Dog`
  - mapiranje metode `greet` provodimo iz konstruktora

```
class Horse:
  def __init__(self, name):
    self.name=name
  def neigh(self):
    return "Njihihi!"

# class PoliceHorse ...
#    would be repetition!

class PoliceAdapter:
  def __init__(self, adaptee, method):
    self.adaptee = adaptee
    self.greet = method        # magic!
```

```
d2=Dog("Atila")
h1=Horse("Caesar")

o1=PoliceAdapter(h1, h1.neigh)
o2=PoliceAdapter(d2, d2.bark)

report(o1) # works!
report(o2) # works!
```

# Prilagodnik: primjeri upotrebe (1)

Prilagodnike vrlo često koristimo za pristupanje bibliotečnim komponentama preko sučelja koje odražava naše specifične potrebe

U Pythonu, metoda `makefile` razreda `Socket` vraća prilagodnik koji matični objekt predstavlja pod sučeljem datoteke

```
address=('www.fer.unizg.hr',80)
s=socket.socket()
s.connect(address)
s.send(b"GET / HTTP/1.1\nHost: www.fer.unizg.hr\n\n")
for line in s.makefile():
  # ...
```

U STL-u (C++) razredi `std::stack` i `std::queue` su implementirani kao prilagodnici općenitijeg razreda `std::dequeue`

- konciznije sučelje prilagođenih razreda može dovesti do jasnijeg kôda

# Prilagodnik: primjeri upotrebe (2)

Prilagodniku su srodni pomoćni razredi s praznim implementacijama apstraktnih sučelja.

Takve pomoćne razrede Swing (Java) definira za mnoga sučelja promatrača (odnosno naredbi)

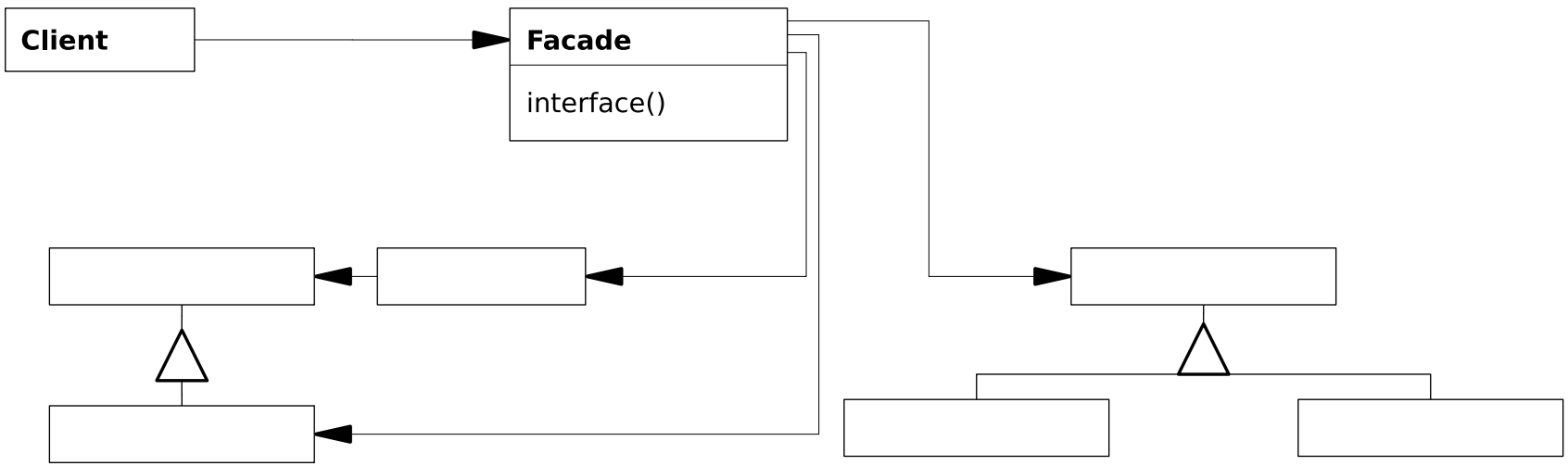
- razredi `XYZAdapter` navode prazne implementacije dojavnih metoda (`update`) sučelja promatrača (`XYZListener`)
- nasljeđivanjem tih "prilagodnika" postižemo kraći kôd kad ne želimo reagirati na sve dojavne metode.

```
// define the adapter class and provide a desired update method
// (deriving from WindowListener would require 7 overrides)
public class MyWindowCloseListener extends WindowAdapter {
    @Override
    public void windowClosed(WindowEvent e) {
        System.exit(0);
    }

// register an adapted object
mywindow.addWindowListener(new MyCloseListener());
```

# Prilagodnik: usporedba

- **Dekorator**: poboljšana funkcionalnost uz zadržavanje sučelja
- Fasada: pojednostavljeno umjesto promijenjeno sučelje
  - struktura vrlo slična Prilagodniku, samo što se iza novog sučelja krije cijeli podsustav
  - namjera: odvajanje klijenata od izvedbenih detalja enkapsuliranog podsustava
  - razred `WindowAdapter` na pola je puta između Prilagodnika i i Fasade



# Kompozit: namjera i motivacija

**Namjera:**
Omogućiti da se grupom objekata može baratati na isti način kao i s instancama pojedinih objekata

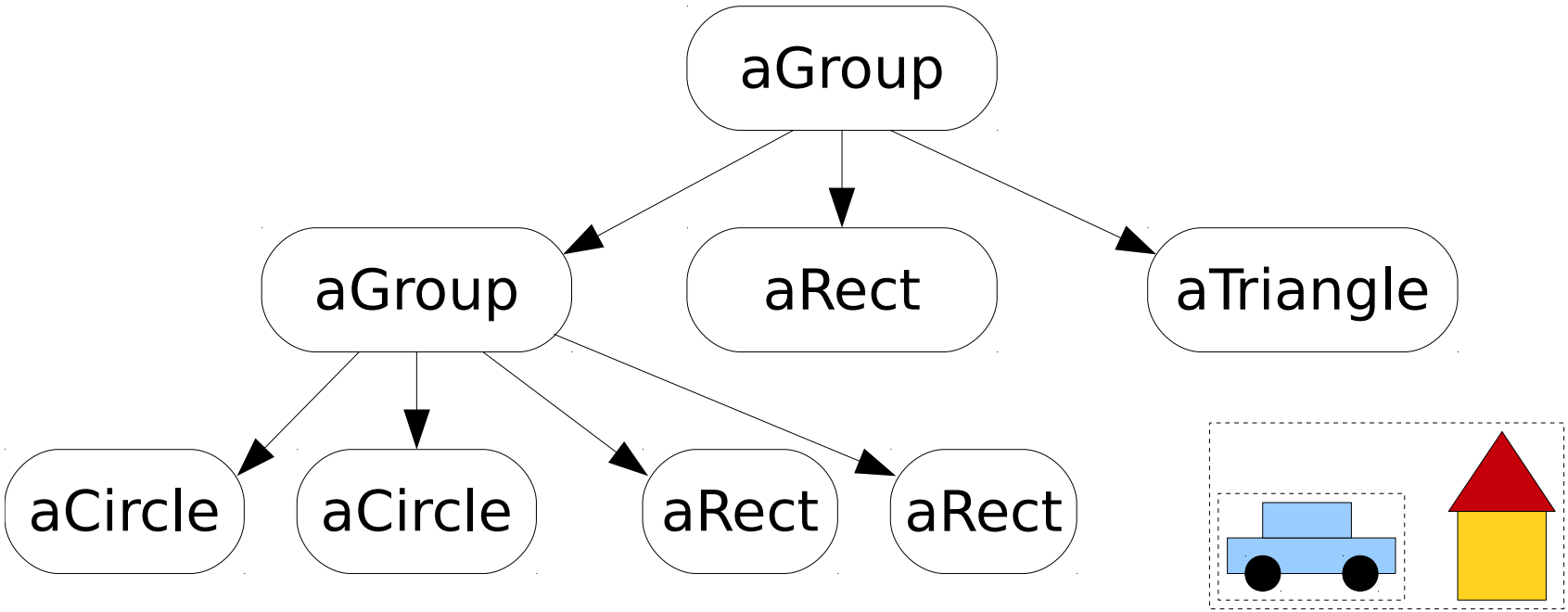
Ukomponirati objekte u rekurzivnu hijerarhiju sačinjenu od pojedinačnih objekata i njihovih grupa

**Motivacijski primjer:**

- Grupiranje pojedinačnih elemenata vektorskog crteža (tekst, linije, teksturirani poligoni, rasterske slike, ...)
- kompozitni objekt postaje ``punopravni'' element crteža (tretira se kao pojedinačni objekti: iscrtavanje, pomicanje, atributi)
- operacije nad kompozitnim objektom delegiraju se sastavnim dijelovima!

# Kompozit: motivacijski primjer

Struktura tipičnog kompozitnog objekta:



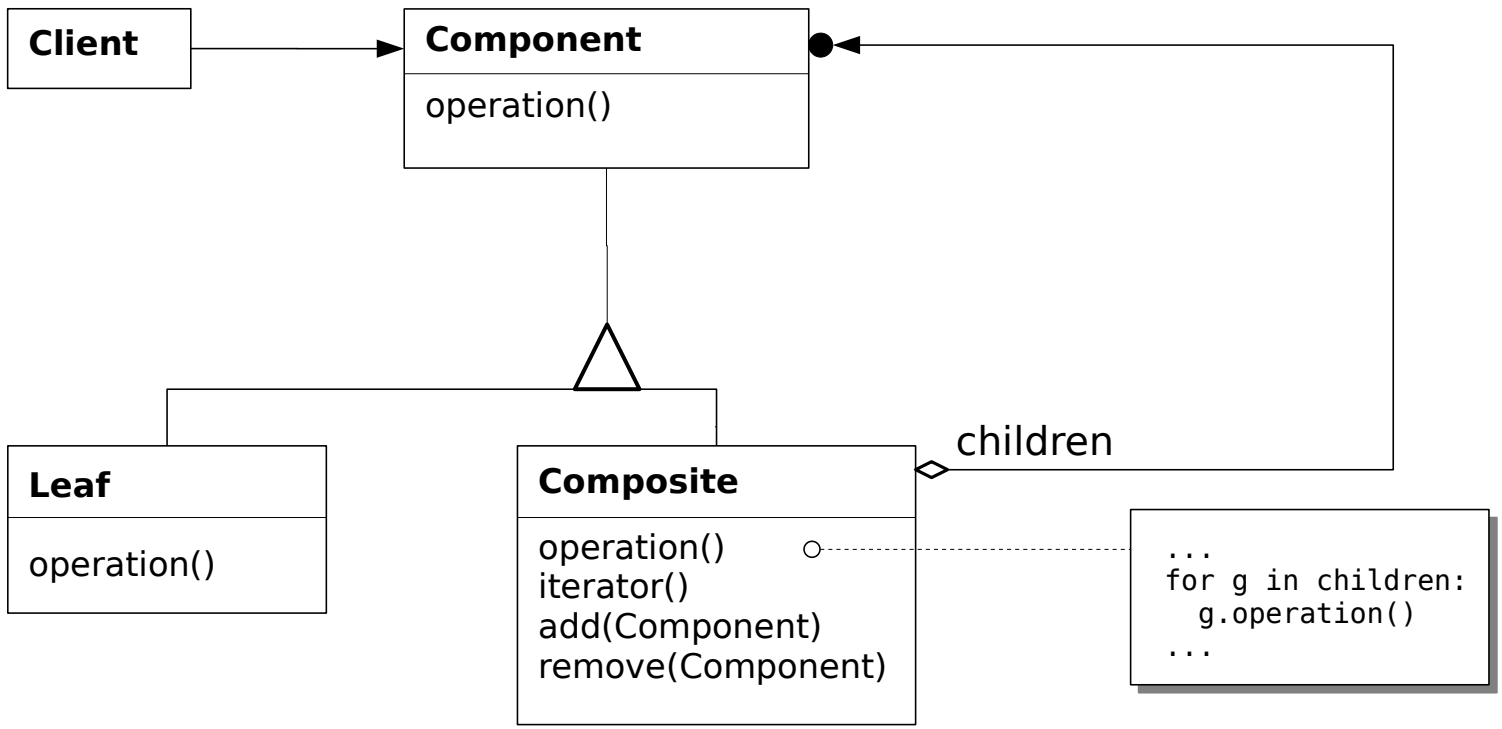
# Kompozit: primjenljivost i struktura

**Primjenljivost** -- Kompozit (eng. composite) se koristi za:

- predstavljanje hijerarhije cjelina i pojedinačnih dijelova
- transparentno operiranje nad elementima takvih hijerarhija

Osnovne značajke:

- **strukturni** obrazac: kompozit **sadrži** komponente
- temeljna ideja -- rekurzivna kompozicija (jesmo li to već imali?)



# Kompozit: sudionici

- **Komponenta** (element vektorskog crteža)
  - deklarira zajedničko osnovno sučelje za primitive i kompozite
- **Primitiv** (poligon, tekst, rasterska slika)
  - predstavlja listove hijerarhije (listovi nemaju djecu)
  - definira ponašanje atomarnih konkretnih komponenata
- **Kompozit** (Grupa)
  - definira ponašanje složenih komponenata (roditelja)
  - odgovoran za pohranjivanje sastavnih komponenata (djece)
  - delegira operacije sastavnim komponentama (djeci)
- **Klijent**
  - transparentno manipulira složenim objektima i primitivima preko sučelja komponente

# Kompozit: suradnja

- Klijenti koriste elemente hijerarhije preko sučelja Komponente
- pozivi nad Primitivima se obrađuju izravno
- pozivi nad Kompozitima rezultiraju delegiranjem poziva sastavnim Komponentama
- Kompoziti mogu obaviti dodatne operacije prije ili poslije delegiranja

# Kompozit: posljedice

- Definira se hijerarhija koja se sastoji od Primitiva i Kompozita: klijent koji radi s Primitivima može primiti i Kompozite (LNS)
- pojednostavnjuju se odgovornosti klijenta jer se Primitivi i Kompoziti mogu tretirati na jednak način
- olakšano dodavanje novih Komponenata: novi Kompoziti i Primitivi automatski se uklapaju u postojeći kod
  (pospješuje se otvorenost za promjene)

# Kompozit: Implementacija

- Eksplicitne reference na roditelja:
  - omogućava prolazak kroz hijerarhiju u oba smjera (zgodno npr. kod brisanja komponenata)
  - logično mjesto za implementaciju je Komponenta
  - referenca se postavlja i briše pri dodavanju odnosno povlačenju Komponente iz Kompozita roditelja
  - u sučelje komponente dodajemo metodu: `Component::getParent`
- Kompoziti mogu cacheirati rezultate operacija kako bi ubrzali izvođenje budućih zadataka: npr, pamćenje pravokutnika opisanog djeci može ubrzati selekciju

# Kompozit: Implementacija (2)

Upravljanje sastavnim dijelovima kompozita može biti deklarirano u:

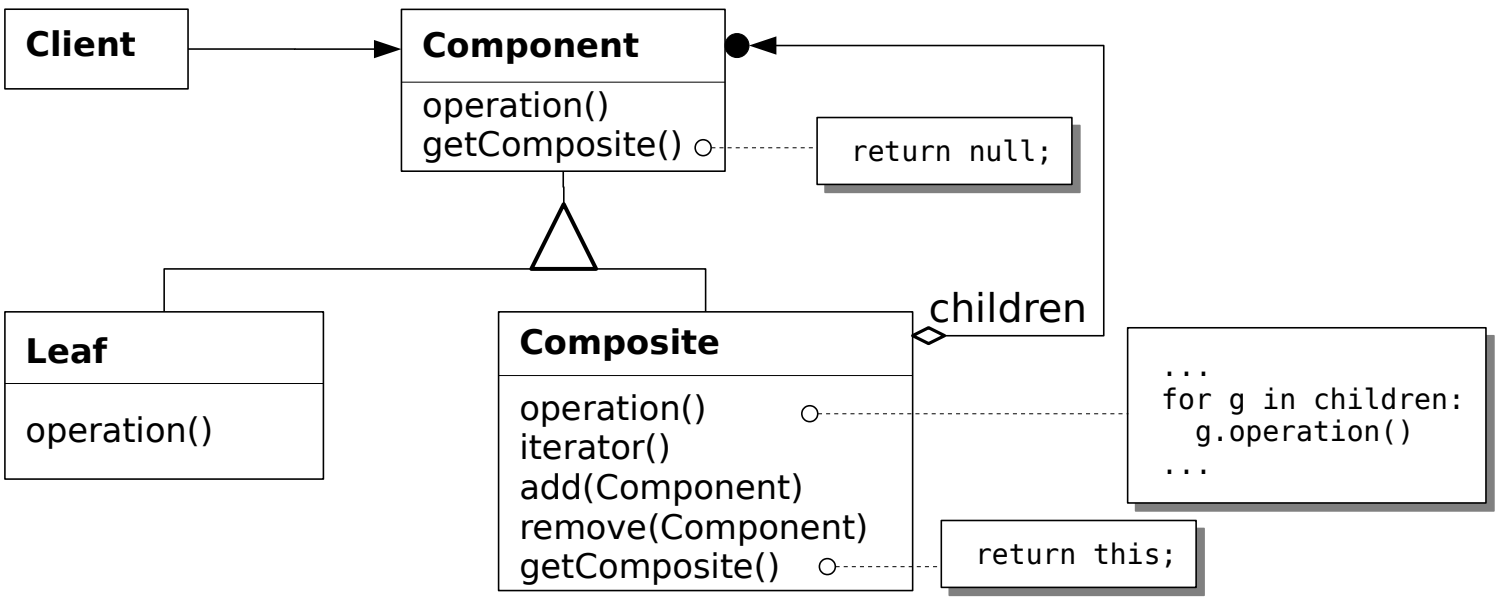
- Kompozitu, uz veći integritet (LNS) i manju transparentnost (?)
  - tipski korektno rješenje za pretvorbu: `Component::getComposite`
    - podrazumijevana implementacija vraća NULL
    - `Composite::getComposite` vraća **`this`**
  - takvo rješenje imamo u strukturnom dijagramu
- Komponenti, uz veću transparentnost (?), suvišne metode u listovima i potrebu za iznimkama
  - što napraviti kad se nad listom pozove `add`?
  - podrazumijevane implementacije metoda `add`, `remove` (a možda i `iterator`) mogu baciti iznimku!
  - takvo rješenje navedeno je u udžbeniku (GoF)

Ako ne koristimo automatsko rukovanje memorijom, odgovornost za brisanje listova je na Kompozitu

# Kompozit: Varijanta (1)

Metoda `getComposite` testira komponentu i prevodi je u kompozitni tip

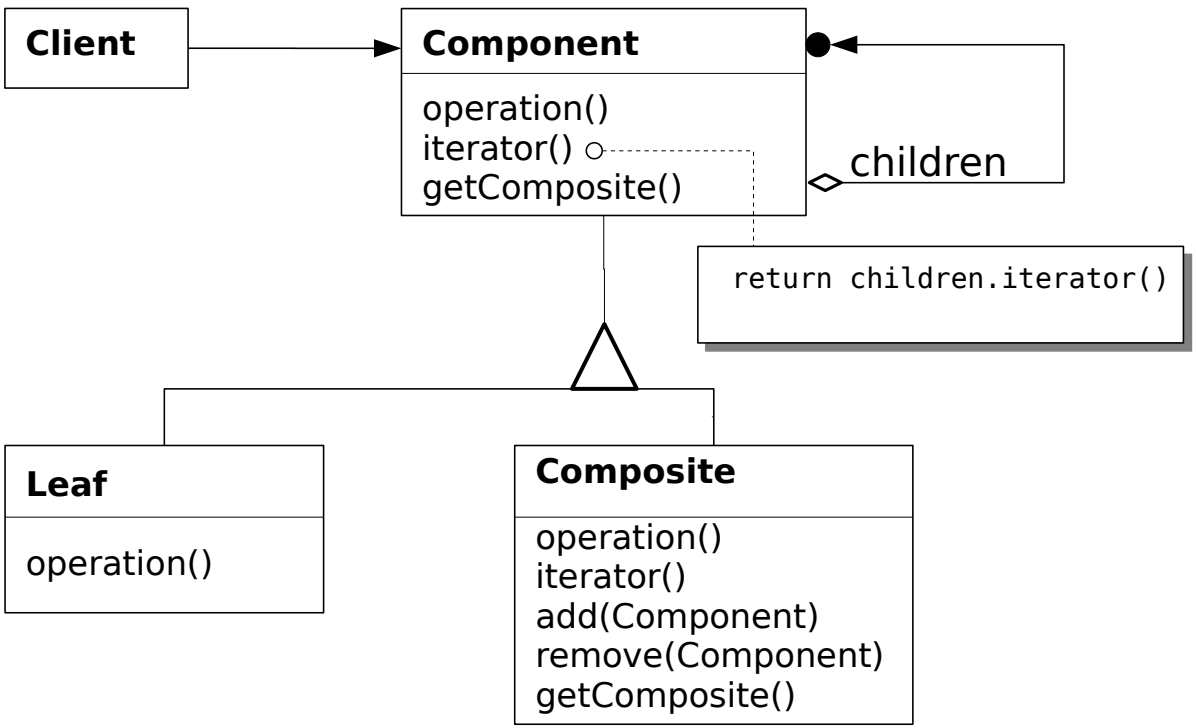
- korisno ako klijenti trebaju iterirati po kompozitima
- npr. tražimo tekst koji može biti na bilo kojoj dubini kompozita
- `Component::getComposite` može vratiti neprozirni `Composite*`
  - razred Component tada sadrži redak: **`class`** `Composite;`
- metoda može biti i **`bool`** `Component::isComposite`, ali tada klijent mora prevesti komponentu operatorom **`dynamic_cast`**



# Kompozit: Varijanta (2)

Komponenta preuzima na sebe odgovornost za iteriranje

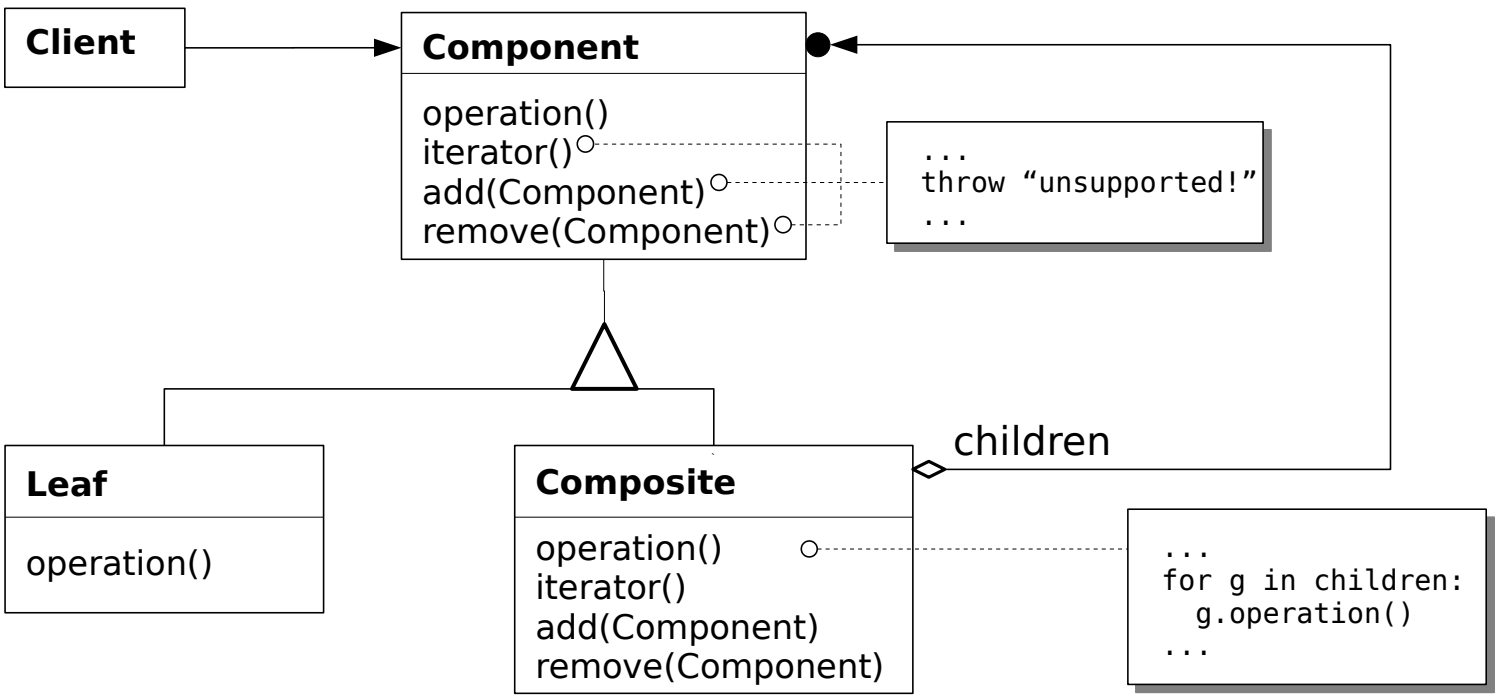
- metoda `iterator` listova vraća iterator na praznu listu (**OK**)
- listovi imaju listu djece (neefikasno)



# Kompozit: Varijanta (3)

Komponenta deklarira sve metode za upravljanje djecom

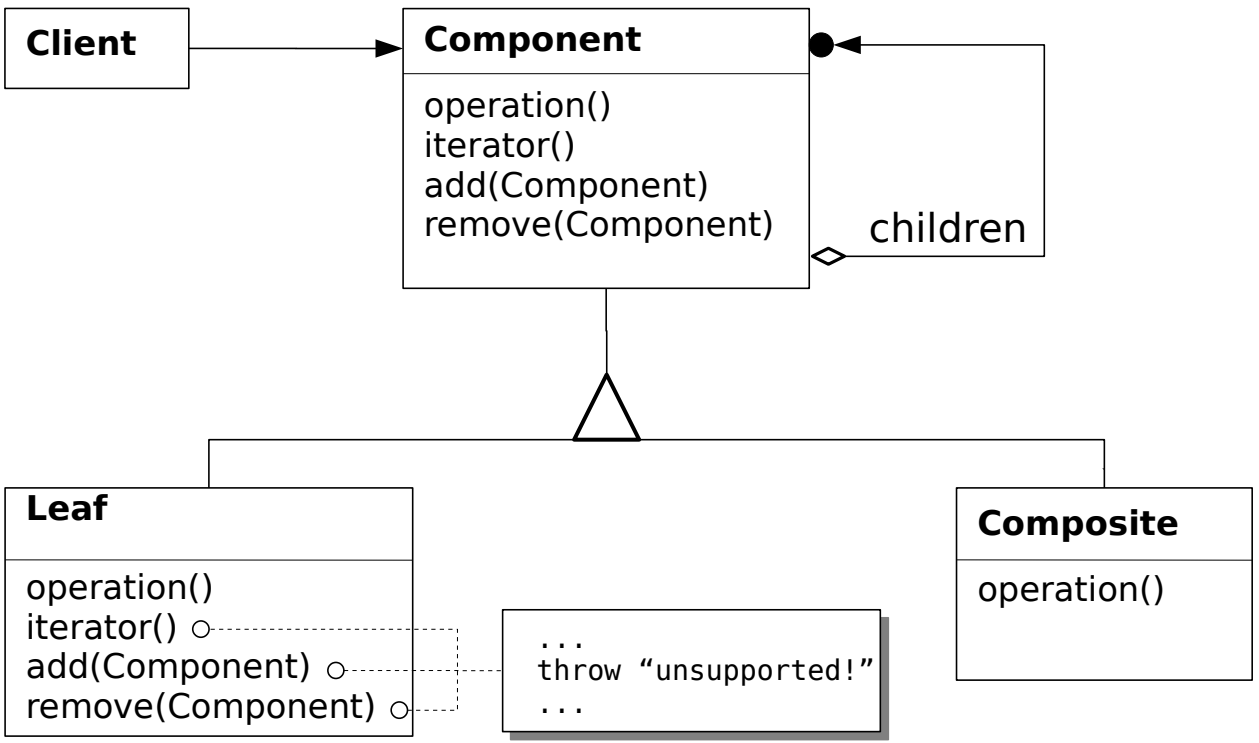
- implementacije metoda u komponenti bacaju iznimke (**OK**)
- prave implementacije pruža kompozit.



# Kompozit: Varijanta (4)

Komponenta deklarira i definira metode za upravljanje djecom

- implementacije metoda u listovima bacaju iznimke (ponavljanje)
- mogućnost cacheiranja operacija otpada



# Kompozit: primjene i srodni obrasci

**Primjene:**

- Gdje god su potrebne rekurzivne hijerarhije: datotečni sustav, računski izrazi, jezični procesori, ...

**Srodni obrasci:**

- Dekorator (1:1) je strukturno vrlo sličan Kompozitu (1:n), iako dva obrasca imaju različite namjere. Dekorator i kompozit mogu se kombinirati.
- Iterator se može koristiti za prolaz kroz elemente Kompozita
  - iteriranje po kompozitu može biti plitko ili duboko
  - duboki iterator mora sadržavati stog plitkih iteratora
- Makro-naredbe mogu biti kompoziti.

# Stanje: namjera i motivacija

**Namjera:**
Omogućiti dinamičko mijenjanje ponašanja objekta

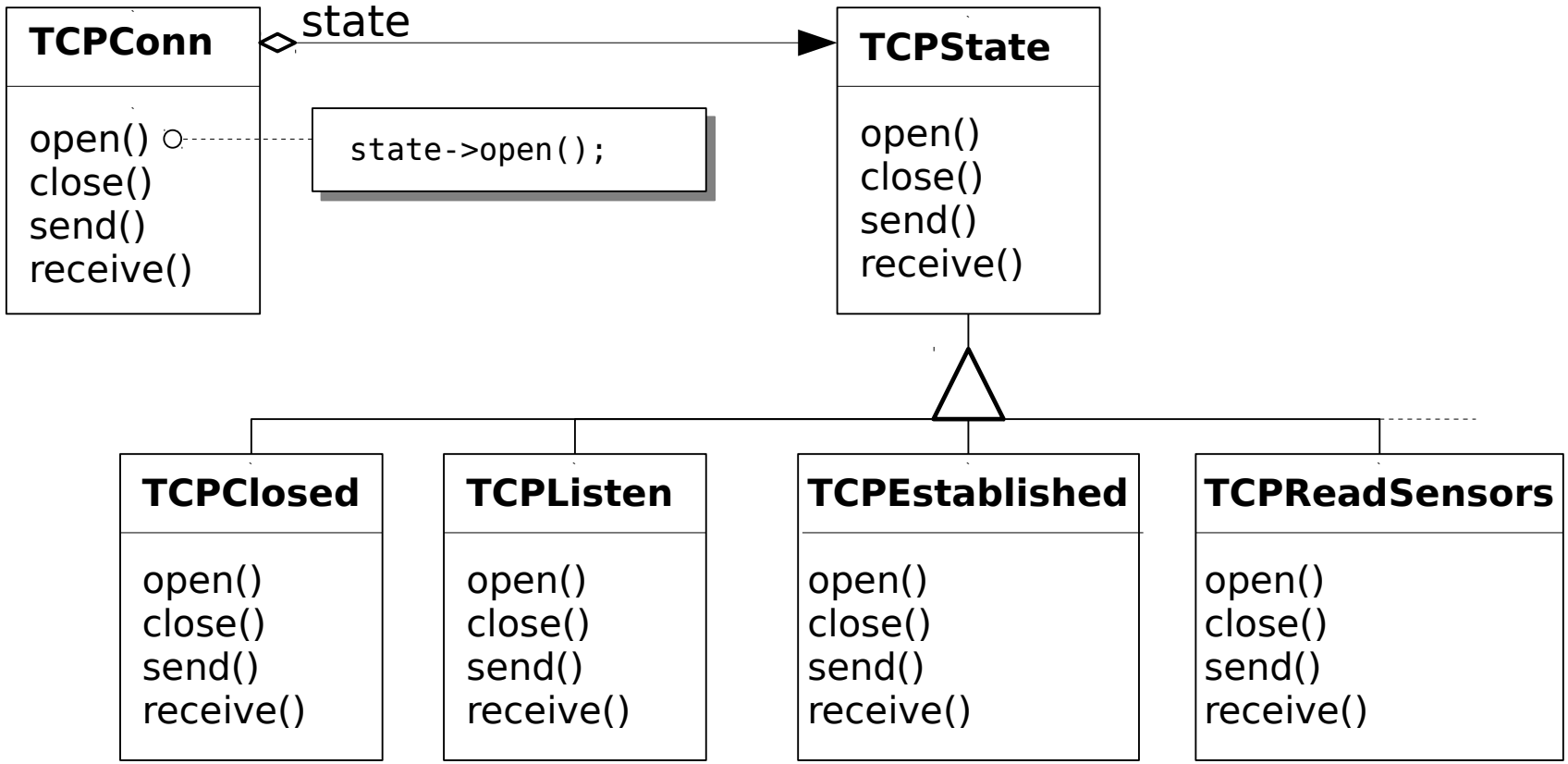
Objektno-orijentirana implementacija konačnog stroja

Efekt: (kao da) objekt dinamički mijenja klasu iz koje je instanciran (!)

**Motivacijski primjer:**

- Senzorski element povremeno otvara bežičnu vezu i prima konekcije udaljenih klijenata
- Veza može biti u tri stanja: zatvorenom, pripremnom i uspostavljenom
- Ponašanje komunikacijske komponente ovisi o stanju veze: npr, `TCPConnection::send()` nema smisla ako veza nije uspostavljena
- Želimo rano detektirati krivo korištenje komunikacijske komponente
- Želimo omogućiti jednostavno širenje komunikacijskog protokola

# Stanje: motivacijski primjer




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

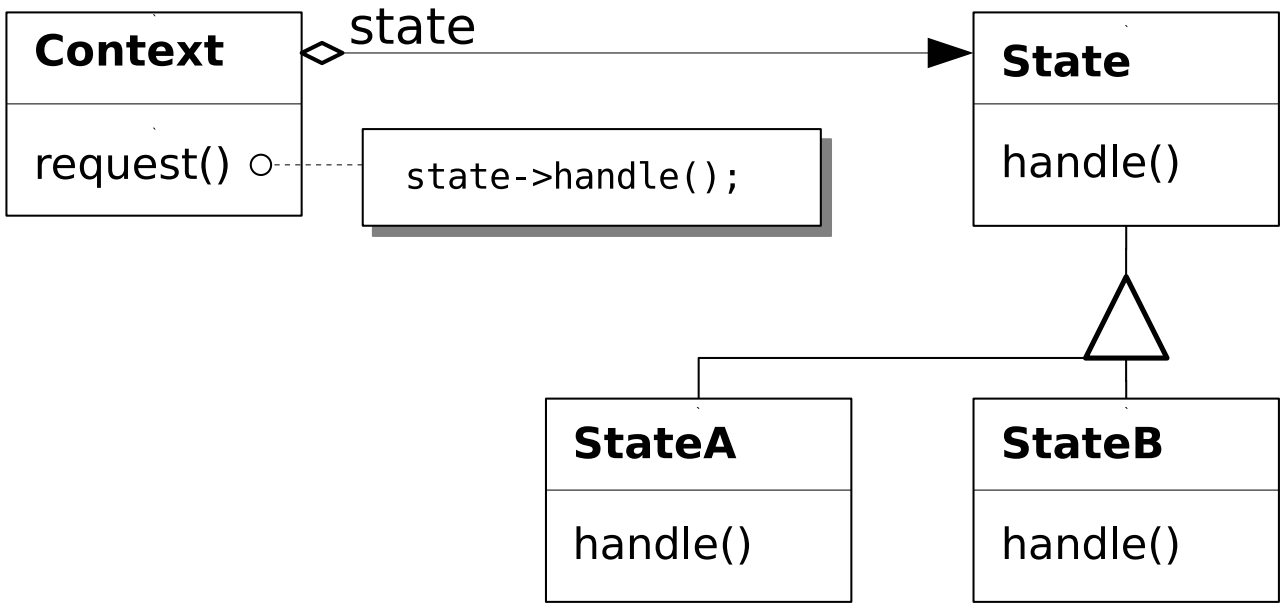
# Stanje: primjenljivost i struktura

**Primjenljivost** -- Stanje (eng. state) se koristi kad:

- ponašanje objekta mora se dinamički uskladiti s tekućim stanjem
- metode imaju istovrsne uvjetne izraze koji ispituju stanje objekta
- obrazac smješta odgovarajuće uvjetne grane u izdvojeni razred, i time omogućava dinamičku konfiguraciju ponašanja delegacijom

Osnovne značajke:

- **ponašajni** obrazac: kontekst referencira apstraktno sučelje stanja



# Stanje: sudionici

- **Kontekst** (TCPConnection)
  - definira sučelje koje koriste klijenti
  - održava referencu na konkretni razred koji definira tekuće stanje
- **Stanje** (TCPState)
  - deklarira sučelje za enkapsuliranje ponašanja koje ovisi o tekućem stanju
- **Konkretno stanje** (TCPEstablished, TCPListen, TCPClosed)
  - svaka izvedena klasa definira skup ponašanja vezanih uz pojedinačno stanje

# Stanje: suradnja

- Kontekst delegira zahtjeve tekućem Konkretnom stanju
- Kontekst može poslati sebe kao argument metode Stanja.
- Klijenti mogu konfigurirati Kontekst s početnim stanjem; kasnije Klijenti u načelu ne barataju izravno s Konkretnim stanjima
- Daljnje prijelaze stanja autonomno vrši Kontekst ili konkretna Stanja.

# Stanje: posljedice

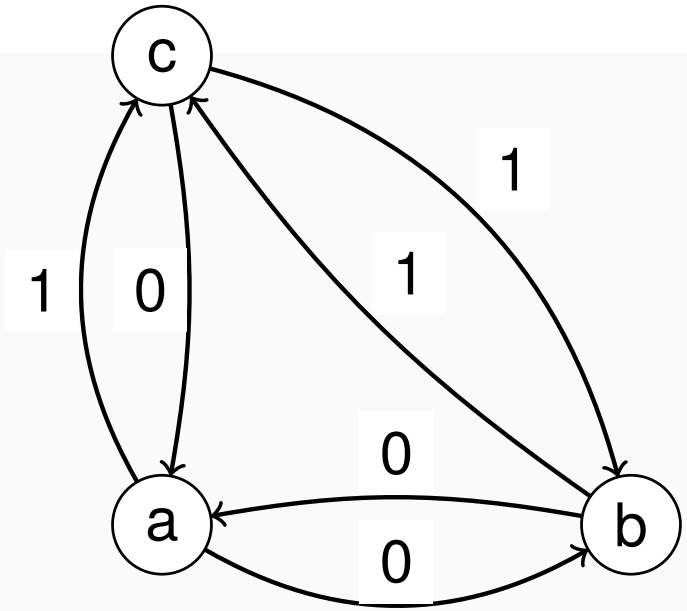
- Lokalizacija ponašanja koja su relevantna za pojedina stanja (pospješuje se ortogonalnost sustava)
- promjena stanja objekta je atomarna i eksplicitna (pospješuje se jasnoća kôda)
- Ako konkretno stanje nema podatkovnih članova, odgovarajuća instanca može biti jedinstvena i dijeljena

# Stanje: Implementacija

- Tko definira prijelaze stanja?
  - najčišće je prijelaze definirati u okviru Konteksta
  - u praksi, često je najprikladnije da prijelaze definiraju stanja
    - ◊ Kontekst pruža sučelje za promjenu stanja
    - ◊ totalna međuovisnost Konkretnih stanja (generička tvornica?)
- alternativni pristup, izgradnja eksplicitne tablice prijelaza stanja, najčešće rezultira složenijim kôdom
- Konkretna Stanja mogu biti kreirana po potrebi (sporije) ili unaprijed (neprikladno kod velikog broja stanja)
- U nekim jezicima (Python), moguće je dinamički promijeniti razred objekta (izvedba obrasca se pojednostavnjuje)
  `http://www.onlamp.com/pub/a/python/excerpt/pythonckbk_chap1/index1.html`

# Stanje: Python

```
class StateMachine:
  class StateA:
    def state(self): return "A"
    def event0(self):
      self.__class__=StateMachine.StateB
    def event1(self):
      self.__class__=StateMachine.StateC
  class StateB:
    def state(self): return "B"
    def event0(self):
      self.__class__=StateMachine.StateA
    def event1(self):
      self.__class__=StateMachine.StateC
  class StateC:
    def state(self): return "C"
    def event0(self):
      self.__class__=StateMachine.StateA
    def event1(self):
      self.__class__=StateMachine.StateB
  def __init__(self):
    self.__class__=StateMachine.StateA
```




```
s=StateMachine()
>>> s.state()
'A'
>>> s.event0()
>>> s.state()
'B'
>>> s.event0()
>>> s.state()
'A'
```

# Stanje: primjene i srodni obrasci

**Primjene:**

- Stanje je korišteno u aplikacijama za crtanje, za definiranje ponašanja koje ovisi o trenutno odabranom alatu (selekcija, crtanje, ispunjavanje itd)

**Srodni obrasci:**

- Konkretna stanja mogu biti Jedinstveni objekti.
- Stanje ima istu strukturu kao i Strategija, ali je namjera drukčija
  - stanja za razliku od strategija često definiraju sljedeća stanja

# Proxy: namjera i motivacija

**Namjera:**

- zamjenski objekt (surogat) upravlja pristupom stvarnom subjektu
- dodatna razina indirekcije: dijeljeni, kontrolirani ili ``pametni'' pristup
- delegirajući omot pojednostavnjuje implementaciju ciljnog subjekta

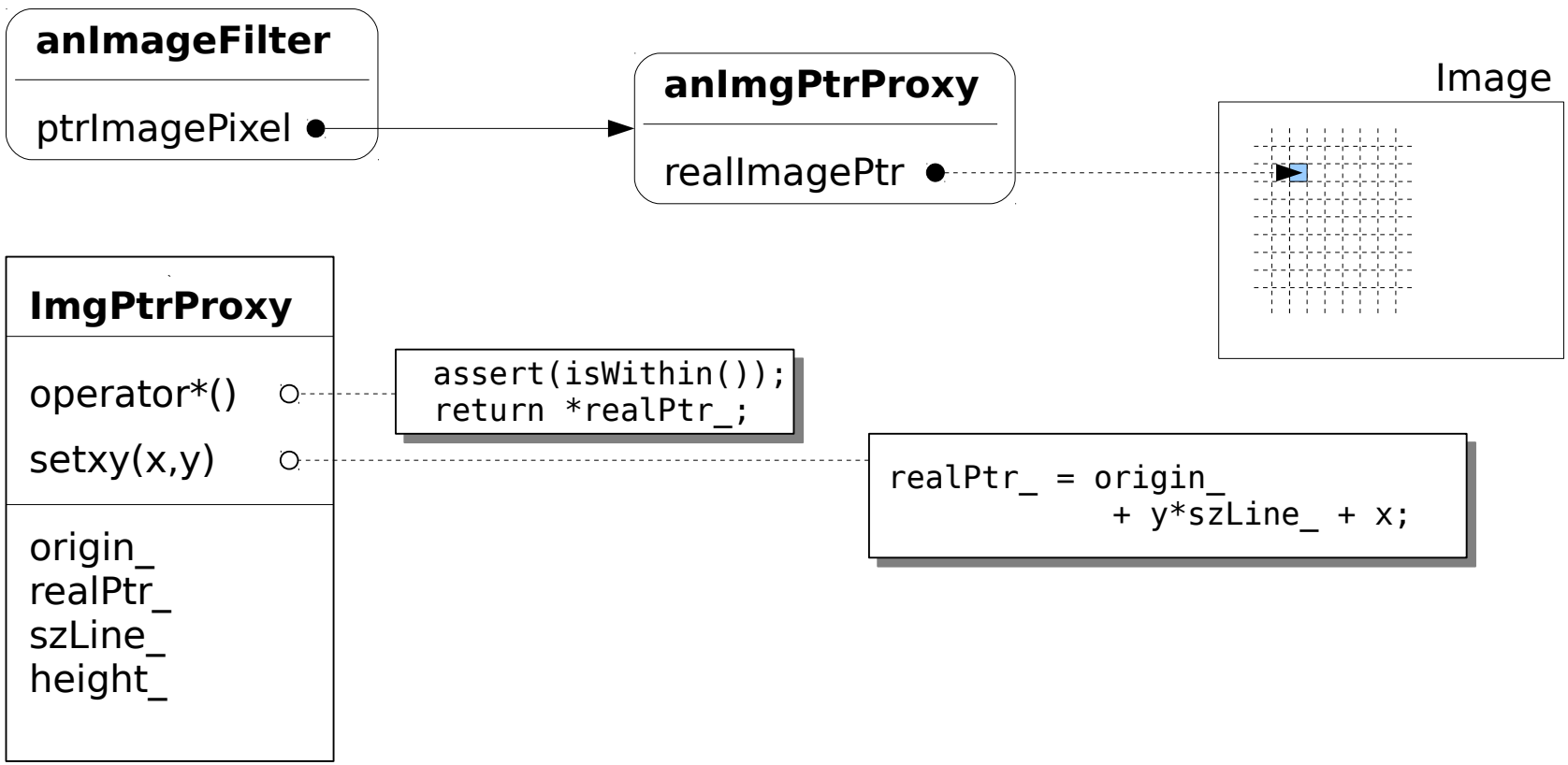
**Motivacijski primjer:**

- Često je korisno odgoditi kreiranje teških objekata
  - pri učitavanju velikog dokumenta, ne učitati odmah i sve slike
  - cilj: omogućiti korisniku što brži početak rada (produktivnost $\uparrow$)
  - sliku učitavamo tek kad nam stvarno treba (npr, treba je iscrtati)
  - primjer koncepta odgođenog izračunavanja (lazy evaluation)
- ideja: postići **transparentni** **pametni** pristup

# Proxy: motivacijski primjer

Pametni pokazivač (smart pointer): učestala varijanta Proxyja

- Omotač oko sirovog pokazivača, transparentna dodatna funkcionalnost
- ciljevi: detekcija grešaka, automatsko recikliranje

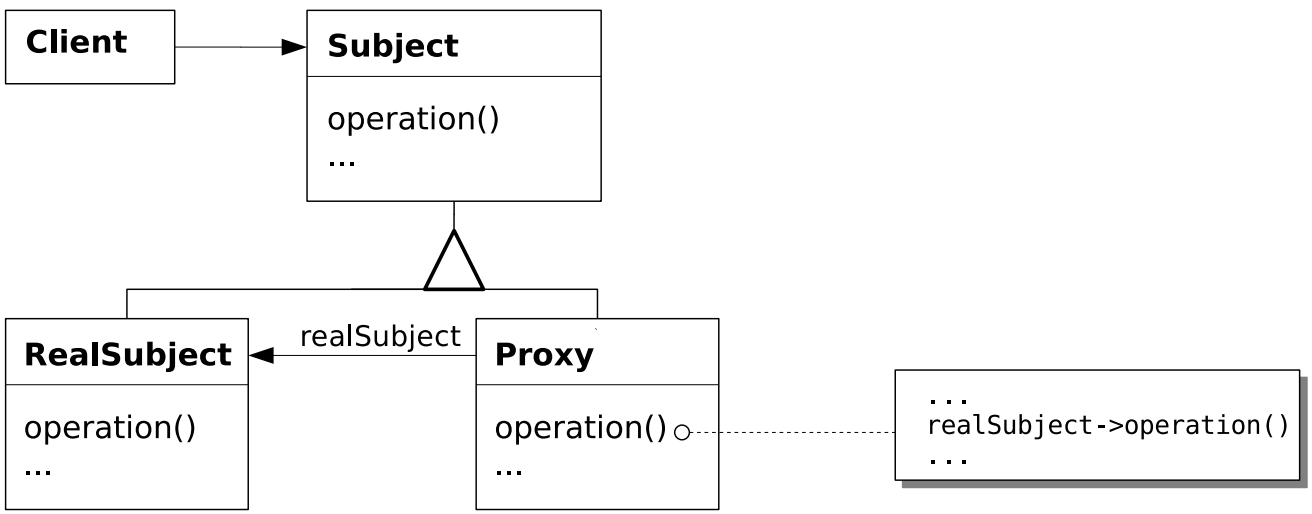


# Proxy: primjenljivost i struktura

**Primjenljivost** -- podatan i/ili sofisticiran pristup resursu

- Udaljeni proxy: ciljni subjekt je u različitom adresnom prostoru
- Virtualni proxy: kreira teške subjekte na zahtjev (s odgodom)
- Sigurnosni proxy: omogućava pozive samo nekim klijentima
- Cacheirajući, sinkronizacijski, podstrukturni, ..., pametni pokazivač

Proxy (surogat, posrednik): **strukturni** obrazac (proxy umata stvarni subjekt)



# Proxy: sudionici i suradnja

**Sudionici:**

- **Subjekt** (ImageBase)
  - Zajedničko sučelje za Proxy i konkretni subjekt
- **Proxy** (ImageProxy)
  - čuva referencu na ciljni subjekt (ili zajednički bazni razred)
  - nasljeđuje sučelje Subjekta kako bi se omogućila transparentnost
  - kontrolira pristup stvarnom subjektu, može biti odgovaran za stvaranje i brisanje (detalji ovise o vrsti Proxyja)
- **Stvarni subjekt** (Image)
  - definira stvarni objekt koji ne zna za Proxy koji ga predstavlja

**Suradnja**:

- Proxy kontrolira pristup Stvarnom subjektu te mu prosljeđuje pozive

# Proxy: posljedice

Dodatna razina indirekcije može donijeti sljedeće konkretne koristi:

- transparentan rad s objektima u različitom adresnom prostoru
- odgođeno kreiranje objekata (ovo uključuje i kopiranje uslijed mijenjanja)
- izvršavanje administrativnih zadataka prije/poslije pristupa objektu: za razliku od dekoratora, zadatci nemaju veze s osnovnom odgovornošću komponente
- transparentni pristup podstrukturama kao samostalnim objektima

Možemo očekivati sljedeće općenitije posljedice:

- bolju preraspodjelu odgovornosti u sustavu (ortogonalnost)
- veću složenost programa, više objekata, sporiji pristup

# Proxy: Implementacija

Kod pametnih pokazivača ključno je *nadomještanje* operatora:

- dereferenciranje pokazivača `*`, pristup članu strukture `->`

Proxy stvarnom subjektu pristupa preko što apstraktnijeg sučelja (virtualni Proxy obično pozna konkretni tip čije primjerke stvara)

- najtransparentnije: Proxy nasljeđuje Subjekt
- to nije uvijek praktično izvedivo zbog cijene polimorfnog poziva (pametni pokazivač, podstrukturni proxy)
- značaj ovoga, kao i obično, ovisi o jeziku (C++, Java, Python)

Referenca na ciljni subjekt može biti različito izvedena: pokazivač, host:port, ime datoteke ...

Udaljeni proxy izravno podržan standardnom bibliotekom Jave:

`java.rmi.server`, `java.rmi.remote`, `rmic`, `rmiregistry`

# Proxy: primjene i srodni obrasci

**Primjene:**

- Virtualni proxy korišten u aplikacijama za obradu dokumenata
- udaljeni proxy standardni element Corbe i Jave
- pametni pokazivači elementi biblioteka libstdc++ (`auto_ptr`) i boost (`scoped_ptr`, `shared_ptr`)
- sigurnosni proxy korišten u biblioteci TCP wrappers
- podstrukturni proxy korišten u biblioteci ublas (boost) za pristup dijelovima matrica i vektora (`matrix_row`, ...)

**Srodni obrasci:**

- Adapter željeni objekt predstavlja pod **različitim** sučeljem
- Dekorator je strukturno vrlo sličan osnovnoj verziji Proxyja. Proxy i dekorator se prvenstveno razlikuju u namjeri.

# Most: namjera i motivacija

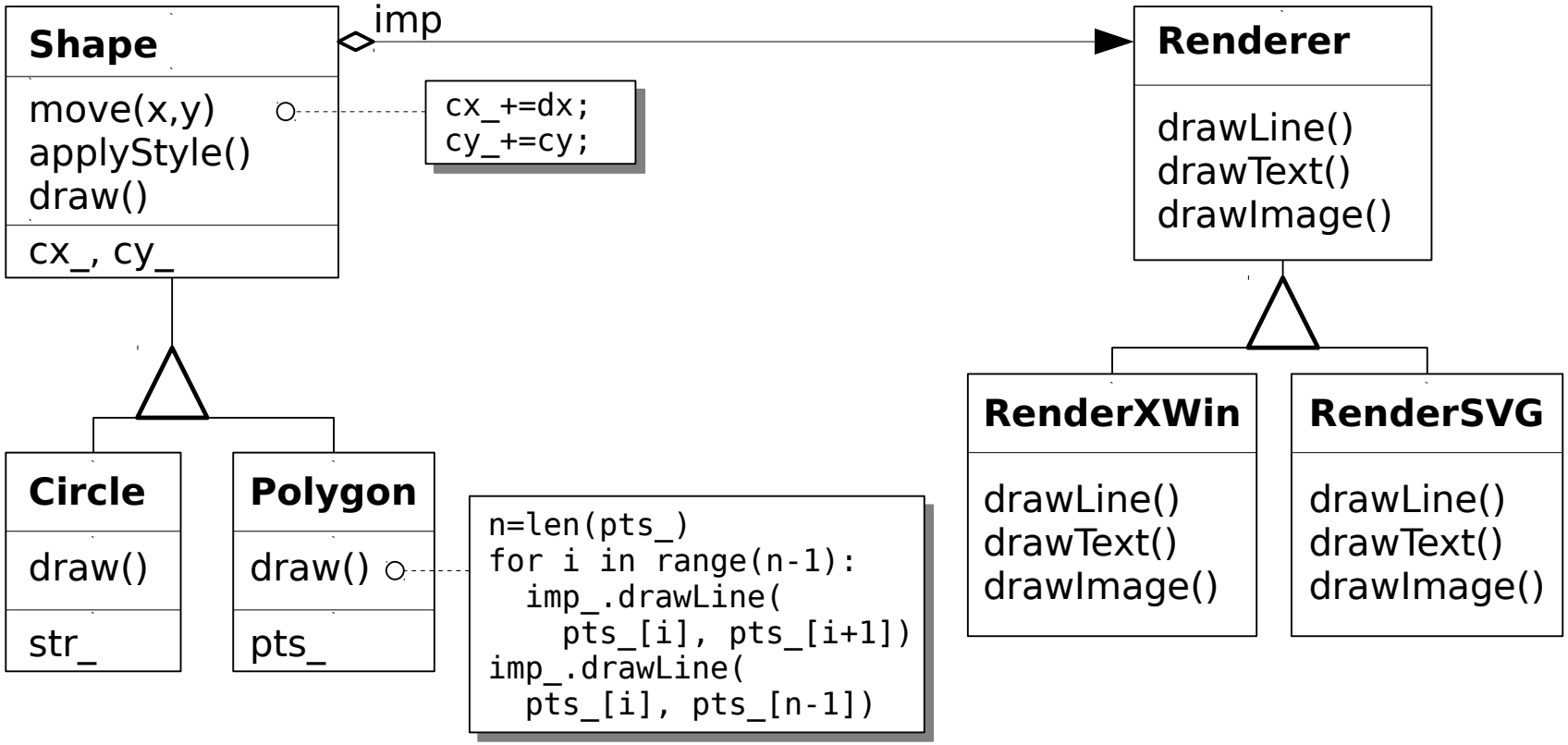
**Namjera:**

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

**Motivacijski primjer:**

- Program za vektorsku grafiku koji barata s poligonima, tekstom i rasteriziranim slikama
- Fokus: metoda za iscrtavanje grafičkog objekta
- Želja: podržati različite biblioteke za crtanje te omogućiti eksportiranje u različite grafičke formate
- Monolitna hijerarhija problematična:
  - sklonost kvadratnom porastu broja razreda: `PolygonWin32`, `PolygonSVG`, ..., `EllipseWin32`, `EllipseSVG`, ...
  - potiče vezivanje kôda uz jednu platformu
- ideja: dvije osi varijacije modelirati odvojenim hijerarhijama

# Most: motivacijski primjer




Geometrijski oblici iscrtavaju se pozivima objekta za crtanje (drivera)

Obitelj Shape ne ovisi o platformski ovisnom kôdu

- to će posebno cijeniti klijenti obitelji Shape!

Potrebno je isplanirati raspodjelu odgovornosti između dviju obitelji

- planiranje uvijek dobro dođe :-)

# Most: Primjenljivost

Izbjegavanje čvrstog vezivanja komponente s izvedbenim detaljem

- npr. kad izvedbeni detalj treba mijenjati tijekom izvršavanja
- npr. kad klijente treba izolirati od izvedbenih detalja

Posebno korisno kada su komponenta i izvedbeni detalj osnovni razredi

- svaku os varijacije modeliramo zasebnom hijerarhijom
- očekujemo promjene u implementacijskoj hijerarhiji, ne želimo da se propagiraju do klijenata

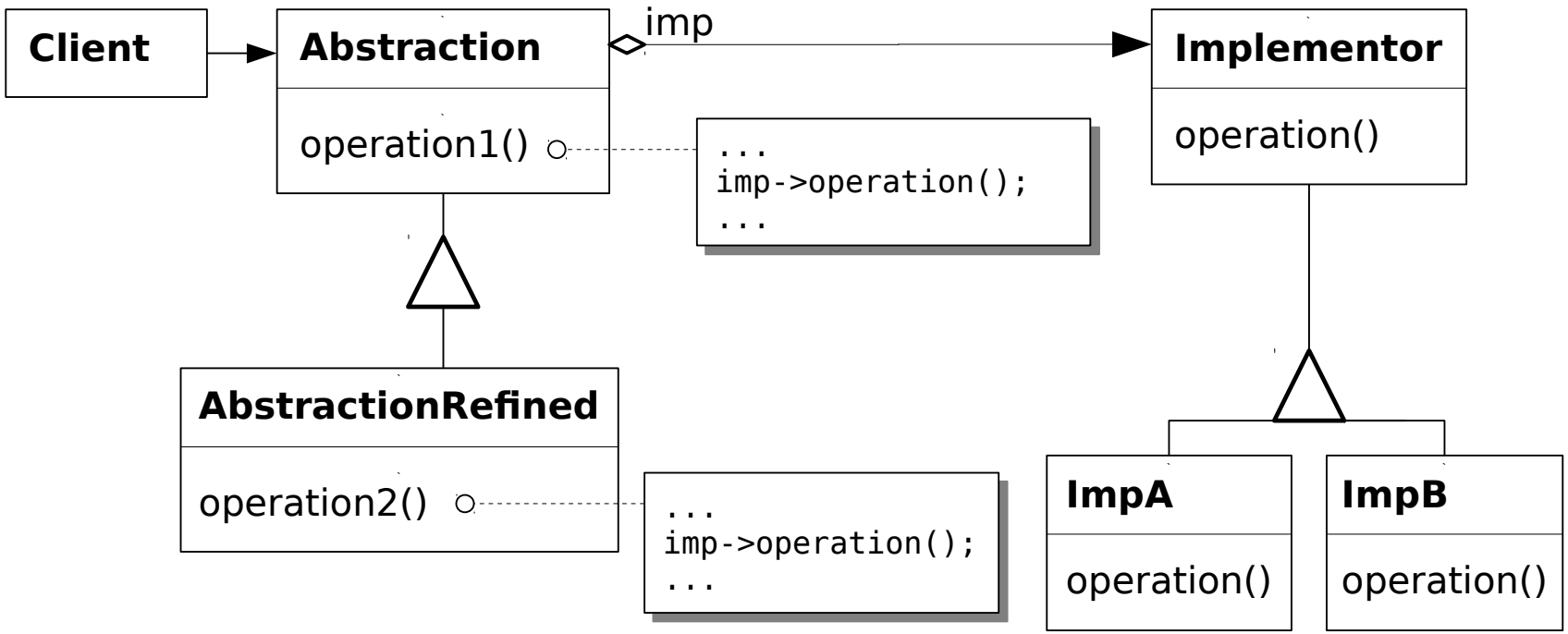
Most možemo koristiti i kao alternativu višestrukom nasljeđivanju:

- MI: GasPoweredLandVehicle $\rightarrow$ GasPoweredVehicle, LandVehicle
- Bridge: LandVehicle $\rightarrow$ Vehicle, GasEngine $\rightarrow$ Engine
- ograničavamo proliferaciju razreda: $O(n)$ vs $O(n^2)$

# Most: struktura

Most (bridge): **strukturni** obrazac

- lijeva hijerarhija *sadrži* izvođača iz desne hijerarhije



# Most: sudionici i suradnja

**Sudionici:**

- **Apstrakcija** (Shape)
  - Sučelje prema klijentima, sadrži referencu na Izvođača
- **Prilagođena apstrakcija** (Polygon)
  - proširuje sučelje Apstrakcije
- **Izvođač** (Renderer)
  - definira sučelje za implementacijske razrede
  - u odnosu prema Apstrakciji, operacije su primitivnije i niže razine
- **Konkretni izvođač** (RenderSVG)
  - izvodi sučelje Izvođača, definira konkretnu implementaciju

**Suradnja**:

- Apstrakcija prosljeđuje zahtjeve zadanom Izvođaču

# Most: posljedice

- Modeliranje rješenja razdvojenim hijerarhijama:
  - klijenti ne moraju ni znati za desnu hijerarhiju
  - bolja ortogonalnost (načelo jedinstvene odgovornosti)
  - dinamička konfiguracija lijeve hijerarhije (načelo nadogradnje bez promjene)
- Pospješujemo proširivost uslijed ortogonalnosti dviju obitelji
- Izoliramo klijente od implementacijskih detalja.
  - Pospješujemo binarnu kompatibilnost
    - ◊ možemo mijenjati desnu stranu (implementacije) bez potrebe za ponovnim prevođenjem lijeve strane

# Most: Implementacija

- Obrazac može biti koristan i ako postoji samo jedna izvedba
  - izolacija klijenata od implementacijskih detalja
  - AKA `pimpl` idiom, cheshire cat
  - obično se koristi u kombinaciji s neprozirnim pokazivačem
- Kako odabrati prikladnog izvođača?
  - izravni odabir (Apstrakcija), npr na temelju veličine kolekcije (npr, rječnik može biti izveden vektorom ili raspršenom tablicom)
  - posredni odabir (generička tvornica)
- Potrebno pripaziti pri kopiranju sučeljnih objekata (Apstrakcija):
  - plitko ili duboko kopiranje?
  - jedan Izvođač može biti korišten od strane više Apstrakcija...
  - ...tada može biti potrebno provesti *copy on write*
- Ponekad svi primjerci apstrakcije dijele samo jednog izvođača
  - izvođač može biti Jedinstveni objekt ili statički član Apstrakcije

# Most: primjene i srodni obrasci

**Primjene:**

- Ostvarivanje platformske neovisnosti (ET++)
  - sučeljni prozor `Window` delegira pozive izvedbenom prozoru `WindowPort` koji ovisi o platformi
- Odvajanje implementacije od sučelja (libg++)
  - skup se implementira listom ili raspršenom tablicom, ovisno o broju elemenata

**Srodni obrasci:**

- Konkretni Izvođač može se kreirati prikladnom tvornicom.
- Prilagodnik je sličan Mostu, a razlikuju se prvenstveno u namjeri.
- Most se razlikuje od Posjetitelja po tome što:
  - kod Mosta je izvođač podatkovni član apstrakcije, dok se kod posjetitelja odgovarajući objekt prenosi kao argument metode
  - konkretni izvođači ne znaju za konkretne razrede lijeve obitelji dok kod posjetitelja to nije slučaj
- Desna obitelj mosta po strukturi odgovara strategiji

# Prototip: motivacija i namjera

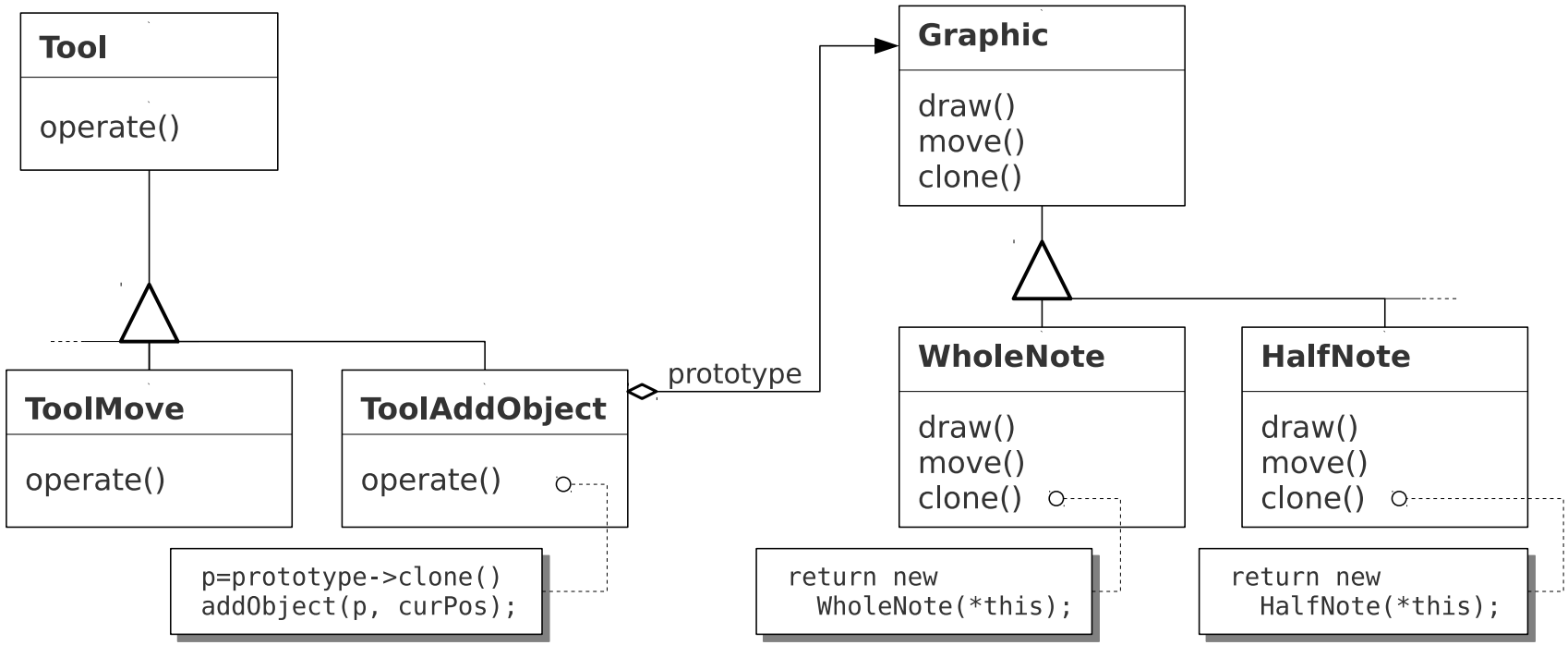
**Motivacijski primjer:**

- Razvijamo program za uređivanje glazbenih partitura (crtovlja, ključevi, note, itd.)
  - služimo se aplikacijskim okvirom u domeni vektorske grafike
  - okvir barata s apstraktnim operacijama "pomakni objekt", "uredi svojstva", "izreži objekt", ...
- kako implementirati "dodaj objekt" bez mijenjanja okvira?
  - unaprijed instancirati po jedan objekt svake vrste
  - instancirane objekte prikazati u prikladnom elementu GUI-ja
  - klik na objekt stvara njegovu kopiju pozivom metode `clone()`

**Namjera:**

- Odrediti razred novog objekta polimorfnim kloniranjem prototipa
- operator new considered harmful

# Prototip: motivacijski primjer



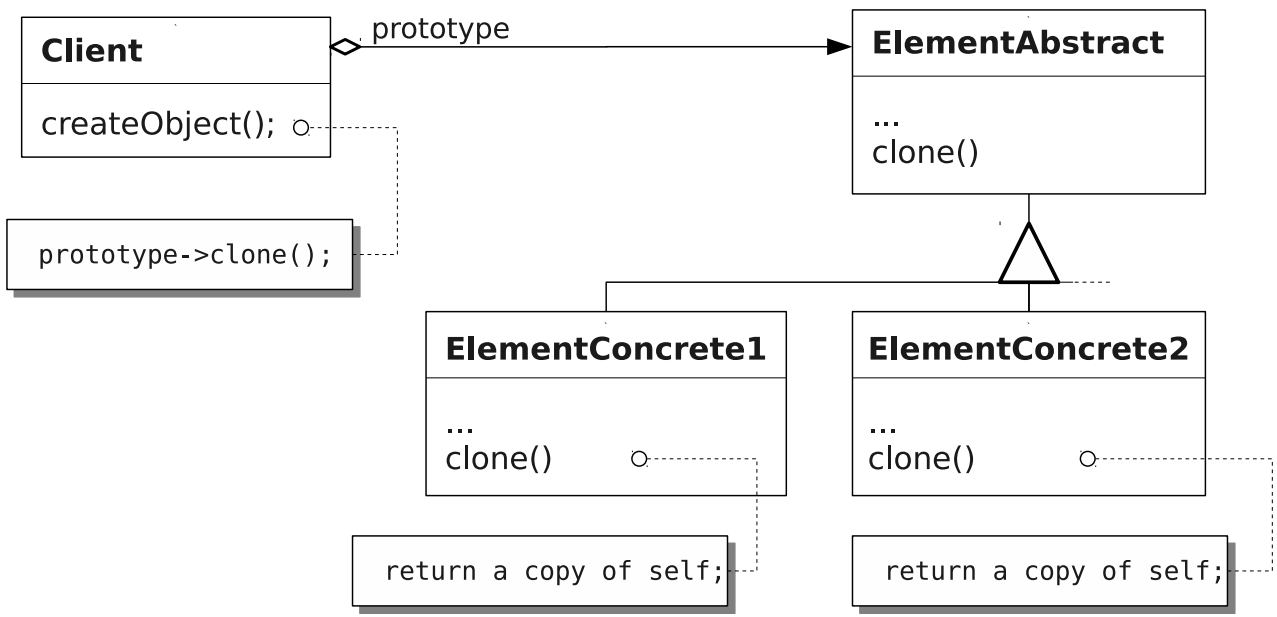
# Prototip: primjenljivost i struktura

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

- kada razredi koje treba instancirati nisu poznati u vrijeme pisanja programa (npr. mogu biti dinamički povezani)
- kada korisnik treba vidjeti objekt prije stvaranja

Prototip: **kreacijski** obrazac

- prototip *stvara* nove primjerke svog razreda



# Prototip: sudionici i suradnja

**Sudionici**:

- **Apstraktni element** (Graphic)
  - deklarira sučelje za kloniranje
- **Konkretni element** (HalfNote)
  - implementira sučelje za kloniranje
- **Klijent** (ToolAddObject)
  - održava pokazivač na aktivni prototip Apstraktnog elementa
  - instancira nove elemente kloniranjem prototipa

**Suradnja**:

- Klijent poziva operaciju kloniranja nad prototipom.

# Prototip: posljedice

- Kao i Tvornice, prototip skriva konkretne razrede od klijenata i tako pospješuje **proširivost bez promjena**
- omogućava se klijentima da instanciraju **naknadno razvijene** elemente (npr. iz dinamičkih biblioteka)
- prototip može biti i **složeni objekt** (kompozit ili dekorator) izveden iz apstraktnog elementa
- prikladan za kreiranje građevnih elemenata Kompozita
- Nedostatak: svaki konkretni element mora izvesti operaciju `clone()` (može se zakomplicirati kod složenih struktura ovisnosti)

# Prototip: Implementacija

- Kad broj Konkretnih razreda nije unaprijed poznat, može se koristiti registar prototipova:
  - asocijativno polje koje vraća prototip koji odgovara danom ključu
- problem s kloniranjem nastaje kad elementi sadrže kompliciranе strukture s cirkularnim vezama
  - za svaki član prototipa: plitko ili duboko kopiranje (serijalizacija)?
- klonirani prototip je potrebno moći konfigurirati kako bi ga se povezalo s postojećim elementima

# Prototip: primjene i srodni obrasci

**Primjene:**

- Programi za vektorsku grafiku, za unos novih elemenata crteža

**Srodni obrasci:**

- Prototip i tvornice su komplementarni obrasci, a mogu se i kombinirati (implementacija tvornice prototipom)
- prototip je prikladan za instanciranje kompozita i dekoriranih objekata

# Posjetitelj: namjera i motivacija

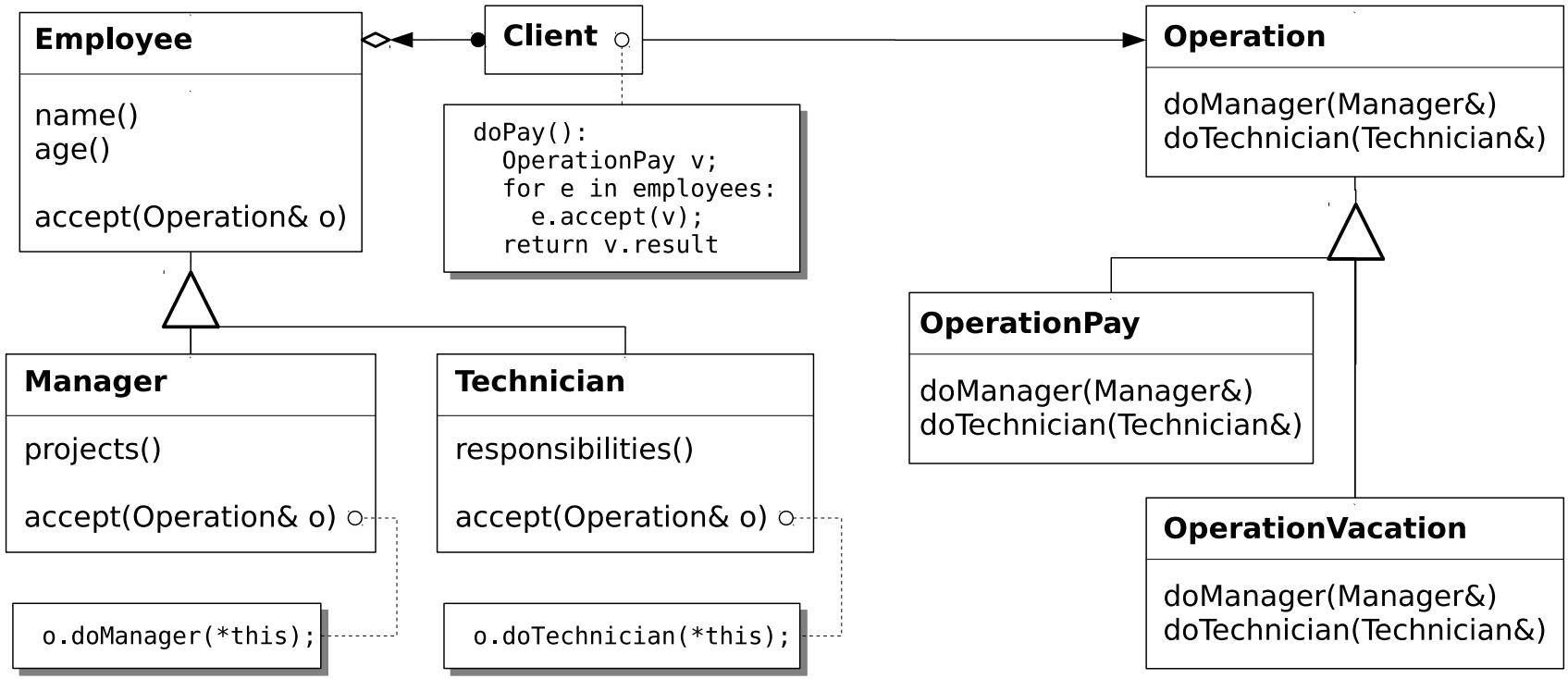
**Namjera:**

- modelirati operacije nad polimorfnim elementima skupnog objekta bez promjene sučelja (operacije ovise o konkretnom razredu)
- polimorfni poziv koji ovisi o konkretnom razredu dvaju objekata (double dispatch, dvostruko prosljeđivanje?)

**Motivacijski primjer:**

- program kadrovske službe obračunava plaće, godišnje odmore, itd
- zadatci ovise o razredu zaposlenika (održavanje, inženjer, direktor)
- ne želimo zadatke smještati u sučelje zaposlenika zbog načela jedinstvene odgovornosti
- poziv ovisi o (i) razredu zaposlenika i (ii) razredu zadatka

# Posjetitelj: motivacijski primjer

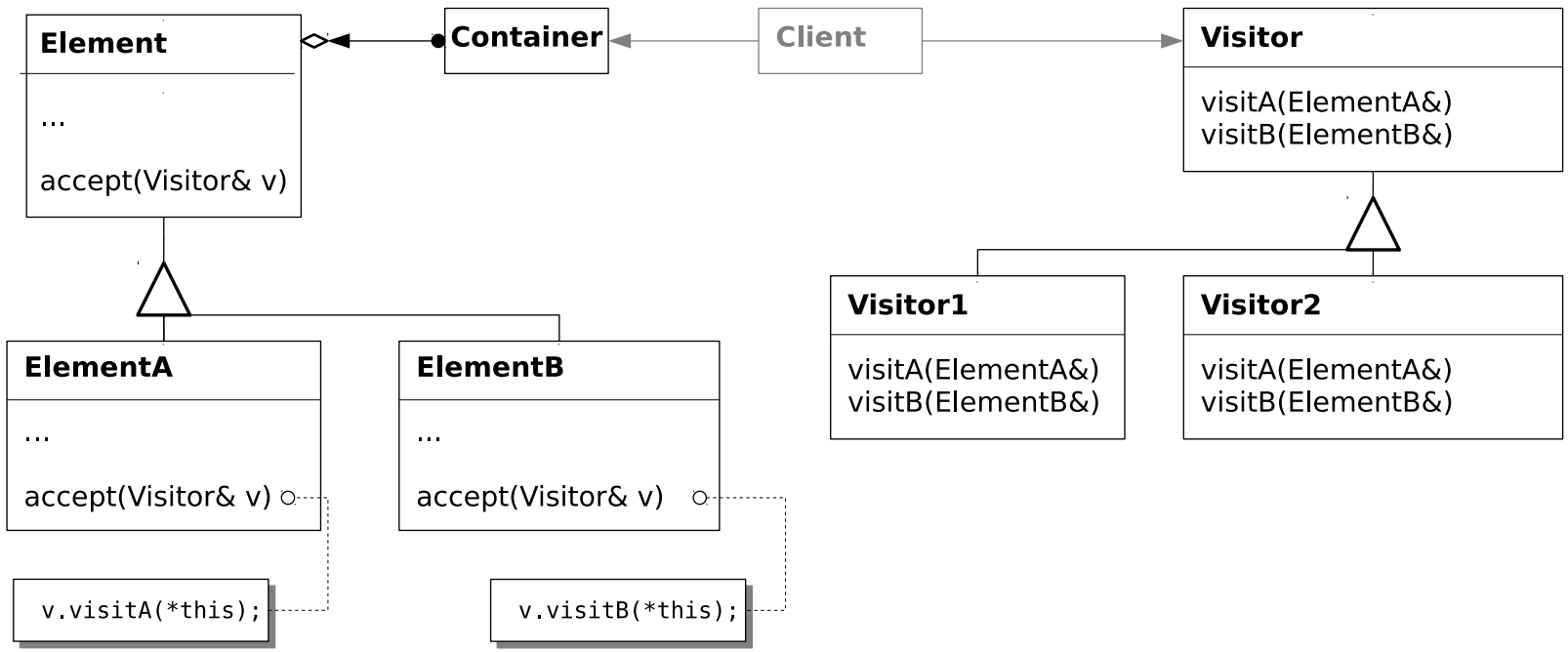


# Posjetitelj: primjenljivost i struktura

**Primjenljivost** -- polimorfni poziv koji ovisi o dva konkretna razreda

- operacije nad polimorfnim objektima ovise o konkretnim tipovima
- operacija ima mnogo, ne želimo ih uguravati u sučelje elemenata
- razredi elemenata su stabilni, operacije se mijenjaju i množe

Posjetitelj: **ponašajni** obrazac (lijeva porodica prima objekt desne porodice)



# Posjetitelj: sudionici

- **Posjetitelj** (Operation)
  - deklarira po jednu varijantu operacije za svaki konkretni objekt
- **Konkretni posjetitelj** (OperationPay)
  - modelira konkretnu operaciju: čuva kontekst, izvodi varijante te akumulira rezultat
- **Element** (Employee)
  - deklarira metodu `accept` koja prima refrencu na Posjetitelja
- **Konkretni element** (Manager)
  - izvodi metodu `accept`, poziva odgovarajuću metodu posjetitelja
- **Klijent(i)** (Client)
  - kreira odgovarajućeg posjetitelja, iterativno ga šalje elementima

# Posjetitelj: suradnja i posljedice

**Suradnja**:

- Klijent koji primjenjuje obrazac mora (i) instancirati konkretni posjetitelj te (ii) proći kroz sve elemente kolekcije
- Konkretni element prosljeđuje poziv `accept` konkretnom posjetitelju

**Posljedice**:

- lako dodavanje novih operacija definiranjem novog Posjetitelja pospješuje se jedinstvena odgovornost
- teško dodavanje novih Konkretnih elemenata: treba mijenjati sve posjetitelje! (nestabilna hijerarhija elemenata je kontraindikacija)
- Elementi moraju izložiti sve relevantne dijelove svog stanja
- Dobivamo sustav premrežen ovisnostima: svaki Konkretni posjetitelj ovisi o svakom Konkretnom elementu

# Posjetitelj: Implementacija

- pojedine metode posjetitelja mogu imati isti naziv (overload) ili različite nazive
- kompozit delegira poziv `accept` svojim elementima
- dvostruko prosljeđivanje $\Rightarrow$ ishod poziva ovisi o dva primatelja:
  - ishod poziva `accept` ovisi o: (i) Konkretnom elementu, (ii) Konkretnom posjetitelju
  - neki jezici DP podržavaju izravno (Lisp, Objective C, ...), neki preko ekstenzija (Python, Java, ...)
- prolaz kroz kolekciju elemenata može biti odgovornost klijenata, iteratora ili samog posjetitelja

# Posjetitelj: primjene i srodni obrasci

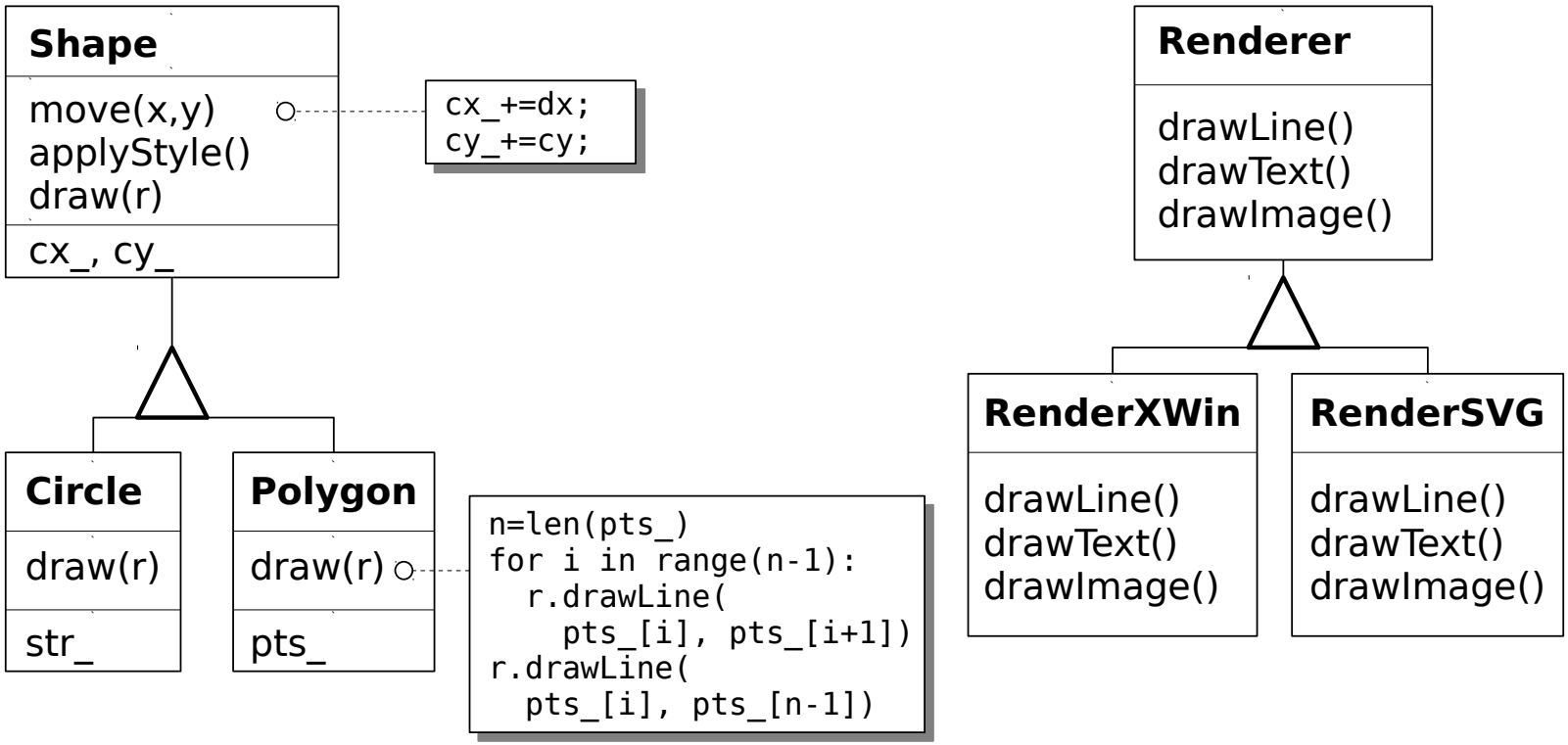
**Primjene:**

- Prevoditelji, za opis višestrukih operacija nad sintaksnom strukturom programa (istovremena ovisnost o semantici i arhitekturi)
- operacije nad elementima vektorskog crteža (iscrtavanje, pretraživanje, primjena atributa, eksportiranje za visio, openoffice, ms-office...)
- operacije nad stablom datotečnog sustava (pretraživanje, backup, brisanje, izvođenje skripti, ...)

**Srodni obrasci:**

- Posjetitelj se često koristi u kombinaciji s Kompozitom i Iteratorom
- Posjetitelj se razlikuje od Mosta po tome što:
  - konkretni elementi primaju posjetitelje preko argumenata metode `accept`, dok je kod mosta izvođač podatkovni član apstrakcije
  - konkretni posjetitelji znaju za konkretne elemente, dok konkretni izvođači ne znaju za prilagođene apstrakcije.

# Posjetitelj: Most vs Posjetitelj: rubni slučaj




Organizacija je slična Posjetitelju jer klijent mora znati za implementacijsku obitelj

Organizacija je slična Mostu jer implementacijska obitelj ne ovisi o sučeljnoj obitelji

# Metoda tvornica

Namjera: definirati sučelje za kreiranje objekta, ali izvedbu prepustiti izvedenim razredima

Engleski nazivi: factory method, virtual constructor

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;
  }
  //...
}
```

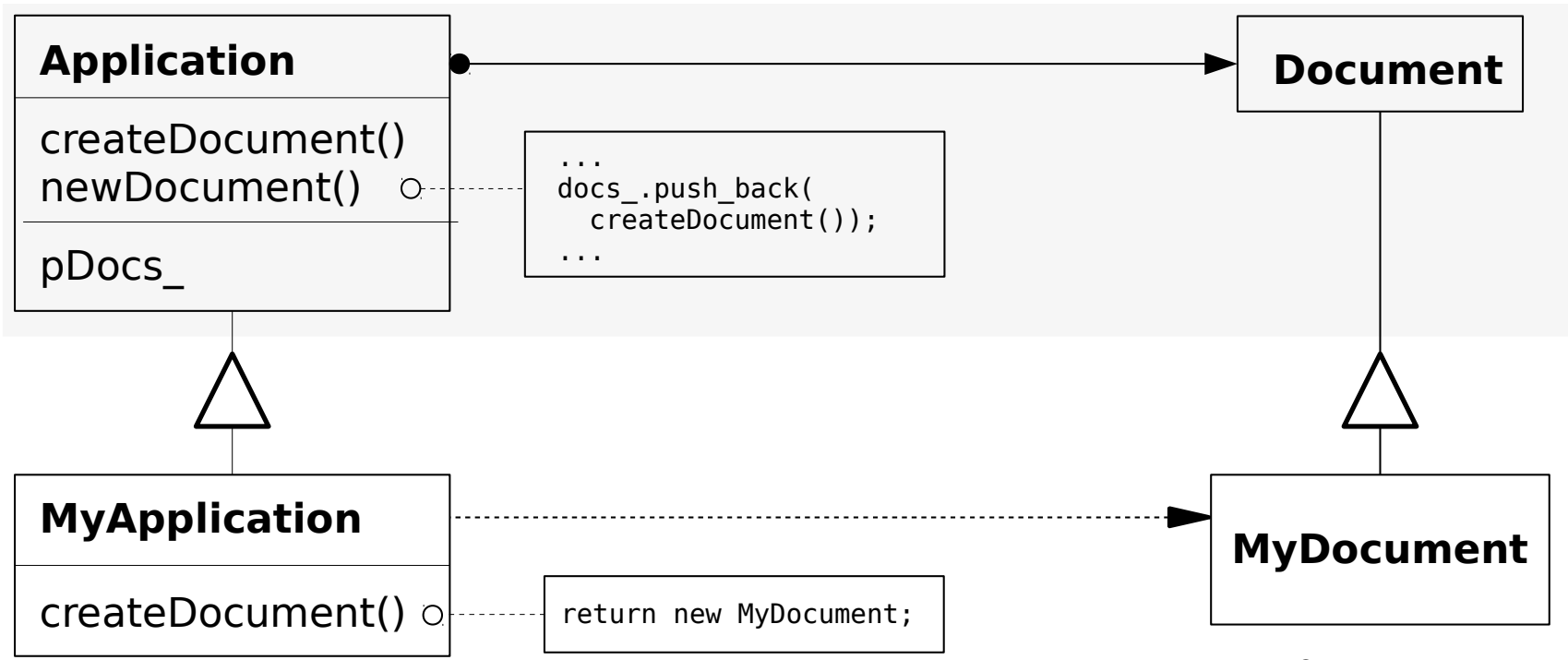
# Metoda tvornica: motivacija

Okvirni razredi (`Document`, `Application`) napisani prije korisničkih razreda

Korisnički razred `MyApplication` instanciran ručno

Metodu `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

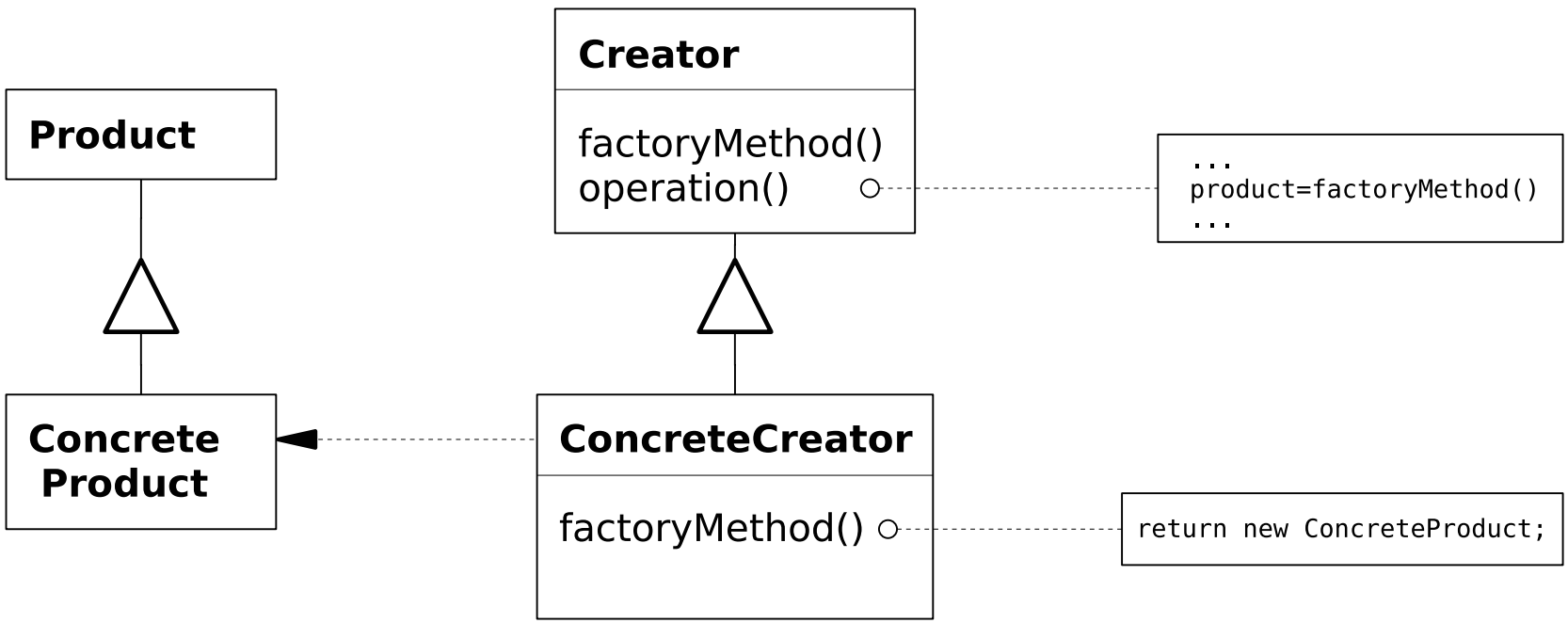
Obrazac metode tvornice koristimo kad:

- 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 (kreator stvara prikladne proizvode)

**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: implementacija, primjene

Kreator može ponuditi apstraktnu metodu tvornicu ili podrazumijevanu implementaciju

Metoda tvornica može biti parametrizirana

Metode tvornice je problematično zvati iz konstruktora apstraktnog Kreatora (zašto?)

Metode tvornice mogu se koristiti za povezivanje paralelnih hijerarhija razreda

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

# MVC: MOTIVACIJSKI PRIMJER

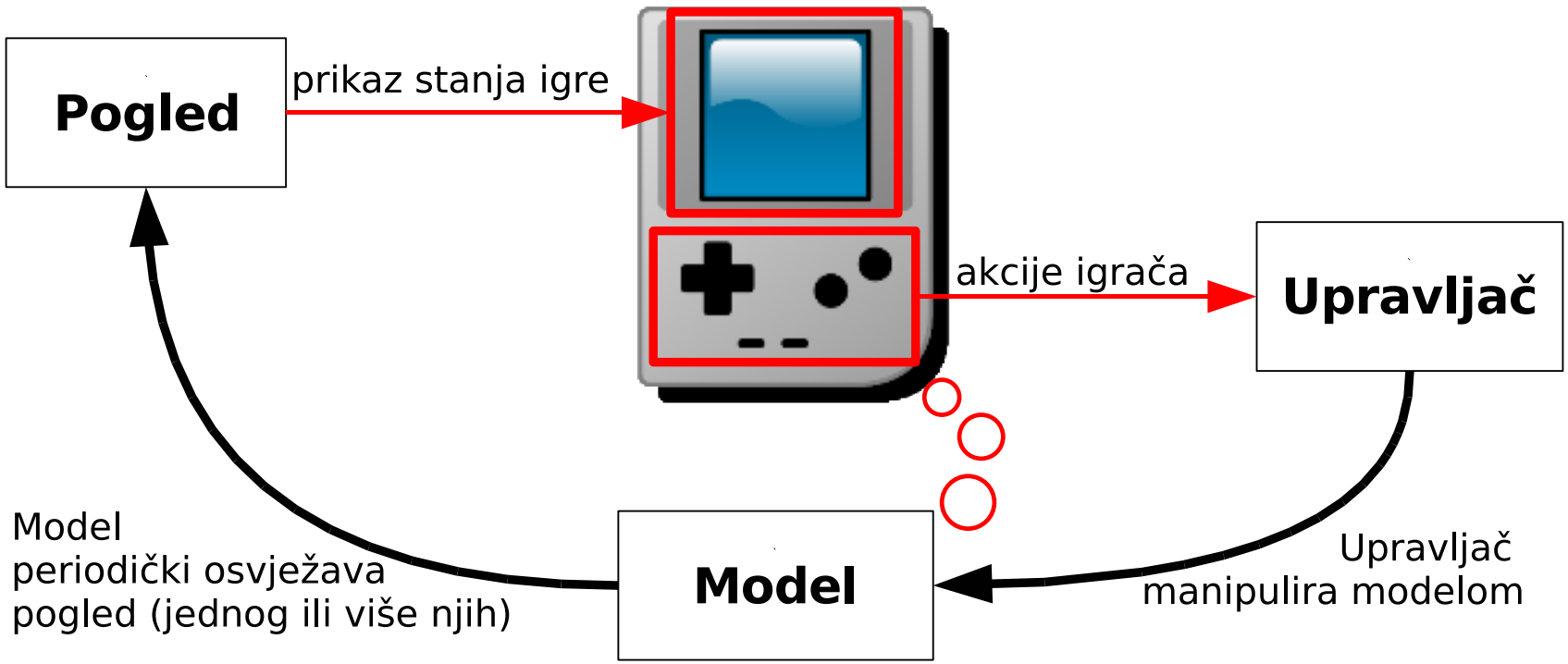
**Problem:**

- Razvijamo upravljački program za mp3 player
  - tri glavne komponente: računalni model, pogled, upravljač
  - **računalni model** obuhvaća aplikacijsku logiku: održavanje popisa pjesama, pristup datotečnom sustavu, dekompresiju podataka, ...
  - **pogled** određuje sadržaj zaslona i reproducira zvuk
  - **upravljač** poslužuje korisničke akcije zadane tipkama
- kako ograničiti međuovisnosti među tri osnovne komponente?

**Rješenje:**

- Složeni obrazac **model-pogled-upravljač** (model-view-controller): Xerox Parc 1979, Smalltalk
- United States Patent 5926177, IBM, predano 1997, odobreno 1999

# MVC: MOTIVACIJSKI PRIMJER



# MVC: namjera

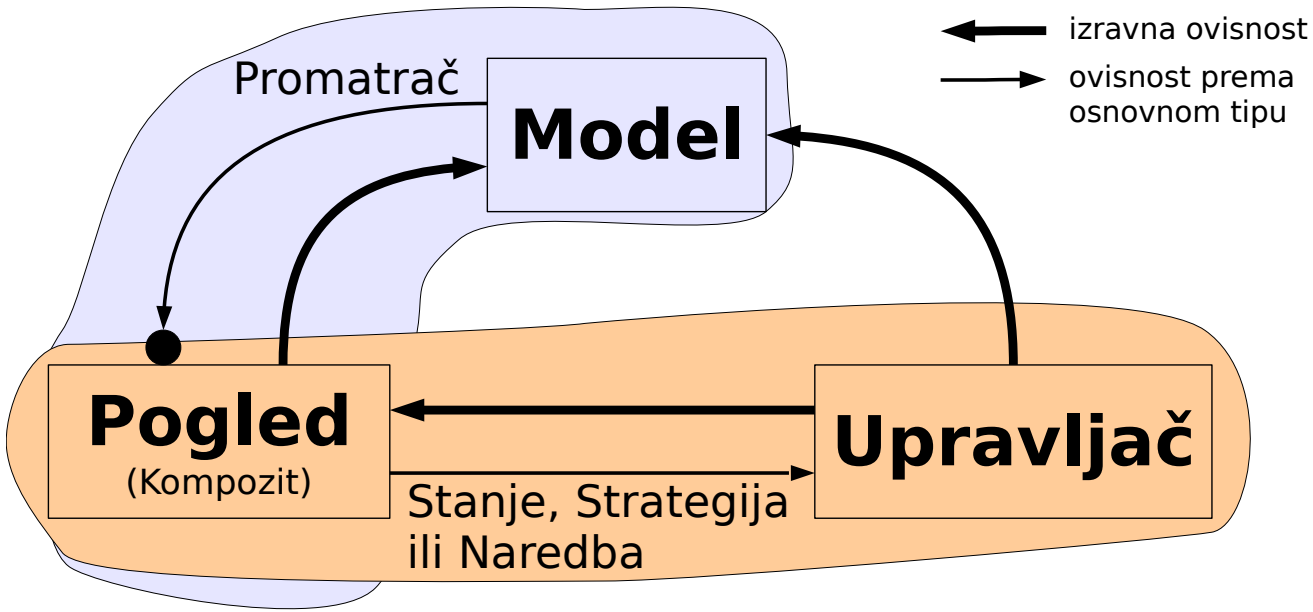
- Particionirati funkcionalnost interaktivne aplikacije u tri dijela: model (obrada), upravljač (ulaz), pogled (izlaz)
- Premostiti jaz između humanog mentalnog modela i digitalnog modela koji postoji u računalu:
  - podržati iluziju izravnog pogleda na elemente domene i njihovu manipulaciju
  - omogućiti da korisnik vidi model s različitih gledišta te utječe na njega preko različitih sučelja

# MVC: PRIMJENLJIVOST I STRUKTURA

**Primjenljivost** -- arhitektonski programski obrazac

- kad želimo međusobno izolirati logiku programa, korisničko sučelje i prikaz, s ciljem lakšeg održavanje programa
- prikladan i za složenije GUIjske aplikacije
- često kombinacija Promatrača, Stanja i Kompozita

MVC: **arhitektonski** obrazac, kombinacija Promatrača, Stanja, ...



# MVC: SUDIONICI I SURADNJA

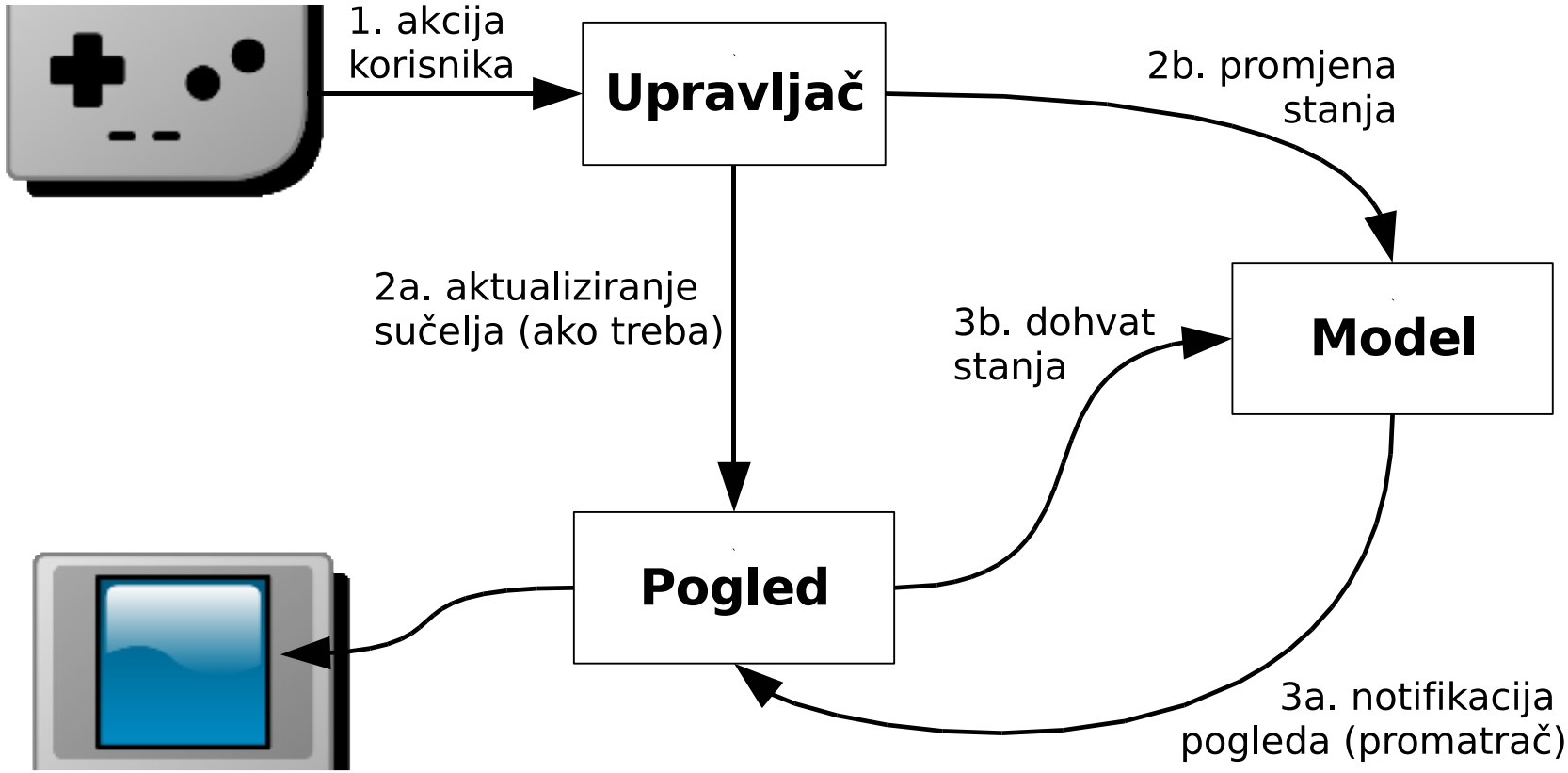
**Sudionici**:

- **Model**
  - sadrži podatke aplikacije i pravila za njihovu manipulaciju
- **Pogled** (View)
  - prikazuje aplikacijske podatke na koristan i praktičan način; može prikazivati i elemente guija
  - može biti više pogleda, istovremeno prikazanih ili ne
- **Upravljač** (Controller)
  - odgovoran za komunikaciju s korisnikom (obrada tipkovnice, miša, ...)

**Suradnja**:

- Upravljač radi sinkrono s korisnikom, šalje korisničke akcije modelu (i pogledu)
- Model evoluira ovisno o korisničkim akcijama (model može evoluirati i neovisno o korisniku)
- Pogled prikazuje elemente modela sinkrono s modelom, te elemente GUIja sinkrono s korisnikom

# MVC: SURADNJA



# MVC: posljedice

- Smanjuje se arhitektonska složenost odvajanjem modela od upravljača i pogleda
- rezultat je konfigurabilna i podatna aplikacija s grafičkim sučeljem (*skinnable!*)
- mogućnost višestrukih i usporednih pogleda u stanje programa (Promatrač)
- mogućnost različite obrade korisničkih akcija (Strategija)
- nedostatak: složenost može ne biti opravdana
  (npr, pogled možda može biti neovisan o modelu)

# MVC: Implementacija

- Upravljač može ujedno biti i jedan od promatrača
- elementi upravljača često implementirani automatski (dvosjekli mač)
- upravljač može biti integriran s pogledom (arhitektura Document - View)
- u ambicioznijim aplikacijama upravljač i model rade usporedno: potreba za sinkronizacijskim mehanizmima

# MVC: PRIMJENE

**Primjene:**

- Rasprostranjena arhitektura za konfigurabilne programe s grafičkim sučeljem
- Brojne biblioteke za građenje GUI i web aplikacija:
  `http://en.wikipedia.org/wiki/Model-view-controller`
- Klasične aplikacije za web (npr. kućno bankarstvo) koriste sljedeću varijantu MVC-a:
  - pogled: dinamički generirana HTML stranica
  - upravljač: skripta koja skuplja podatke od korisnika i u skladu s modelom generira HTML
  - model: stvarni podatci spremljeni u udaljenoj bazi podataka
  - za razliku od desktop varijante, ovdje pogled generira upravljač:
    - ◊ model se ne mijenja bez akcije korisnika
    - ◊ u slučaju duže neaktivnosti aplikacija može prekinuti transakciju

# Zaključak: Obrasci

- Programski obrazac je **rješenje** problema u **konktekstu**
- Programski obrasci su akumulirano znanje o čestim problemima
- Programski obrasci nisu *silver bullet*!!
  (nemojte biti arhitekti-astronauti)
- Pravi pristup: biti **sposoban** uočiti obrazac tamo gdje on prirodno liježe
- Prednost vokabulara: na sastancima, u neformalnoj komunikaciji, dokumentaciji i komentarima, ...

# 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, Win32, ...)

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();
}
```

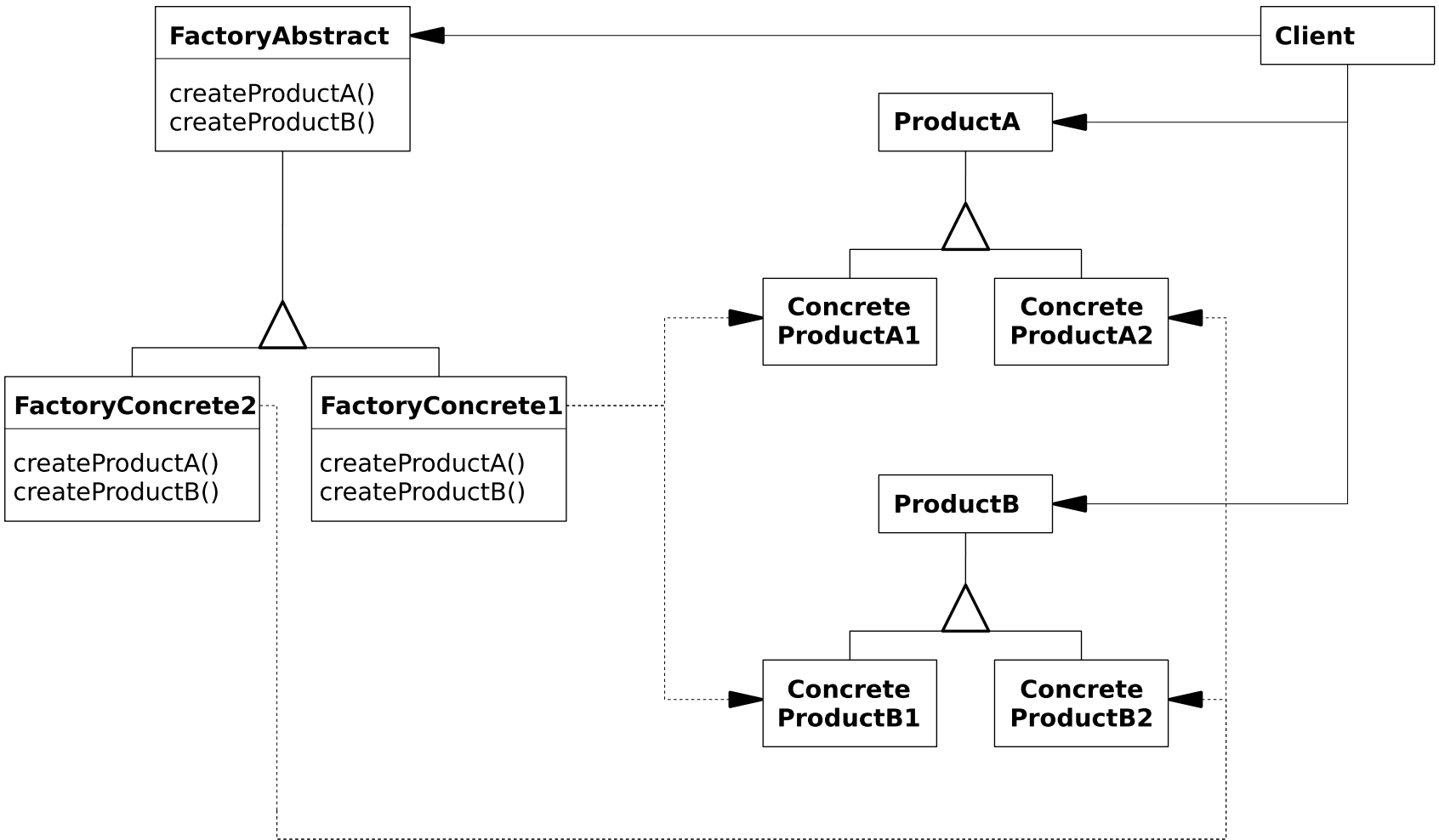
# Apstraktna tvornica: primjenljivost

- 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 (tvornica stvara objekte prikladne porodice)

**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)