

Oblikovni obrasci u programiranju

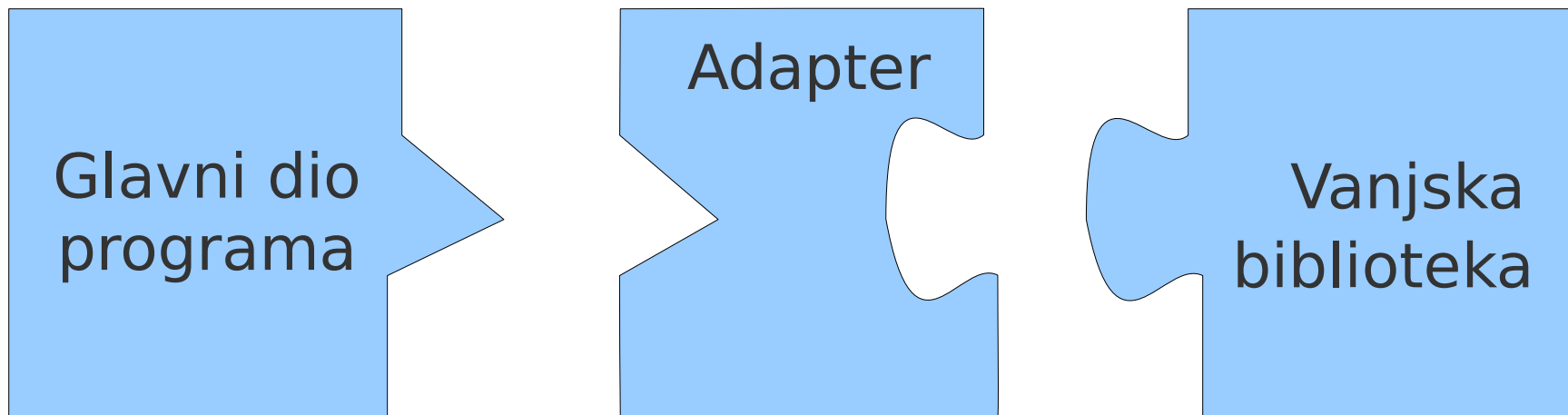
još rješenja učestalih oblikovnih problema

Siniša Šegvić

Zavod za elektroniku, mikroelektroniku,
računalne i inteligentne sustave
Fakultet elektrotehnike i računarstva
Sveučilište u Zagrebu

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

⇒ 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

Glavna **ideja** obrasca:

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

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

PRILAGODNIK: PRIMJENLJIVOST

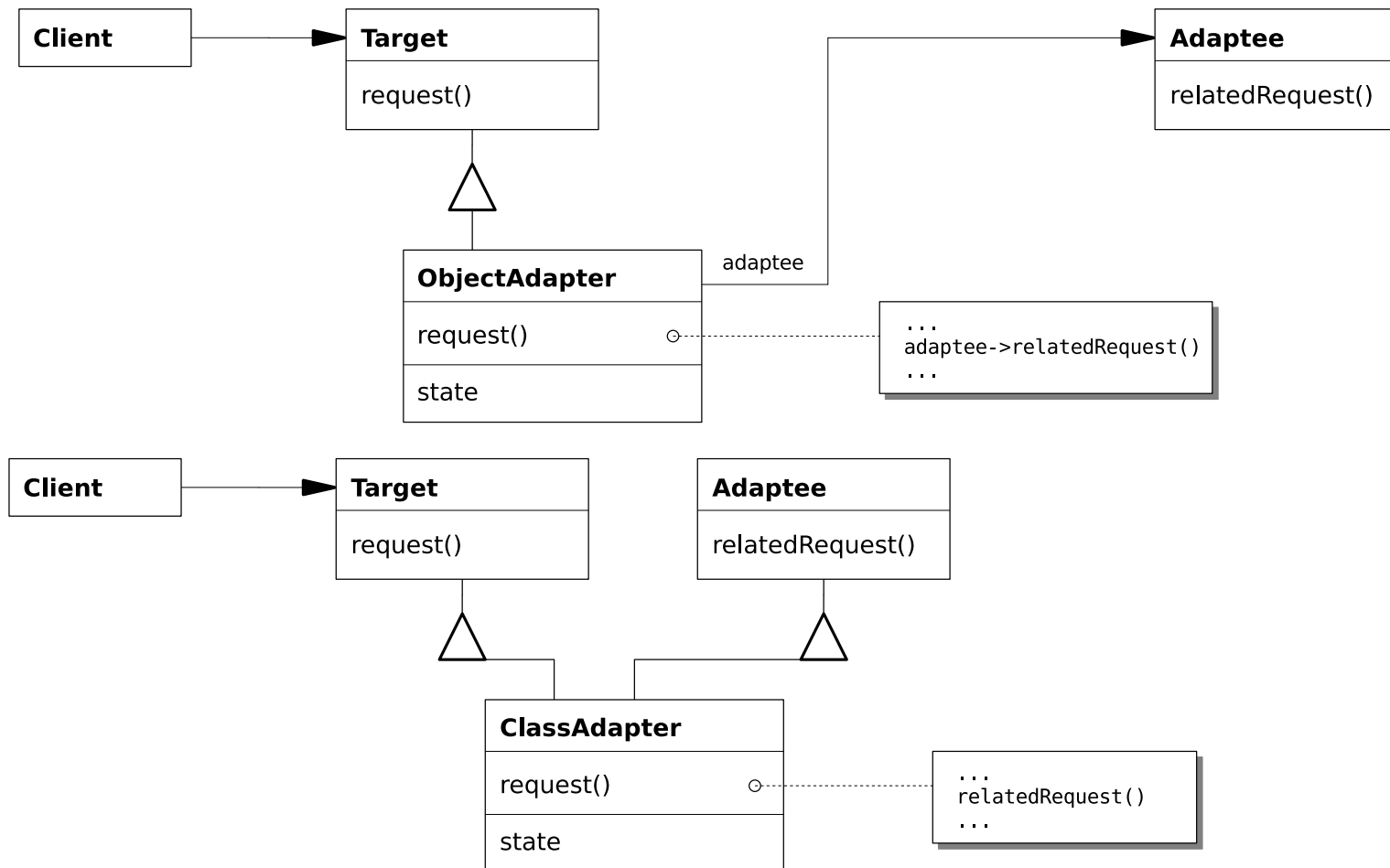
Prilagodnike koristimo ako vrijedi bilo što od sljedećeg:

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

PRILAGODNIK: STRUKTURNI DIJAGRAM

Prilagodnik (adapter, wrapper): **strukturni** obrazac u domeni **objekata** (ili razreda)

Prilagodljivost se ostvaruje povjeravanjem (ili nasljeđivanjem)



PRILAGODNIK: SUDIONICI I SURADNJA

Sudionici:

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

Suradnja:

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

PRILAGODNIK: POSLJEDICE

Potporna 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 sve razrede izvedene iz Vanjskog razreda

Prednosti razrednog prilagodnika:

- mogućnost utjecanja na Vanjski razred nadomjesnim polimorfnim funkcijama
- rukujemo samo s jednim objektom

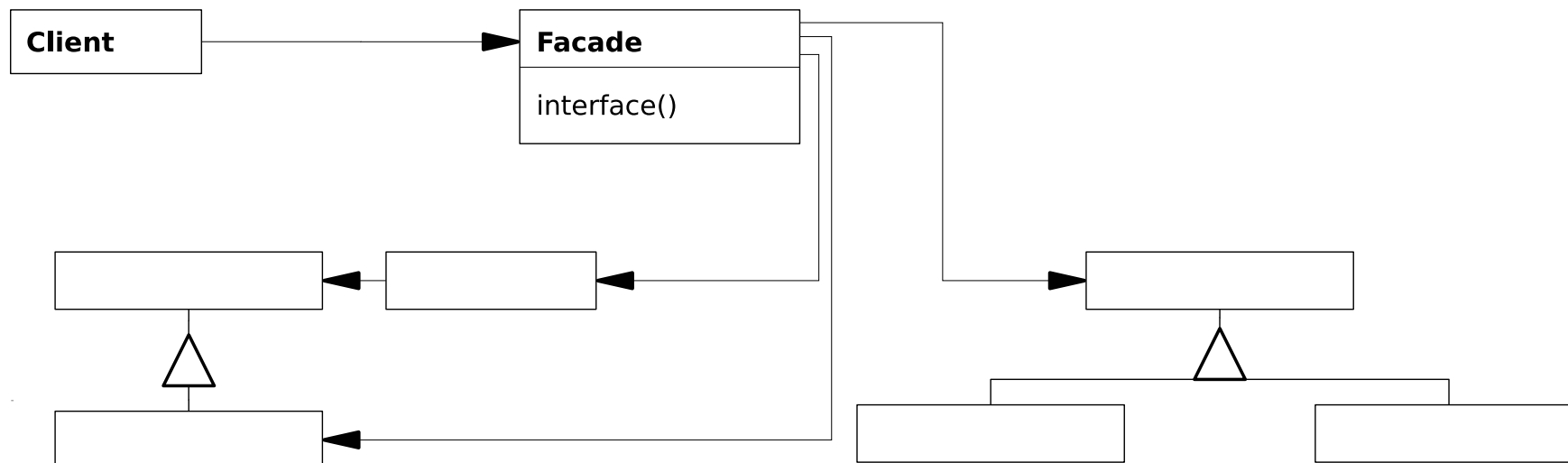
PRILAGODNIK: IMPLEMENTACIJA

Implementacija:

- Izvedba razrednog Prilagodnika:
 - javno nasljeđivanje Ciljnog razreda
 - privatno nasljeđivanje Vanjskog razreda
- Izvedba objektnog Prilagodnika:
 - javno nasljeđivanje Ciljnog razreda
 - 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



OKVIRNA METODA: NAMJERA

Definirati okvirni postupak koji neke korake prepušta izvedenim razredima.

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

U usporedbi sa **Strategijom** okvirna metoda je jednostavnije i manje općenito rješenje

OKVIRNA METODA: MOTIVACIJSKI PRIMJER

```
class GameWithTemplateMethod {
    //...
protected:
    virtual void init() =0;
    virtual void makeMove() =0;
    virtual bool endOfGame() =0;
    virtual void printWinner() =0;

public:
    // A template method:
    void playOneGame() {
        init();
        while (!endOfGame()){
            makeMove();
        }
        printWinner();
    }
};
```

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

OKVIRNA METODA: PRIMJENLJIVOST

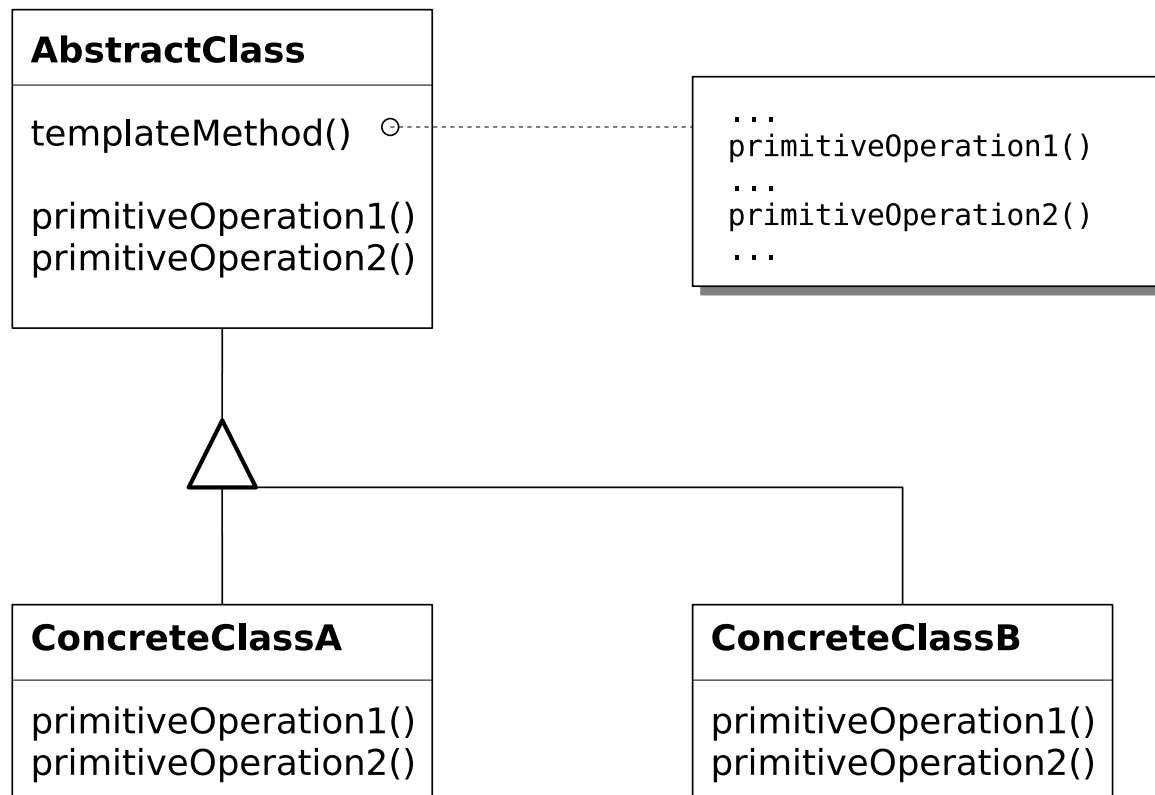
Okvirna metoda koristi se kad:

- stalne dijelove algoritma valja prikupiti na jednom mjestu, dok izvedeni razredi definiraju promjenljive korake: istu grubu strukturu postupka (**okvir**) dijele više izvedenih razreda
- zajedničko ponašanje izvedenih razreda želimo izdvojiti da izbjegnemo dupliciranje kôda
- izvedenim razredima treba omogućiti proširivanje funkcionalnosti:
 - okvirna metoda poziva tzv. metode za proširenje (*engl.* hooks)
 - osnovni razred pruža praznu podrazumijevanu implementaciju

OKVIRNA METODA: STRUKTURNI DIJAGRAM

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

Prilagodljivost se ostvaruje nasljeđivanjem



OKVIRNA METODA: SUDIONICI I SURADNJA

Sudionici:

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

Suradnja:

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

OKVIRNA METODA: POSLJEDICE I IMPLEMENTACIJA

Posljedice:

- Fundamentalna tehnika višestrukog korištenja, korisna za izdvajanje zajedničkog kôda u bibliotekama razreda
- Pospješuje se inverzija ovisnosti jer klijenti okvirnu metodu mogu pozivati preko osnovnog sučelja

Implementacija:

- Okvirne metode pozivaju apstraktne primitive i metode za proširenje
 - primitivi se *moraju* izvesti, za razliku od proširenja
 - potrebno **dokumentirati** što je što, najbolje u okviru jezika:
 - ◇ okvirna metoda - običan poziv
 - ◇ primitivi - zaštićene čiste virtualne metode
 - ◇ proširenja - zaštićene virtualne metode, prazna implementacija
- C++: **protected**, **virtual**, =0; Java: **protected**, **final**

OKVIRNA METODA: PRIMJENE I SRODNI OBRASCI

Primjene:

- Vrlo raširena u praksi, kao najjednostavniji OO oblik višestrukog korištenja

Srodni obrasci:

- Metode tvornice se najčešće pozivaju iz okvirnih metoda (tvornica tad igra ulogu primitivne operacije)
- Strategija kao općenitije ali i složenije rješenje umjesto nasljeđivanja koristi delegaciju

ITERATOR: NAMJERA

Omogućiti uniformirani slijedni pristup elementima skupnog objekta bez otkrivanja njegove interne strukture

Više od indeksiranja **brojačem**: podržane proizvoljne strukture podataka

Generalizacija **pokazivača**, enkapsulira se:

1. slijedni prolaz kroz spremnik objekata,
2. pristup tekućem elementu

ITERATOR: MOTIVACIJSKI PRIMJER

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 C;
C.insert(std::make_pair("skrpina",5)); //škrpina je riba za pet!
for (MyMap::const_iterator it = C.begin(); it != C.end(); ++it){
    std::cout <<it->first <<" " <<it->second <<"\n";
}
```

Implicitni iteratori u Javi:

```
Map<String, Integer> mymap = // zapravo prevoditelj napravi:
    new HashMap<String, Integer>();
mymap.put("skrpina", 5);
// mi pišemo:
for (Map.Entry e: mymap.entrySet()){
    System.out.println(
        e.getKey()+" "+e.getValue());
}
Iterator it = mymap.
    entrySet().iterator();
while (it.hasNext()) {
    Map.Entry e = (Map.Entry)it.next();
    System.out.println(
        e.getKey()+" "+e.getValue());
}
```

Implicitni iteratori u Pythonu:

```
mymap=['škrpina':5,'tabinja':3]
# mi kažemo:
for x in mymap:
    print(x, mymap[x])
# zapravo, izvodi se kod desno
it = d.__iter__() # kraće: iter(L)
try:
    while True:
        key=it.__next__() # kraće: next(it)
        print(key, d[key])
except StopIteration: pass
```

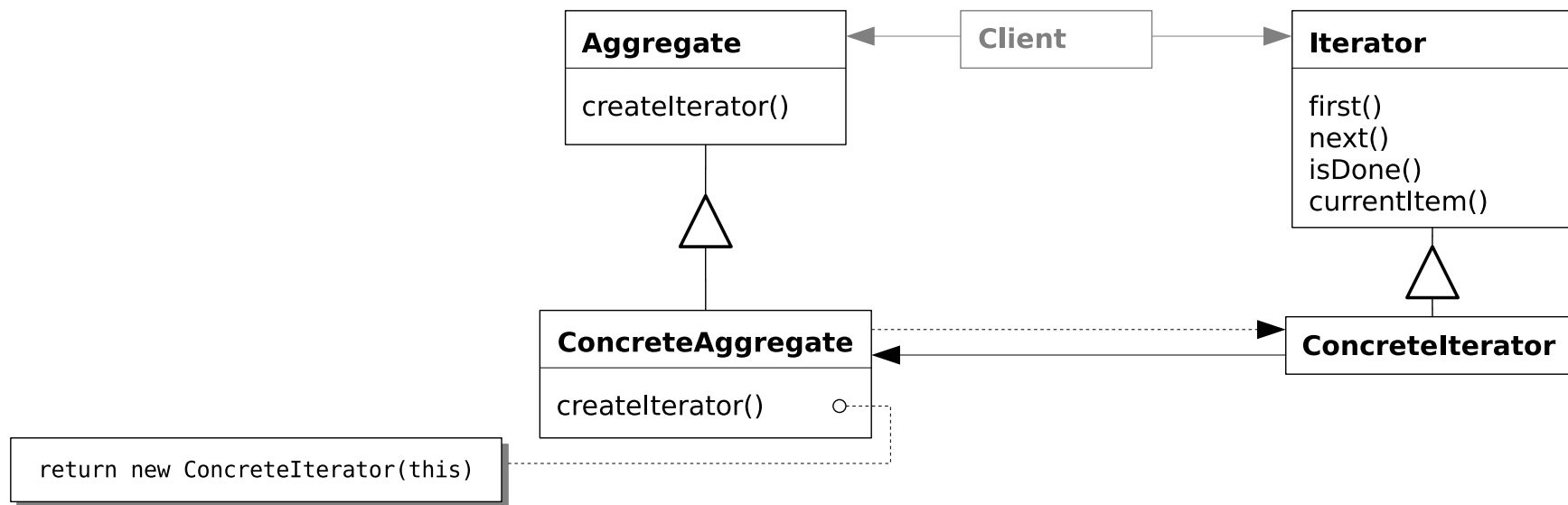
ITERATOR: PRIMJENLJIVOST

Iterator se koristi za postizanje:

- pristupa sadržaju skupnog objekta neovisno o njegovoj strukturi
- mogućnosti višestrukih usporednih prolaza kroz skupni objekt
- uniformnog polimorfnog sučelja za prolaz kroz različite skupne objekte
- u različitim jezicima koriste se različite varijante polimorfizma (C++: statički, Java: dinamički)
- prednosti se očituju i ako nema potrebe za polimorfnim pristupom (jasnija upotreba)

ITERATOR: STRUKTURNI DIJAGRAM

Iterator (Cursor): **ponašajni** obrazac u domeni objekata



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)

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!)
- C++ nema dedikiranu sintaksu za implicitnu iteraciju:

```
class f{
public:
    void operator()(string s){
        cout <<s <<"\n";}
};
// molim vas tri bureka:
vector<string> V(3,"burek");

for_each(V.begin(), V.end(), f());
```

```
List<String> mylist =
    new ArrayList<String>();

mylist.add(
    new String("burek"));

for (String s: mylist){
    System.out.println("Item: " + s);
}
```

ITERATOR: IMPLEMENTACIJA (2)

- Iteratori s različitim mogućnostima, npr STL: Forward, Bidirectional (list) i Random Access (vector, za binary_search)
- Iteratori s dinamičkim polimorfizmom impliciraju sporiji prolaz
 - može biti bitno, iako najčešće nije
 - STL ne koristi polimorfne iteratore
- U STL-u, spremnici definiraju:
 - tipove iterator i const_iterator
 - metode begin() i end()
- U STL-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);
```

ITERATOR: PYTHON (1)

Generatori: iteriranje po slijedovima koji ne zauzimaju memoriju!

Želimo implementirati varijancu korištenjem funkcije `sum()`:

```
# sum iterates over its argument
# in the same manner as the for command
def variance(L):
    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
```

Da bismo izbjegli kreiranje privremenog objekta, oblikujemo novi razred:

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

# variance2 uses a generator class
# to avoid a temporary list
def variance2(L):
    mean=sum(L)/len(L)
    mygen=SqrdevGenerator(L,mean)
    return sum(mygen)/len(L)

# can we make SqrdevGenerator
# less verbose?
```

Razred `SqrdevGenerator` istovremeno implementira sučelja skupnog objekta i iteratora, i to računanjem elemenata slijeda na zahtjev!

Ovo je primjer tehnike lijenog evaluiranja (lazy evaluation).

ITERATOR: PYTHON (2)

Za poboljšanje jasnoće i konciznosti Python uvodi generatorske funkcije i generatorske izraze.

SqrdevGenFun stvara objekt ekvivalentan primjerku SqrdevGeneratora:

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

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

# yet another generator function
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)
```

Identičan rezultat možemo još kraće dobiti generatorskim izrazom!

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

ITERATOR: PRIMJENE I SRODNI OBRASCI

Primjene:

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

Srodni obrasci:

- Polimorfni iteratori se obično kreiraju u metodi tvornici konkretnog skupnog objekta

KOMPOZIT: NAMJERA I MOTIVACIJA

Namjera:

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

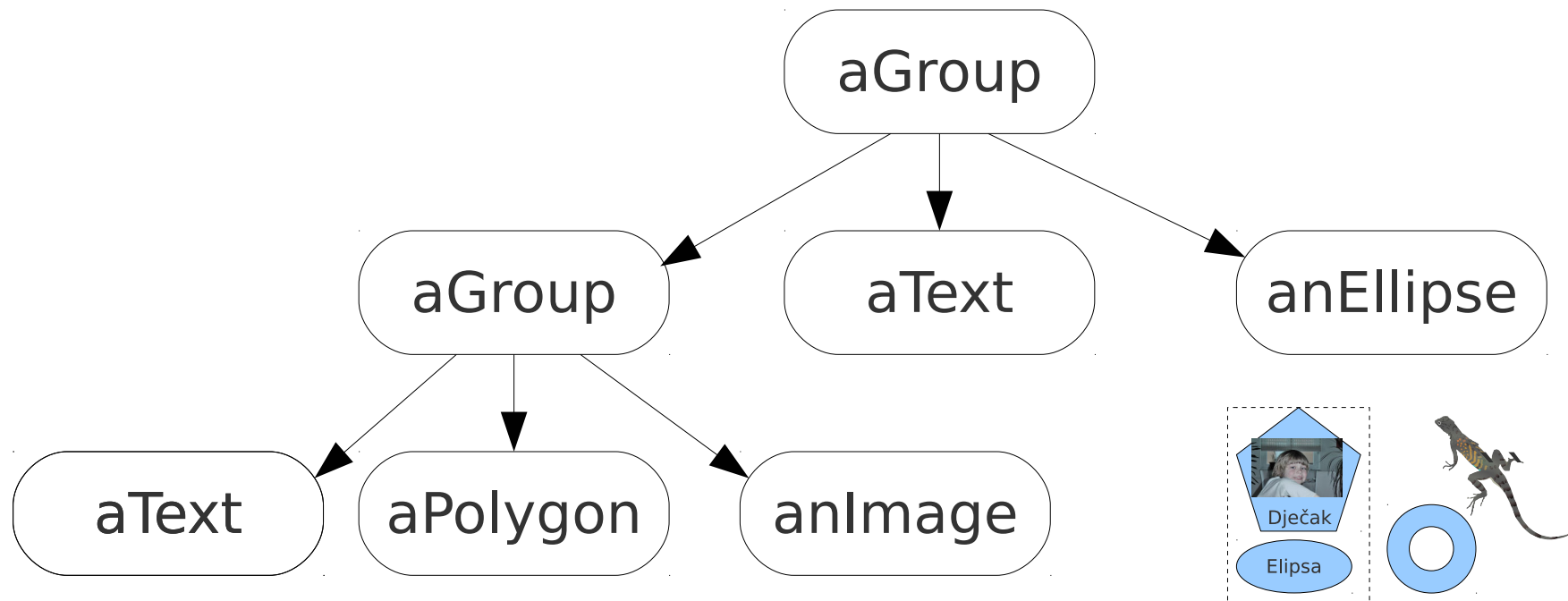
Ukomponirati objekte u rekurzivnu hijerarhiju sačinjenu od pojedinačnih objekata i njihovih grupa

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 se delegiraju sastavnim dijelovima!

KOMPOZIT: MOTIVACIJSKI PRIMJER

Struktura tipičnog kompozitnog objekta:

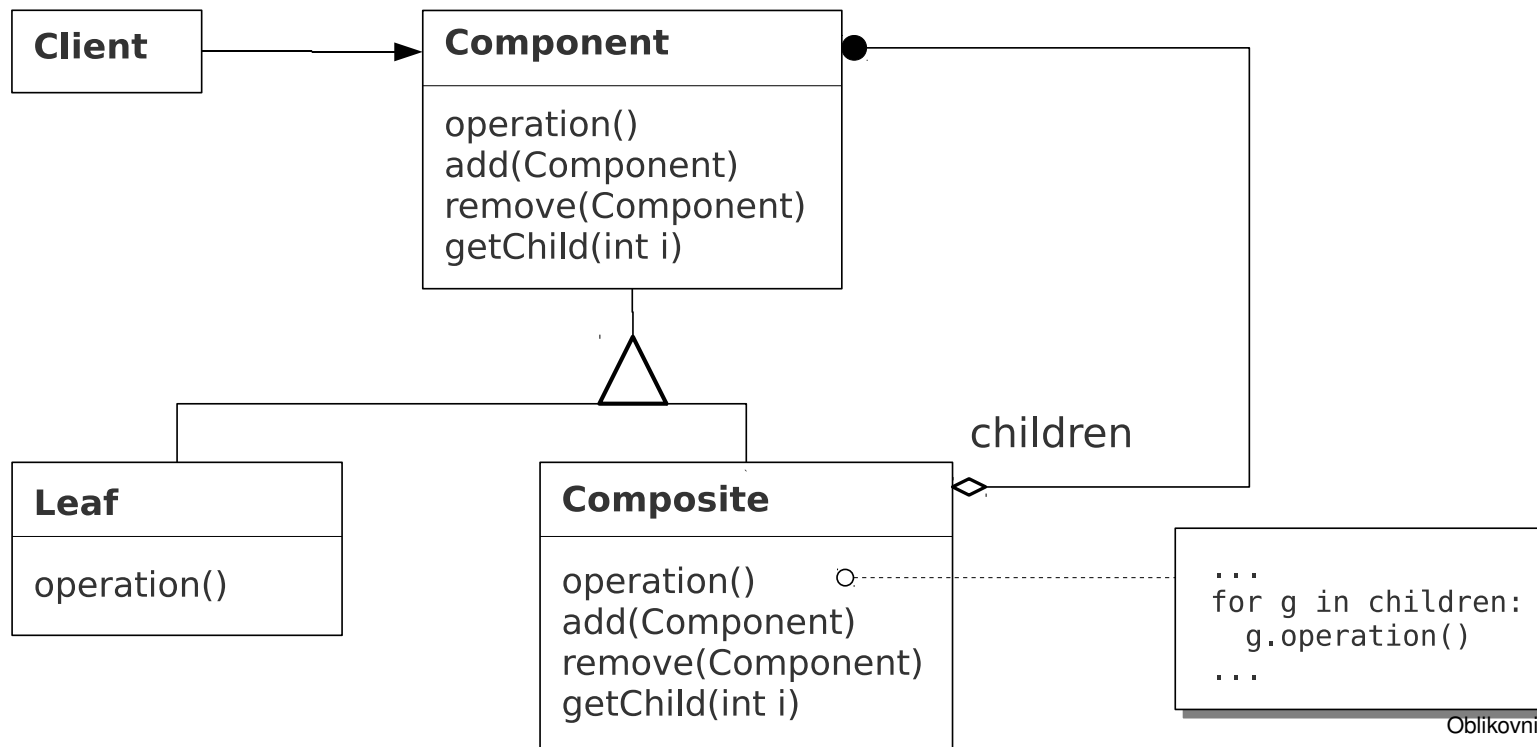


KOMPOZIT: PRIMJENLJIVOST I STRUKTURA

Primjenljivost – Kompozit se koristi za:

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

Kompozit (composite): **strukturni** obrazac u domeni **objekata**, temeljna ideja – rekurzivna kompozicija



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 komponenti
- **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

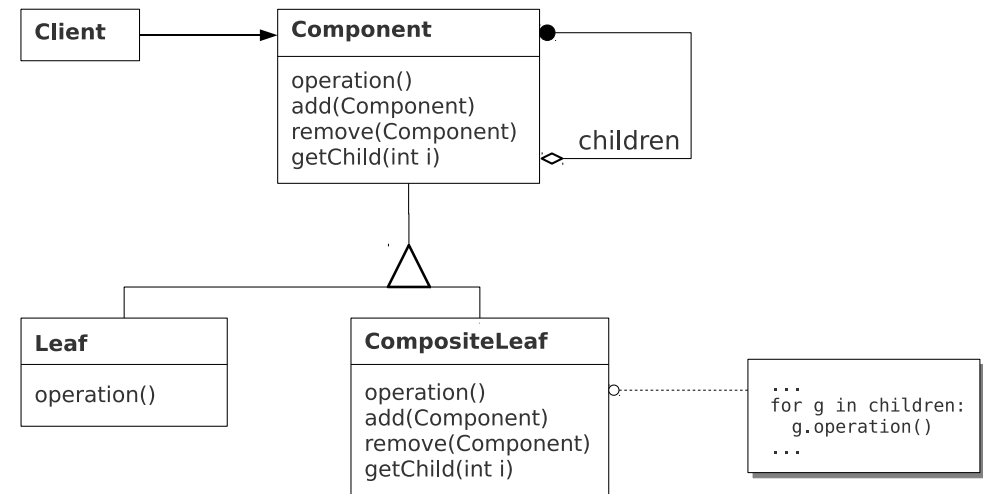
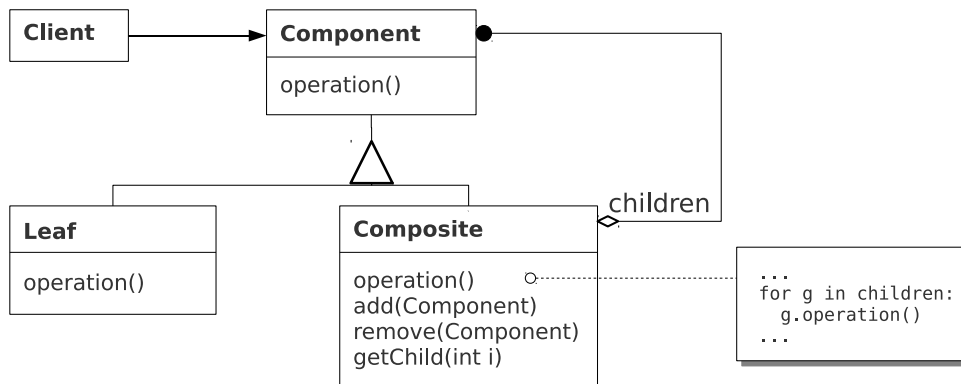
- Definira se hijerarhija koja se sastoji od Primitiva i Kompozita: gdje god klijent radi s Primitivima može primiti i Kompozite
- pojednostavnjuju se odgovornosti klijenta jer se Primitivi i Kompoziti mogu tretirati na jednak način
- olakšano dodavanje novih Komponenti: novi Kompoziti i Primitivi automatski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

KOMPOZIT: IMPLEMENTACIJA

- Eksplicitne reference na roditelja:
 - olakšavaju slijedni prolaz kroz hijerarhiju
 - logično mjesto za implementaciju je Komponenta
 - referenca se postavlja i briše pri dodavanju odnosno povlačenju Komponente iz Kompozita roditelja
- Kompoziti mogu cacheirati rezultate operacija kako bi ubrzali izvođenje budućih zadataka: npr, pamćenje pravokutnika opisanog djeci može ubrzati selekciju
- upravljanje sastavnim dijelovima kompozita može biti definirano u:
 - Komponenti (kao i u strukturnom dijagramu)
 - ◇ podrazumijevane implementacije metoda `add`, `remove`, `getChild` bacaju iznimku!
 - Kompozitu, uz veći integritet i manju transparentnost

KOMPOZIT: IMPLEMENTACIJA (2)

Različite mogućnosti smještanja metoda koje upravljaju elementima kompozita:



KOMPOZIT: PRIMJENE I SRODNI OBRASCI

Primjene:

- Gdje god su potrebne rekurzivne hijerarhije: datotečni sustav, aritmetički 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
- Makro-naredbe mogu biti kompoziti.

STANJE: NAMJERA I MOTIVACIJA

Namjera:

Omogućiti dinamičko mijenjanje ponašanja objekta

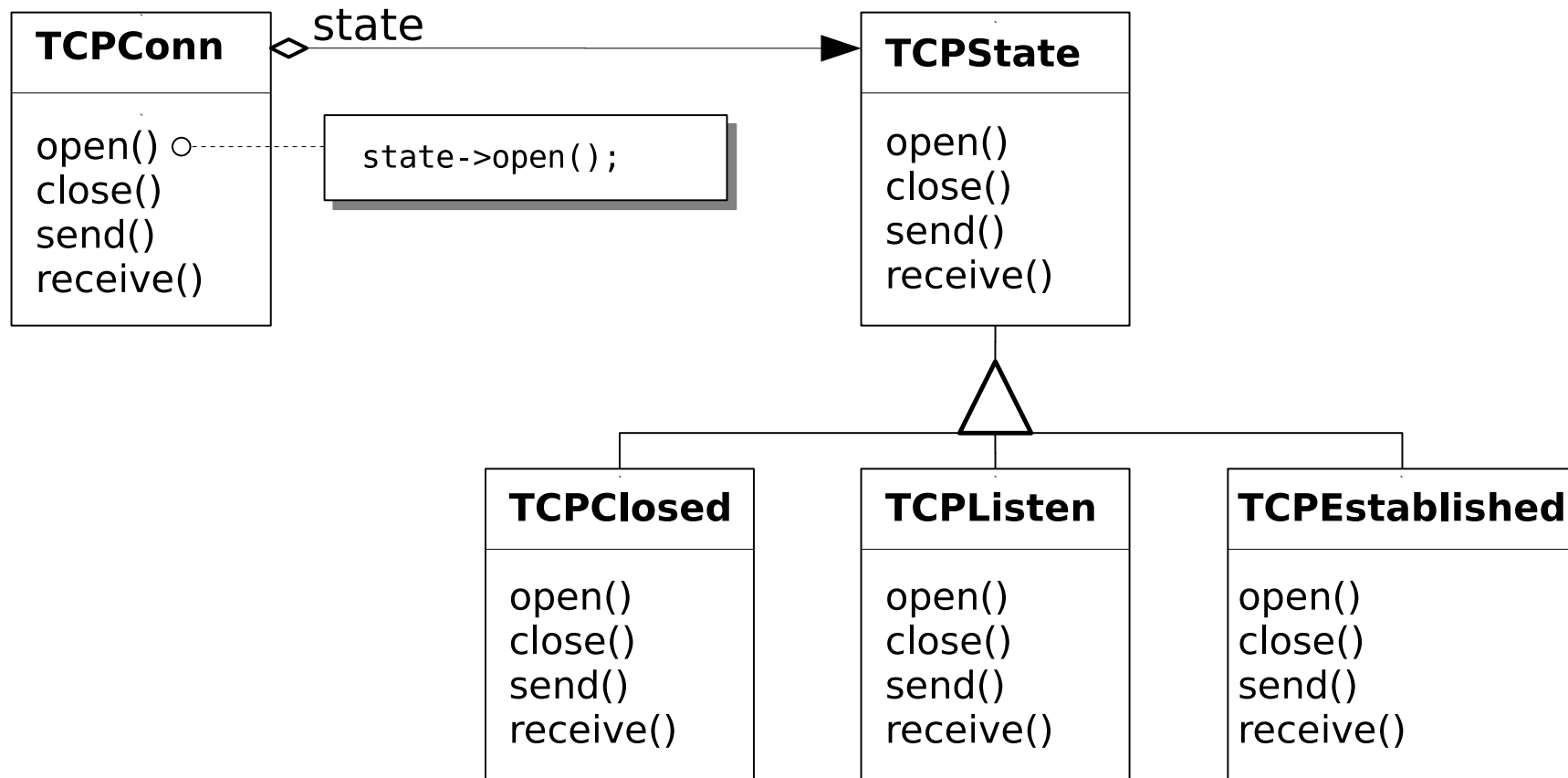
Objektno-orijentirana implementacija konačnog stroja

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

Motivacijski primjer:

- Ponašanje objekta u mnogome ovisi o tekućem stanju;
npr, funkcionalnost `TCPConnection::send()` ovisi o tome da li je TCP veza uspostavljena ili ne
- Ideja: funkcije čije ponašanje ovisi o stanju izdvojiti u poseban apstraktni osnovni razred
- Ponašanje u okviru svakog zasebnog stanja definirati odgovarajućim konkretnim razredom

STANJE: MOTIVACIJSKI PRIMJER

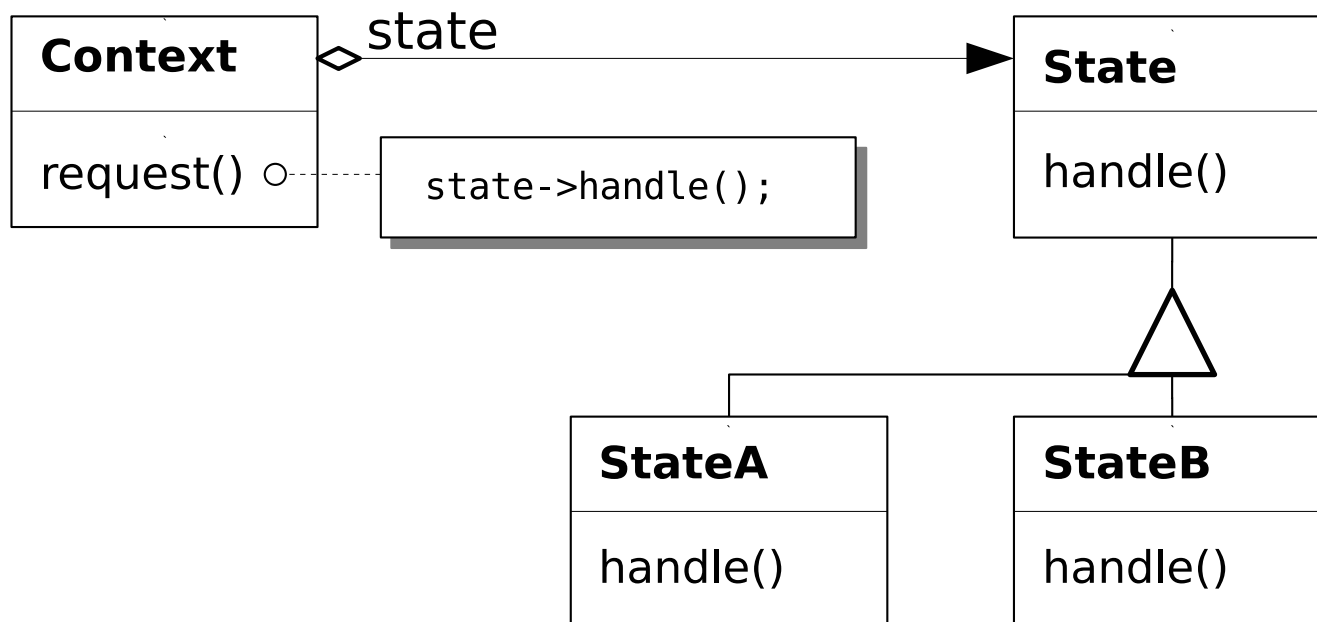


STANJE: PRIMJENLJIVOST I STRUKTURA

Primjenljivost – Stanje 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

Stanje (state): **ponašajni** obrazac u domeni **objekata**



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

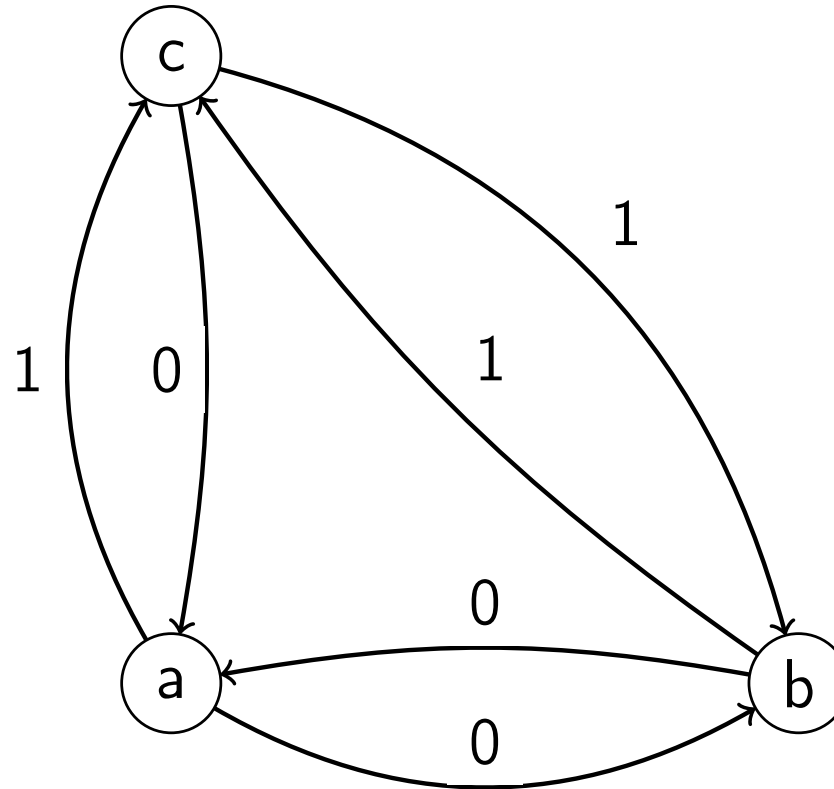
- Tko definira prijelaze stanja?
 - najčišće je prijelaze definirati u okviru Konteksta
 - ako to nije prikladno, Kontekst pruža sučelje za promjenu stanja kojeg pozivaju Konkretna stanja
 - ◇ **totalna međuovisnost** Konkretnih stanja (tvornica, registar?)
- alternativni pristup, izgradnja eksplicitne 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 StateB:
        def state(self): return "B"
        def event0(self):
            self.__class__=StateMachine
        def event1(self):
            self.__class__=self.StateC
    class StateC:
        def state(self): return "C"
        def event0(self):
            self.__class__=StateMachine
        def event1(self):
            self.__class__=self.StateB
    def state(self): return "A"
    def event0(self):
        self.__class__=self.StateB
    def event1(self):
        self.__class__=self.StateC
```

```
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

PROXY: NAMJERA I MOTIVACIJA

Namjera:

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

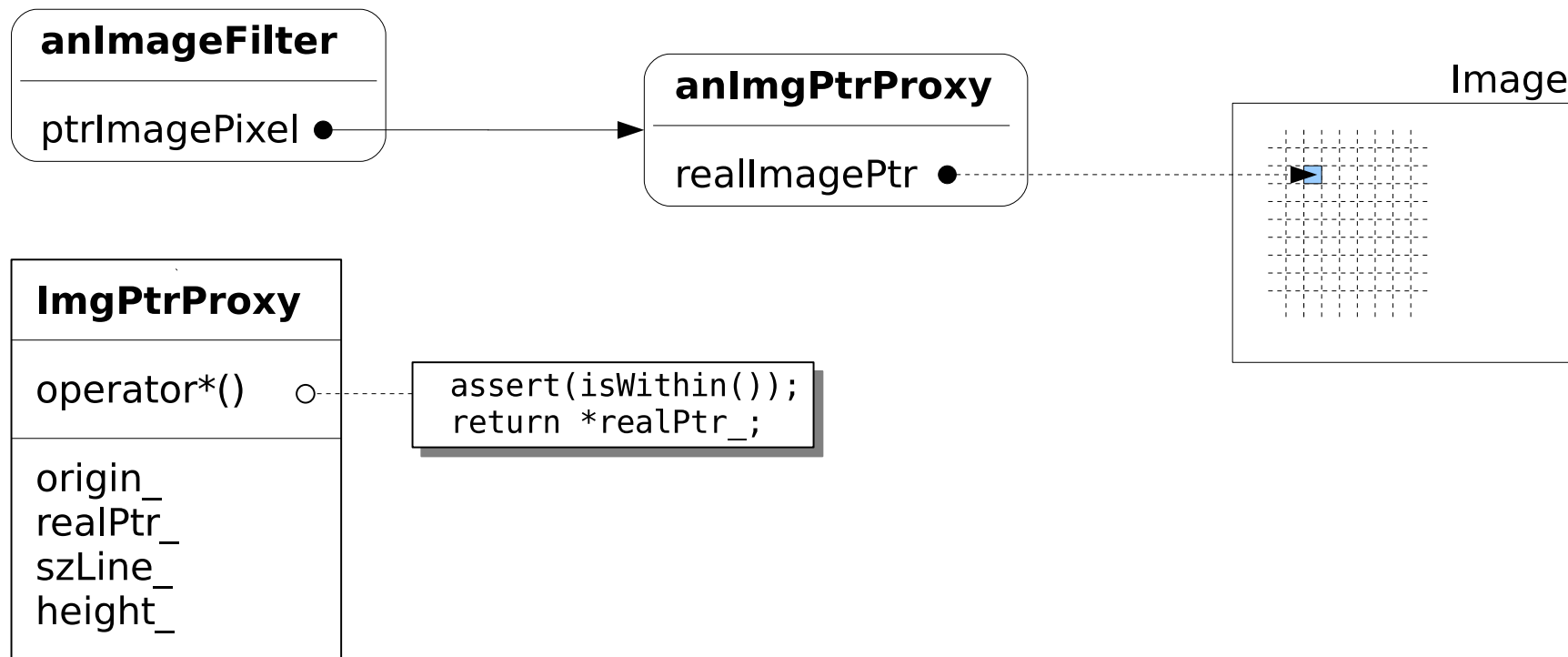
Motivacijski primjer:

- 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 ↑)
 - sliku učitavamo tek kad nam stvarno treba (npr, treba je iscrtati)
 - primjer koncepta odgođene evaluacije (lazy evaluation)
- ideja: postići **transparentni** sofisticirani pristup

PROXY: MOTIVACIJSKI PRIMJER

Pametni pokazivač (smart pointer): učestala varijanta Proxyja

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

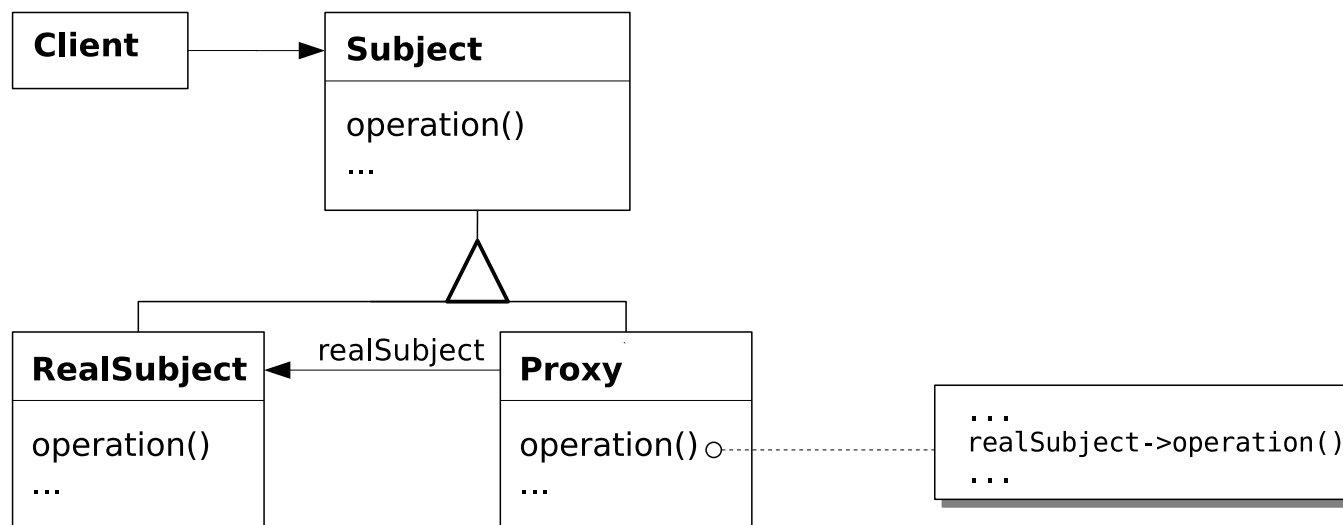


PROXY: PRIMJENLJIVOST I STRUKTURA

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 u domeni **objekata**



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 se ne tiču krajnjeg cilja programa
- 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 *proširenje* operatora:
 - dereferenciranja pokazivača *
 - pristupa članu strukture ->
- Proxy ciljnom subjektu pristupa preko što apstraktnijeg sučelja (npr. virtualni Proxy teško može ne ovisiti o konkretnom objektu kojega stvara)
- Najpodatnije: Proxy i Subjekt izvode zajedničko apstraktno sučelje
 - to nije uvijek praktično izvedivo zbog cijene polimorfnog poziva (pametni pokazivač, podstrukturni proxy)
 - značaj ovoga, kao i obično, ovisi o jeziku (C++, Java, Python)
- 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

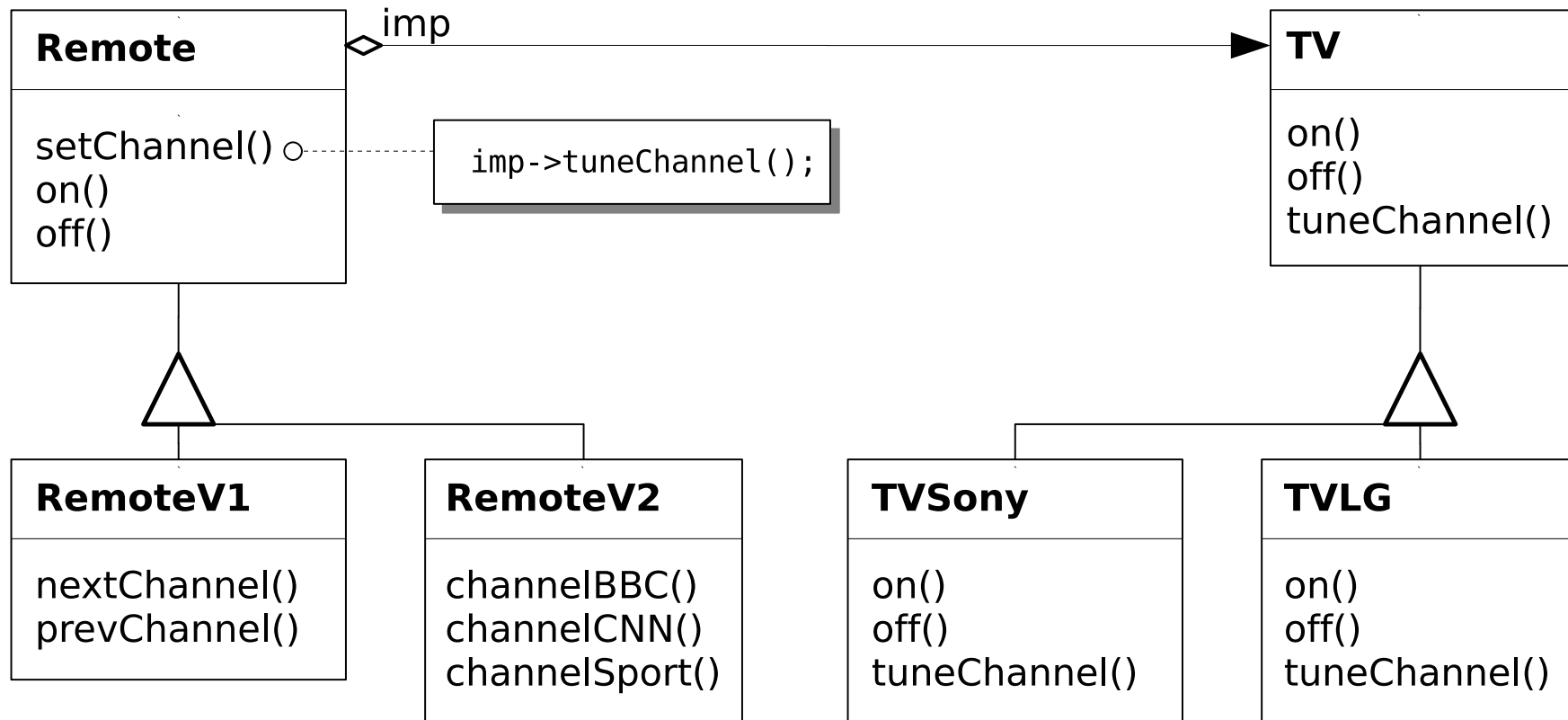
Namjera:

- odvojiti sučelje od implementacije tako da se oboje mogu mijenjati nezavisno

Motivacijski primjer:

- Programska podrška za univerzalni daljinski upravljač
- želja: podržati različite TV uređaje, osigurati kompatibilnost s prošlom verzijom proizvoda
- Monolitna hijerarhija problematična:
 - sklonost kvadratnom porastu broja razreda
 - potiče platformski ovisan kôd
- ideja: dvije osi varijacije modelirati dvjema odvojenim hijerarhijama

MOST: MOTIVACIJSKI PRIMJER



MOST: PRIMJENLJIVOST

Most koristimo za izražavanje koncepata koji imaju više osi varijacije

- svaku odgovornost modeliramo zasebnom hijerarhijom

Najčešće razdvajamo dvije osi varijacije, i to sučelje i implementaciju:

- očekujemo promjene u domeni implementacije, ne želimo da se promjene propagiraju do klijenata

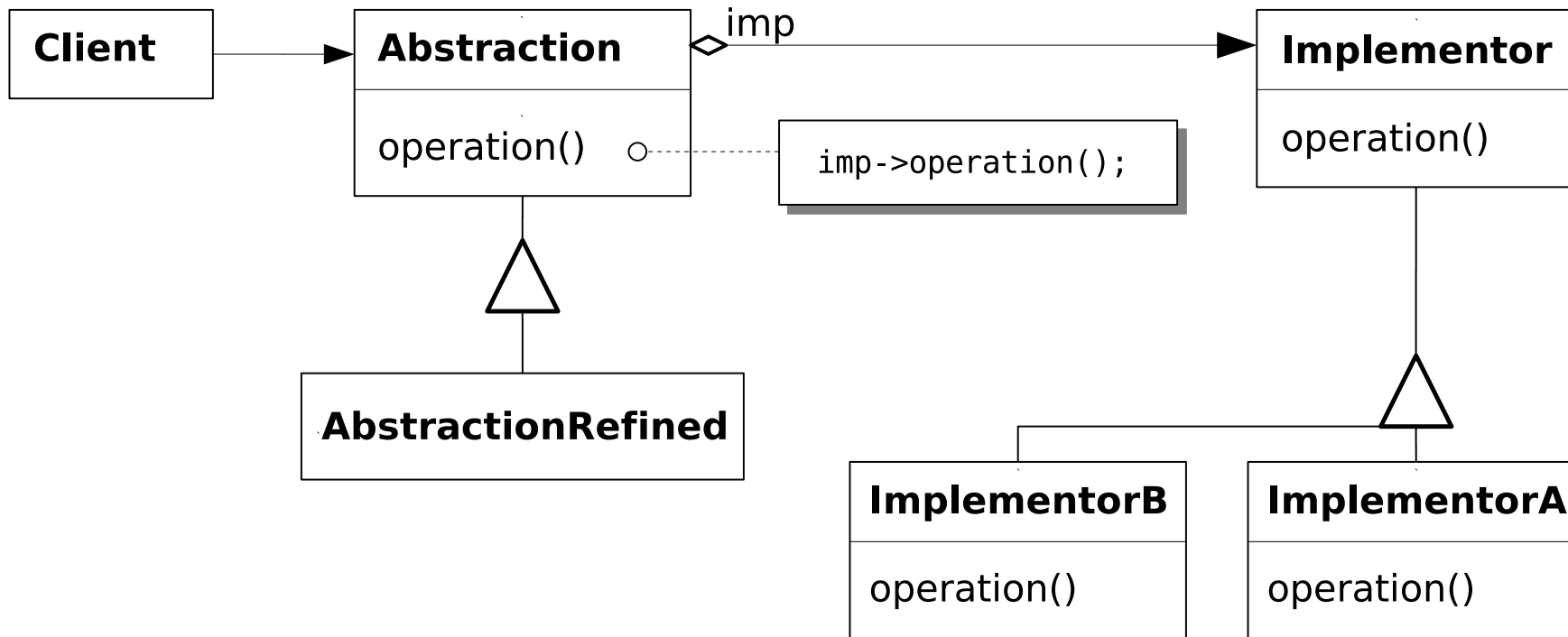
Most možemo koristiti i kod ostalih složenih koncepata, kao alternativu višestrukum nasljeđivanju:

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

Primjenljivost – modeliranje koncepta zasebnim hijerarhijama

MOST: STRUKTURA

Most (bridge): strukturni obrazac u domeni objekata



MOST: SUDIONICI I SURADNJA

Sudionici:

- **Apstrakcija (Remote)**
 - Sučelje prema klijentima, sadrži referencu na Izvođača
- **Prilagođena apstrakcija (RemoteV1)**
 - proširuje sučelje Apstrakcije
- **Izvođač (TV)**
 - definira sučelje za implementacijske razrede
 - u odnosu prema Apstrakciji, operacije su primitivnije i niže razine
- **Konkretni izvođač (TVSony)**
 - izvodi sučelje Izvođača, definira konkretnu implementaciju

Suradnja:

- Apstrakcija prosljeđuje zahtjeve zadanom Izvođaču

MOST: POSLJEDICE

- Modeliranje koncepta odvojenim hijerarhijama:
 - bolja ortogonalnost (načelo jedinstvene odgovornosti)
 - dinamička konfiguracija (načelo nadogradnje bez promjene)
- Za slučaj razdvajanja sučelja i implementacije:
 - pospješujemo binarnu kompatibilnost
 - pospješujemo proširivost uslijed ortogonalnosti sučelja i izvedbe
 - izoliramo klijente od implementacijskih detalja

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 objekta (npr, list vs hash tablica)
 - posredni odabir (apstraktna 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*

MOST: PRIMJENE I SRODNI OBRASCI

Primjene:

- Ostvarivanje platformске neovisnosti (ET++)
- odvajanje implementacije od sučelja (libg++)

Srodni obrasci:

- Prikladni Izvođač se može kreirati i konfigurirati Apstraktnom tvornicom.
- Prilagodnik je sličan Mostu, a razlikuju se prvenstveno u namjeri.

POSJETITELJ: NAMJERA I MOTIVACIJA

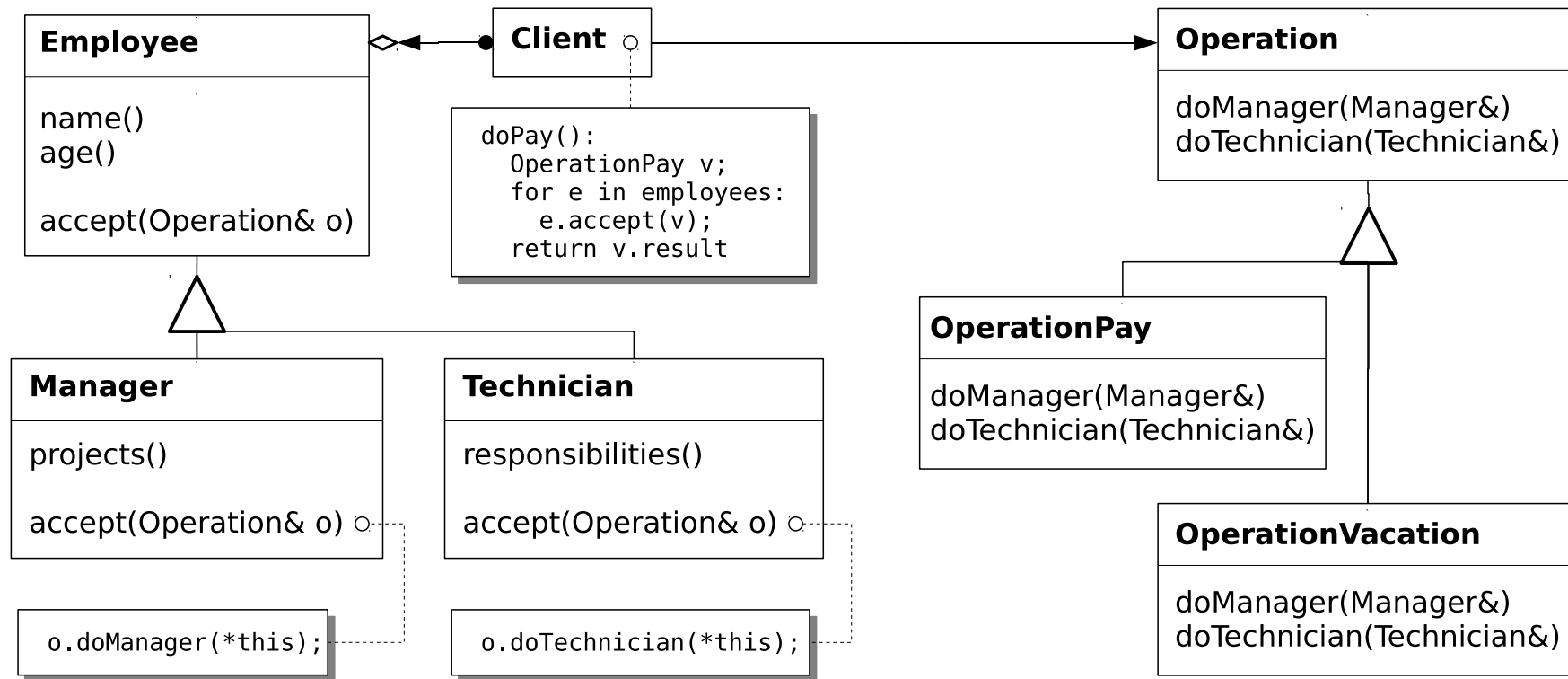
Namjera:

- 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

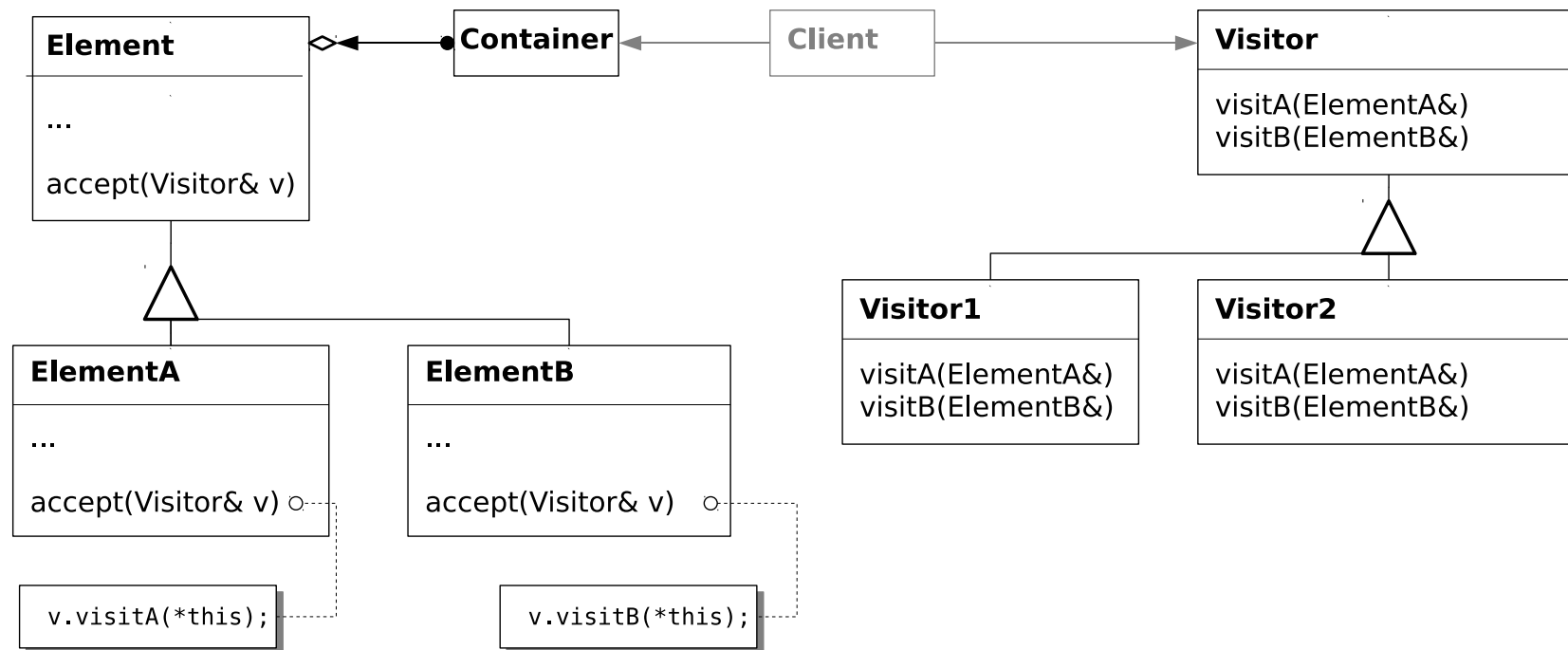


POSJETITELJ: PRIMJENLJIVOST I STRUKTURA

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** u domeni **objekata**



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 referencu 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, ...)

Srodni obrasci:

- Posjetitelj se često koristi u kombinaciji s Kompozitom i Iteratorom

PROTOTIP: MOTIVACIJA I NAMJERA

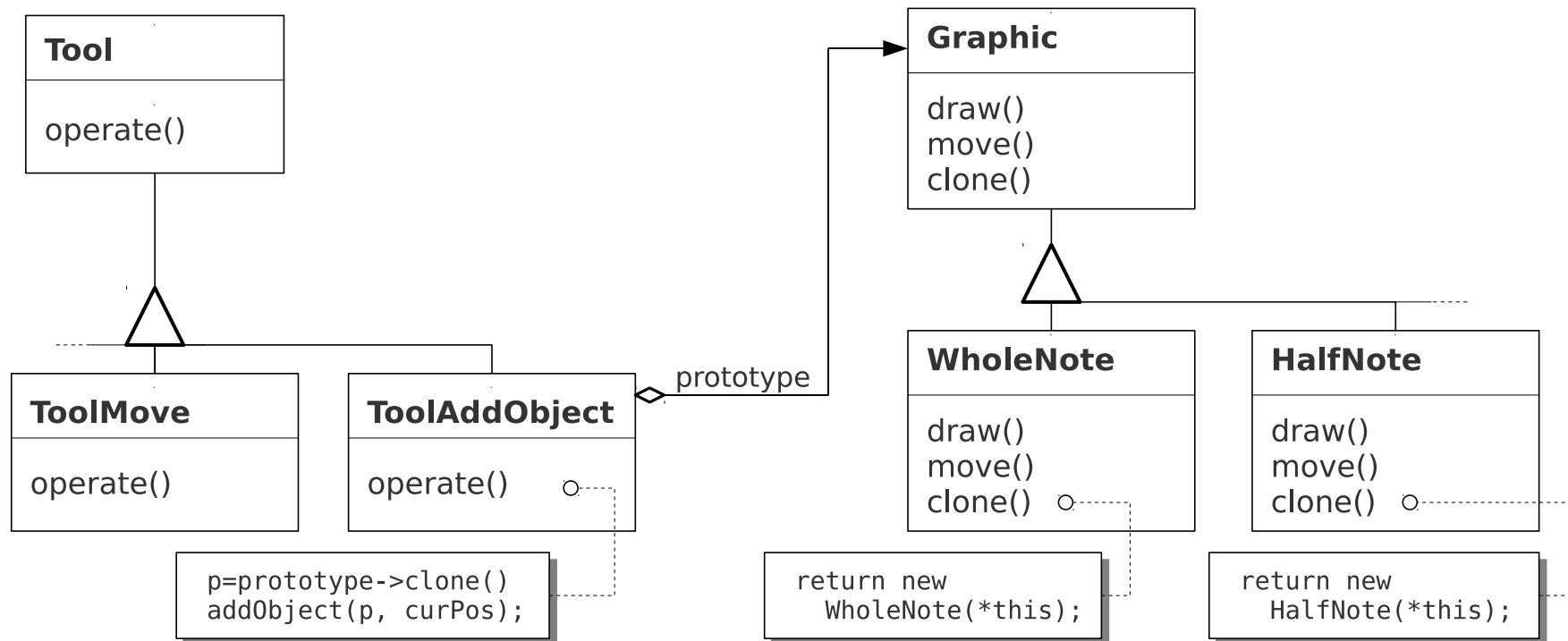
Motivacijski primjer:

- Razvijamo program za uređivanje glazbenih partitura (crtovlja, ključevi, note, itd.)
 - služimo se aplikacijskim okvirom u domeni vektorske grafike
 - okvir barata s apstraktnim operacijama "dodaj objekt", "pomakni objekt", "uredi svojstva", ...
- kako implementirati "dodaj objekt" bez zadiranja u aplikacijski neovisni dio okvira?

Namjera:

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

PROTOTIP: MOTIVACIJSKI PRIMJER

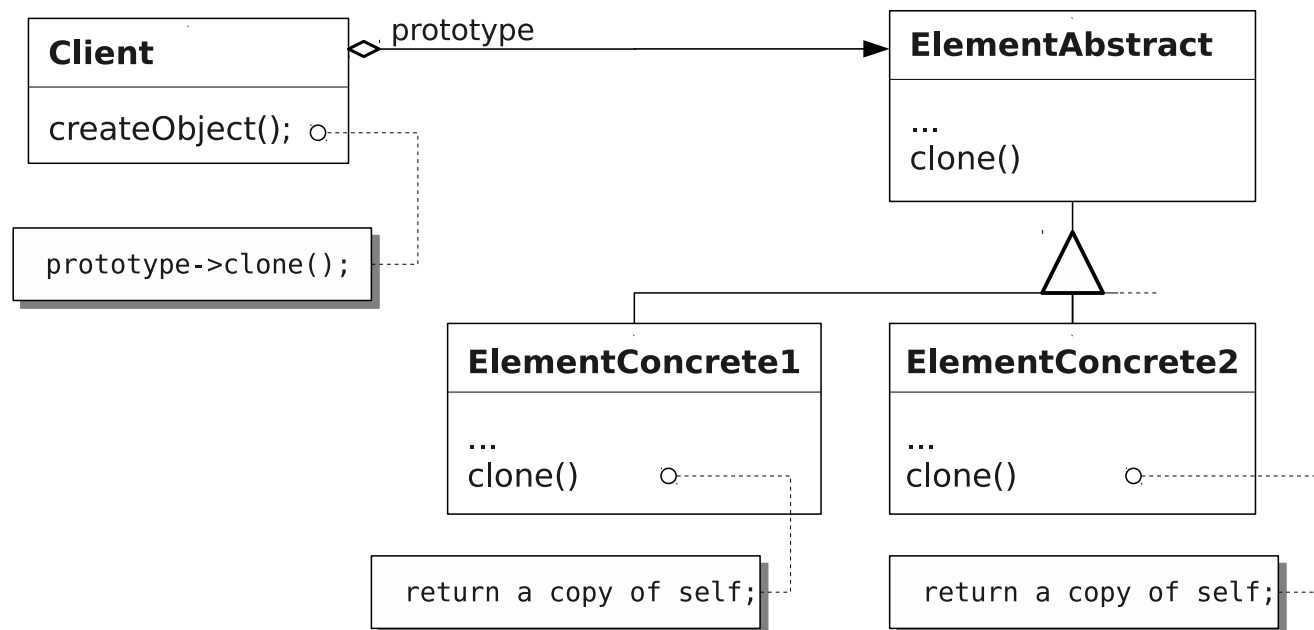


PROTOTIP: PRIMJENLJIVOST I STRUKTURA

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)
- želimo izbjeći korištenje parametrizirane tvornice

Prototip: **kreacijski** obrazac u domeni **objekata**



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 komplicirane 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

MVC: MOTIVACIJSKI PRIMJER

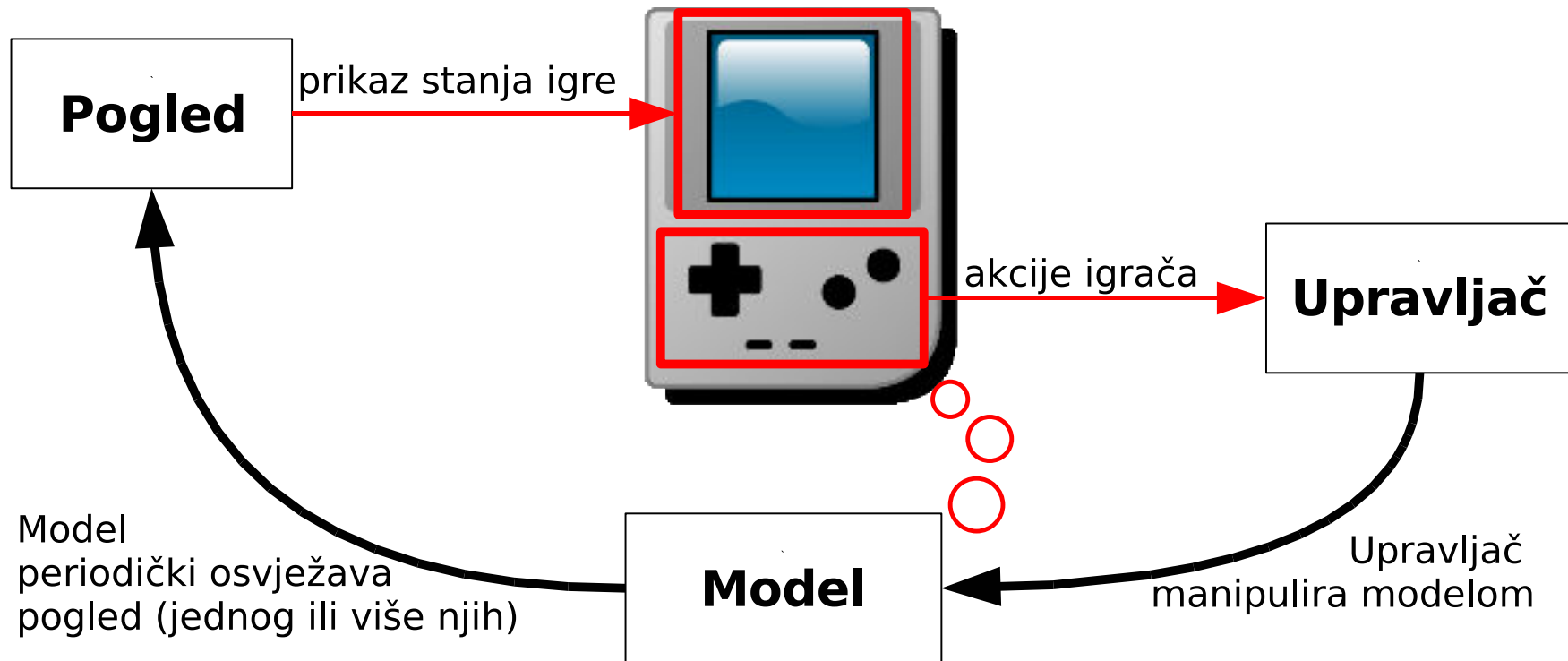
Problem:

- Razvijamo upravljački program za mp3 player
 - tri glavne komponente: računalni model, pogled, upravljač
 - **računalni model** obuhvaća aplikacijsku logiku: pristup datotečnom sustavu, dekompresiju podataka, ...
 - **pogled** određuje sadržaj zaslona i reproducira zvuk
 - **upravljač** održava sučelje za primanje korisničkih akcija
- 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

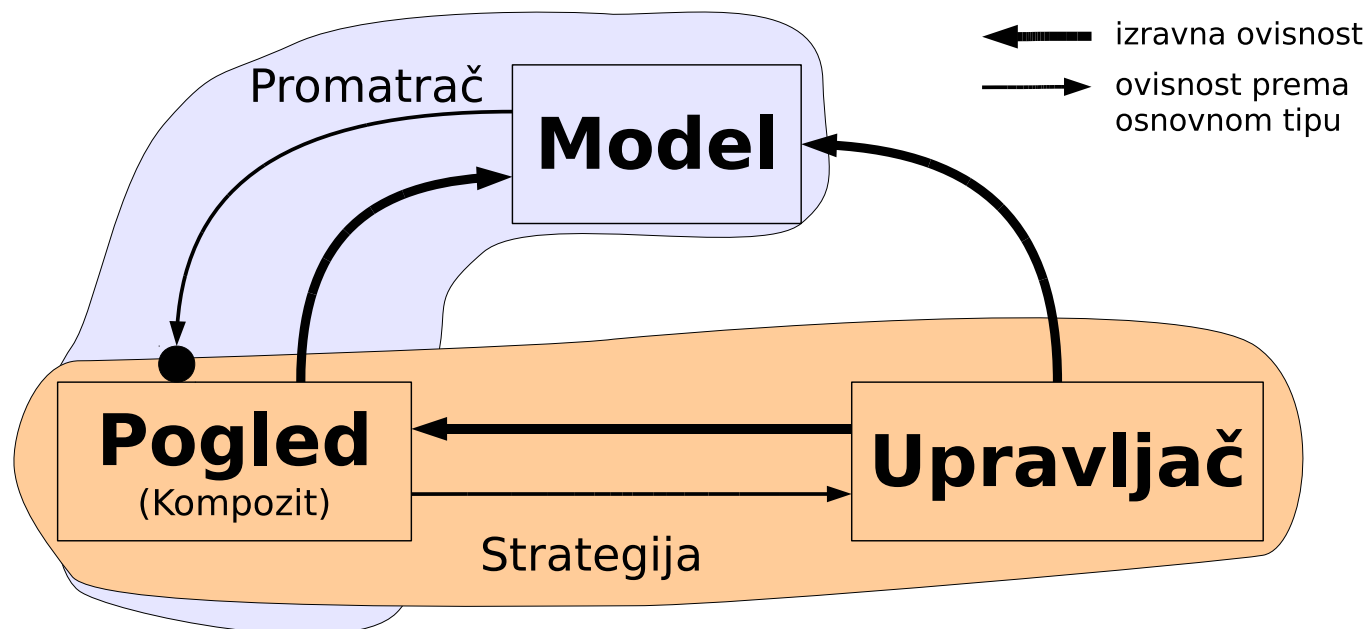
- Partitionirati funkcionalnost interaktivne aplikacije u tri dijela: model (obrada), upravljač (ulaz), pogled (izlaz)
- Premostiti jaz između humanog mentalnog modela i digitalnog modela koji postoji u računalu:
 - podržati iluziju izravnog pogleda na elemente domene i njihovu manipulaciju
 - omogućiti 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, Strategije i Kompozita

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



MVC: SUDIONICI I SURADNJA

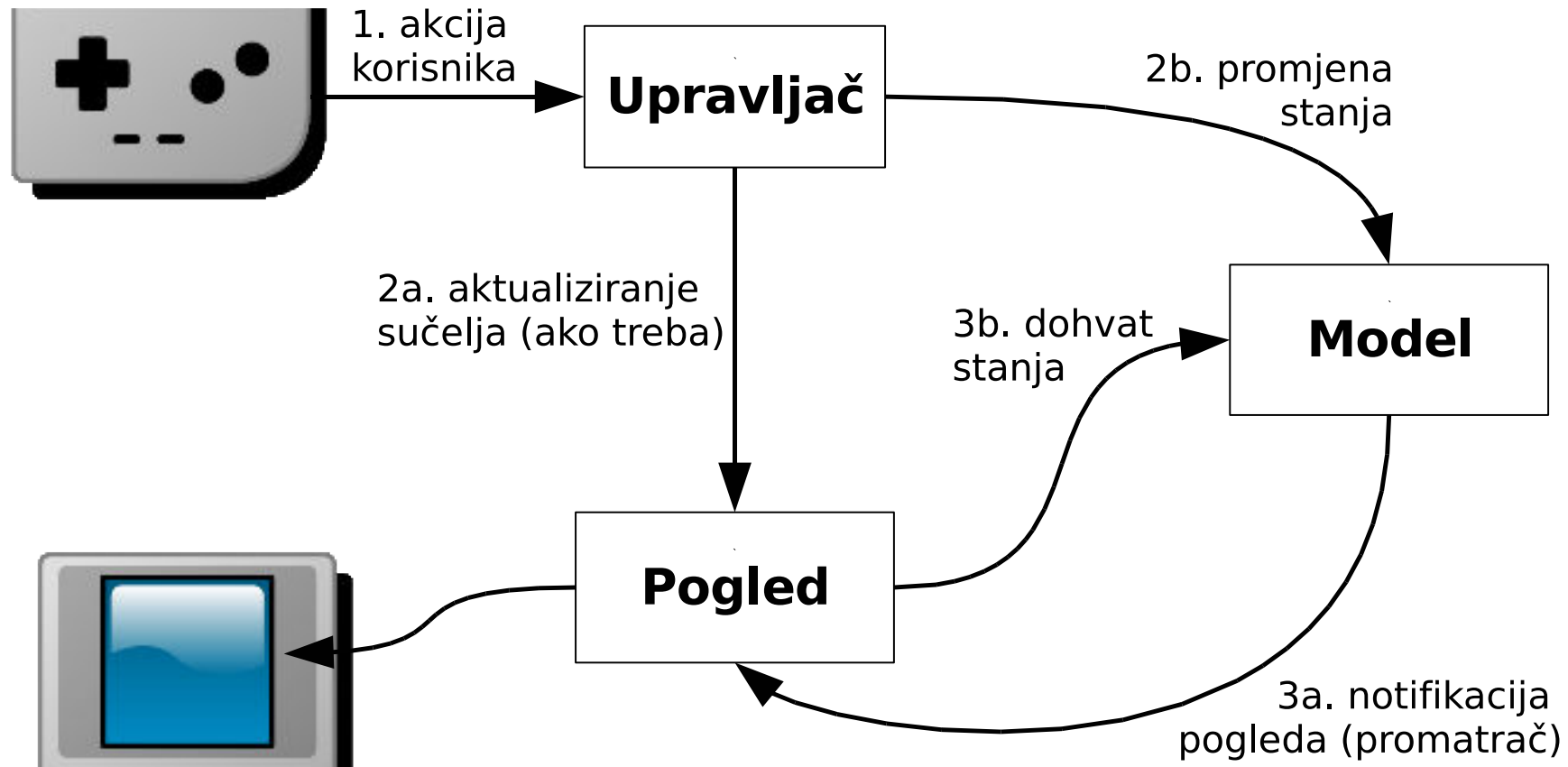
Sudionici:

- **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)

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 biti 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`
- Varijanta MVC-a često prisutan u aplikacijama na webu:
 - pogled: dinamički generirana HTML stranica
 - upravljač: skuplja podatke od korisnika i u skladu s modelom generira HTML
 - model: stvarni podatci spremljeni u udaljenoj bazi podataka

ZAKLJUČAK: OBRASCI

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