Oblikovni obrasci u programiranju

Sažetak prvog ciklusa 2013./2014.

V 1.0

Sadržaj

Uvod

Gradivo obuhvaćeno međuispitom akademske godine 2013./2014.: tehnike za ostvarivanje fleksibilnog koda, načela oblikovanja te oblikovni obrasci strategija, promatrač, okvirna metoda, dekorator, naredba.

Statička i dinamička svojstva koja želimo ostvariti:

1. Korektnost – program obavlja svoj posao
2. Zadovoljavajuća performansa – program radi dovoljno brzo
3. Ugodan izgleda – lijepo korisničko sučelje
4. Lako održavanje – razumijevanje, ispitivanje, nadogradnja
5. Fleksibilnost – otpornost na promjene

Dobru dinamiku programskoj projekta omogućit će kod:

- Kojeg je lako ispitati
- Kojeg je lako razumjeti
- Kojeg je lako nadograditi/promijeniti
- Koji je otporan na promjene

Simptomi programa koji propada, urušava se ili je naprosto neprikladno organiziran:

- **Krutost** – teško nadograđivanje, promjene rezultiraju domino efektom
  - Program je teško promijeniti čak i na jednostavne načine, jer svaka promjena zahtijeva nove promjene – *domino-efekt*
  - Uzrok: *dugi lanac eksplicitne ovisnosti*:
    - A ovisi o B, B ovisi o C, …, Y ovisi o Z
  - Protiv krutosti borimo se kraćenjem lanca ovisnosti uz pomoć *apstrakcije* i *enkapsulacije*
- **Krhkost** – lako unošenje suptilnih grešaka
  - Tendencija programa da puca uslijed *ponavljanja*
  - Uzrok: *implicitna međuovisnost* uslijed *ponavljanja*
    - Jedna konceptualna izmjena mora se unijeti na više mjesta
  - Propusti rezultiraju suptilnim greškama koje je teško pronaći
  - Krhkost *jača krutost*
- **Nepokretnost** – teško višestruko korištenje
  - Otežano višestruko korištenje prethodno razvijenih modula
  - Uzrok: pretjerana *međuovisnost* zbog *neadekvatnih sučelja*
    - Postojeći modul ima *previše prtljage* (overhead-a) koju nije lako eliminirati
    - Moduli se pišu iznova, umjesto evolucije kroz ponovno korištenje
  - Nepokretnost potiče ponavljanje – *krhkost* i *krutost*
- **Viskoznost** (trenje) – sklonost k slabljenju integriteta programa (pretjerana složenost, redundancija, …)
  - Teško je nadograđivati uz očuvanje konceptualnog integriteta programa
  - Dvije vrste viskoznosti:
    - *Viskoznost programske organizacije* – nadogradnje koje čuvaju integritet zahtijevaju puno manualnog rada ili nisu očite. Potreban znatan napor za održavanje integriteta programa
    - *Viskoznost razvojnog procesa* – spora, neefikasna razvojna okoline. Komplicirani sustav za verziranje implicira rjeđe sinkronizacije koda te kasnije otkrivanje problema u vezi s integracijom
  - Uzrok: zbog neadekvatnog oblikovanja ili izmijenjenih zahtjeva dolazi do neplaniranih promjena
    - Izmjene uzrokuju degradiranje organizacije – neželjena *međuovisnost* komponenata ili funkcionalnosti, *ponavljanje* / redundancija

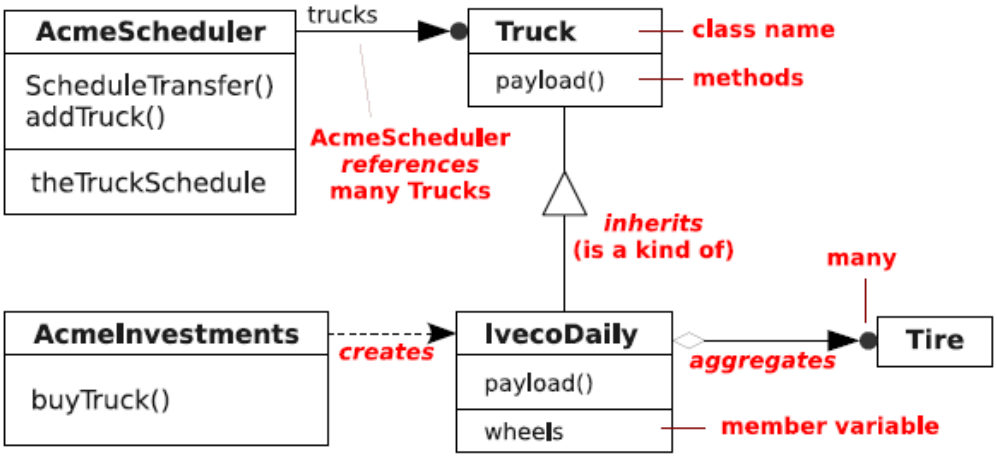
Tehnike

Logičko i fizičko oblikovanje

- Logičko oblikovanje – elementi programskog jezika (moduli: razredi i funkcije)
- Fizičko oblikovanje – raspodjela funkcionalnosti po datotekama
    - Komponenta
        - Je temeljna jedinica: sastoji se od *sučelja* (*.hpp*) i *implementacije* (*.cpp*, *.a*, *.dll*, …)
        - Sadrži jednu ili više logičkih jedinica
        - Je temeljna jedinica pri verziranju i testiranju
- Dobra organizacije poštuje i logička i fizička načela

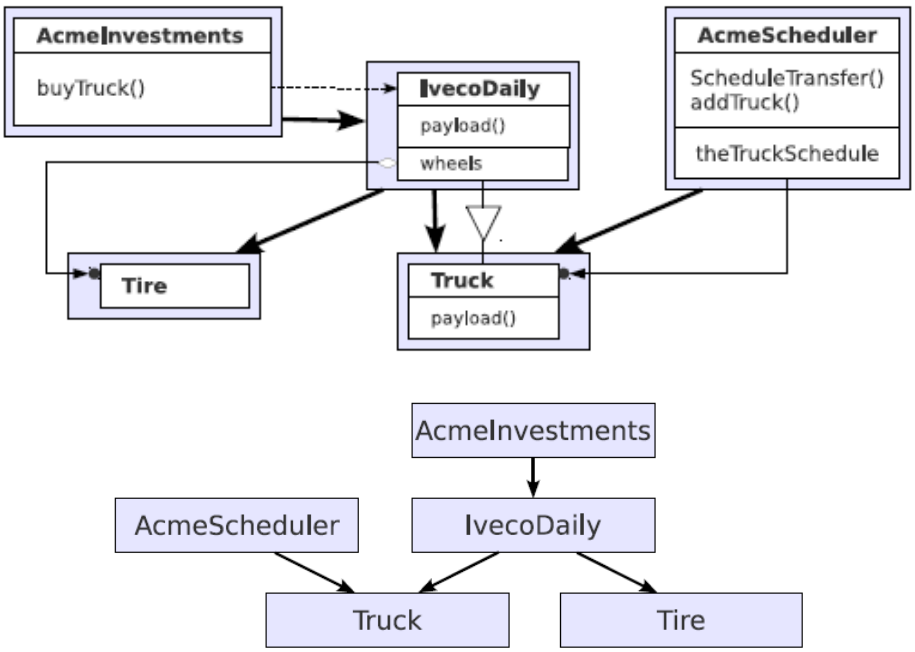
Gof OMT

- Object Modelling Technique – jezik za *modeliranje* programske podrške (prethodnik UML-a)
- Koristimo pojednostavljene dijagrame razreda za opis *logičkih* odnosa – nasljeđivanje, referenciranje, sadržavanje, stvaranje



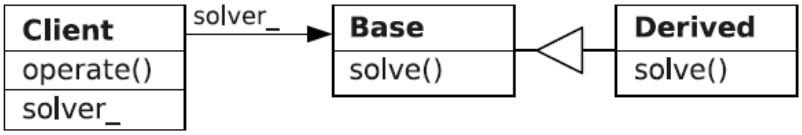
Logičko oblikovanje

- OMT ne može prikazivati odnose u fizičkom oblikovanju pa uvodimo hibridnu notaciju [Lakos96]. Komponente fizičkog oblikovanja prikazujemo sivim pravokutnicima. Veze između sivih pravokutnika opisuju ovisnost komponenata.



Fizičko oblikovanje

Dinamički polimorfizam



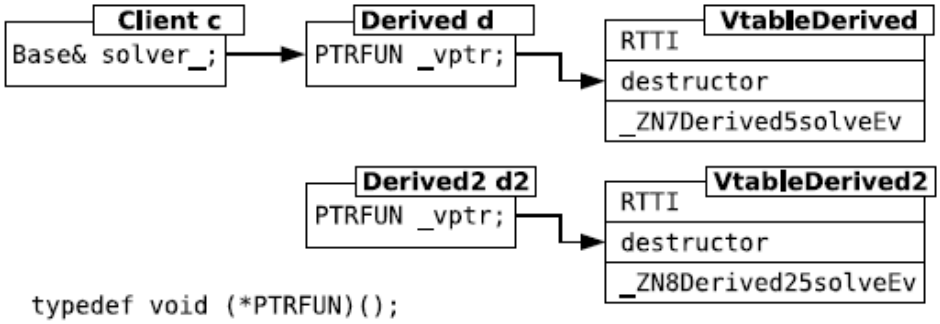
- `Client` poziva `Derived::solve` preko sučelja osnovnog razreda
- `Client` pozna `Base`, ali ne pozna `Derived`
- Objekt tipa `Client` poziva `Derived` bez da ovisi o njemu

Za poziv `solver_.solve()` kažemo da je *polimorfan* jer odredište u trenutku pisanja programa nije poznato

- Radu se o *dinamičkom polimorfizmu* jer odredište postaje poznato tek u trenutku izvođenja
- Dinamički polimorfizam je ključni koncept OO oblikovanja

Što se događa kod poziva `solver_.solve()`?

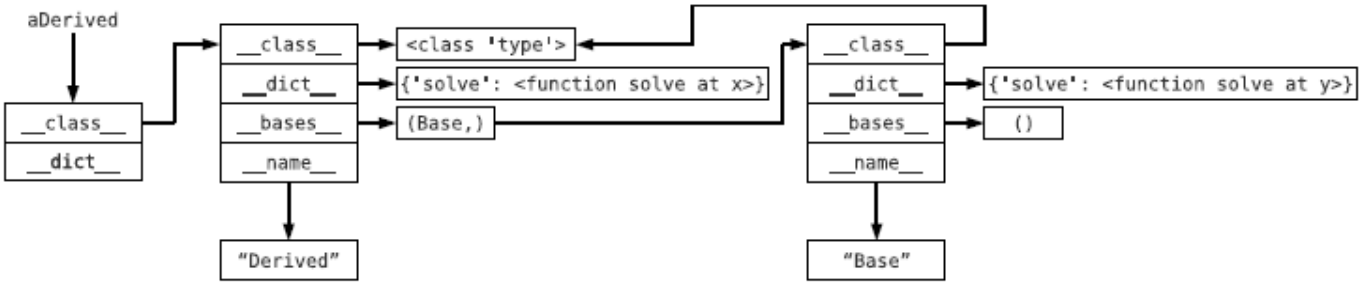
- `(solver_.vptr[1](&solver_));`



- Dinamički polimorfizam u C-u može se ostvarit:
  - Korištenjem tablice pokazivača na funkcije
  - Korištenjem pretprocesorskih makroa
  - Pozivanjem regularnih funkcija C-a
  - Korištenjem vanjskih biblioteka

U dinamički tipiziranim jezicima kao što je Python, dinamički polimorfizam funkcionira znatno drugačije nego u statički tipiziranim jezicima kao što su Java i C++.

- Puno *fleksibilnija* izvedba (duck typing)
- Mogućnost *dodavanja metoda objektu tijekom izvođenja*
- *Slabije performanse*, no moguće je ubrzanje uporabom *cachiranja* i *JIT* prevodioca
- *Sve je objekt*, pa čak i korisnički napisani razredi na temelju kojih stvaramo objekte



Statički polimorfizam – generici

Statički polimorfizam - ispitna komponenta *parametrizira* predložak `Client` *objektom proizvoljnog razreda*:

```
template <typename Solver>
class Client{
public:
   Client(Solver &s);
   void operate();
   Solver& solver_;
};

template <typename Solver>
Client<Solver>::Client(Solver &s):
  solver_(s){}

template <typename Solver>
void Client<Solver>::operate(){
  std::cout <<solver_.solve()
    <<"\n";
}
```

```
//==== mysolver.hpp
class MySolver{
public :
  int solve();
};

//==== mysolver.cpp
int MySolver::solve(){
  return 42;
}

//==== test.cpp
int main (){
  MySolver s;
  Client<MySolver> c(s);
  c.operate();
}
```

Jedini zahtjev na parametar predloška je da ima metodu `solve`. Nedostatak: povezivanje se zbiva prilikom prevođenja – *manja fleksibilnost* nego kod dinamičkog polimorfizma.

Generičko programiranje – proširena gramatika omogućava *parametriziranje funkcija i razreda*.

```
#include <iostream>

template <class T>
inline T mymax(T x, T y) {
  if (x < y)
    return y;
  else
    return x;
}

int main(){
  std::cout <<mymax(3, 7) <<"\n";
  std::cout <<mymax(3.1, 7.1) <<"\n";
}
```

Prevođenje se odgađa do trenutka kada parametri postaju poznati – nakon instanciranja predloška koristi se osnovna gramatika.

Generički izvršni kod obično ima povoljniju složenost

- Manja vremenska složenost – razrješavanje polimorfnog poziva tijekom prevođenja
- Manja prostorna složenost – nema potrebe za pokazivačima na tablice funkcije

Generičko programiranje stvara veći izvršni kod.

Ortogonalnost

Neovisnost algoritama i spremnika podataka – npr. algoritam reverse možemo zvati nad poljem, vektorom i vezanom listom.

OOP (Object-oriented programming)

OOP razmatra *dinamiku* (evolucije) sustava:

- „Kako postići da kod kojeg pišemo *danas* ispravno radi s kodom kojeg ćemo pisati *dogodine*?“
- „Što će se vjerojatno *mijenjati* u budućnosti?“

DbC (Design by Contract)

Komponente surađuju ispunjavanjem obveza definiranih *ugovorom*.

- Preduvjeti – reguliraju obaveze klijenta prema pružatelju
- Invarijante – interni pokazatelji integriteta komponente
- Postuvjeti – reguliraju obaveze pružatelja prema klijentu

Korištenje konstrukta `assert` ili bacanje iznimaka (`throw`).

Logička načela

Nadogradnja bez promjene (NBP)

Dodavanje funkcionalnosti *bez utjecaja na klijente*. Stari kod radi s novim kodom. Skrivanje informacija i apstrakcija podataka.

Cilj – omogućiti proširenje funkcionalnosti komponente bez mijenjanja njene implementacije.
Fleksibilnost – nadogradnja ne utječe na klijente.

Open-closed principle – komponente su otvorene za nadogradnju, ali zatvorene za promjene.

NBP možemo ostvariti:

- Nasljeđivanjem implementacije
  - Novi razredi pozivaju temeljni implementaciju nasljeđivanjem starog razreda, *bez polimorfnih poziva*
  - Postojeći klijenti *ne mogu doći do nove funkcionalnosti*
  - Novi kod poziva stari
  - Danas se preferira kompozicija
- **Nasljeđivanjem sučelja**
  - Klijenti transparentno pristupaju novoj implementaciji *polimorfnim pozivom preko starog sučelja*
  - *Stari kod poziva novi kod*
- Generičkim programiranjem

Proceduralni stil uzrokuje kruti i krhki kod. Enkapsulacija i apstrakcija poboljšavaju dinamička svojstva programa.
Enkapsulacija – klijenti razreda ne smiju izravno referencirati podatkovne članove.
Apstraktni razredi – nemaju podatkovnih elemenata, imaju virtualne funkcije.
Virtualne funkcije – moguće pozivanje modula napisanih godinama nakon klijenta.
Generički programi – injekcija novog koda u stari pri prevođenju.

Nadomjestivost osnovnih razreda (LNS, Liskovino načelo supstitucije)

Ako A izvodi iz B, onda A možemo koristiti i kao B. Npr. tko god zna voziti auto, zna voziti i Yugo.

- Nasljeđivanje modelira relaciju *je_vrsta*
- Izvedeni razredi trebaju poštivati ugovore osnovnog razreda
- Javno nasljeđivanje rijetko koristimo za ponovno korištenje
- Kršenje principa je često posljedica slabog poznavanja domene

Kršenje LNS-a može se popraviti na 3 načina:

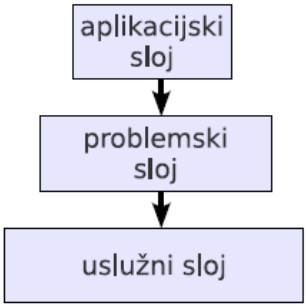
1. Smanjiti odgovornost osnovnog razreda
2. Povećati odgovornost izvedenog razreda
3. Odustati od izravnog roditeljskog odnosa dvaju razreda

Statički tipizirani jezici (C++, Java, C#, …)	Implicitno tipizirani jezici (Python, Ruby, JS, …)
LNS predstavlja recept za korištenje *nasljeđivanja.* Nasljeđivanje koristimo za modeliranje nadomjestivih tipova. Za ponovno korištenje funkcionalnosti preferiramo kompoziciju.	Poopćeni LNS formuliramo neovisno o nasljeđivanju, uz pomoć pojma nadomjestivosti tipova. Nadomjestivost naknadno oblikovanih pružatelja je preduvjet da klijent bude NBP.

Inverzija ovisnosti (NIO)

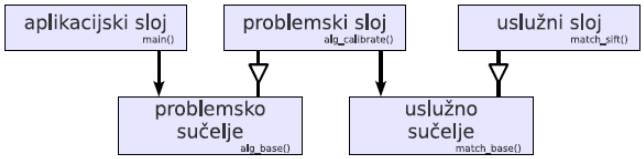
Usmjeravanje ovisnosti prema apstraktnim sučeljima.

Proceduralni stil dovodi do piramidalne strukture međuovisnosti:



- Moduli visoke razine ovise o izvedbenim detaljima – ne mogu se ni prevesti ni ispitati ako niži moduli nisu dovršeni
- Pri mijenjanju modula niže razine često se javlja domino-efekt – promjene se propagiraju prema višim razinama.

Ukoliko ovisnosti idu prema apstrakcijama:



- Ne ovisimo više o nepostojanim modulima s detaljnim implementacijama – kažemo da je ta *ovisnost invertirana*
- Promjer grafa je smanjen – putevi ovisnosti su kraći

Važan problem – stvaranje objekata konkretnih razreda:

- Stvaranje implicira ovisnost – kako izbjeći ovisnost glavnog programa o modulima niske razine?
- Primjena *injekcije ovisnosti* i *obrasca tvornice*

Injekcija ovisnosti

Umjesto hardkodiranog kompliciranog konkretnog člana, uvodi se konfiguriranje preko *reference na osnovni razred*:

- Ovisnost uslijed stvaranja izvlači se u zasebnu komponentu
- Referenciraju se apstraktni razredi, a ne konkretni
- Maksimiziramo inverziju ovisnosti, odnosno lokaliziramo ovisan kod
- Mogućnost *neovisnog ispitivanja*

Načelo jedinstvene odgovornosti (NJO)

Komponente modeliraju koncepte koji imaju jasnu odgovornost:

- Programski moduli moraju imati *samo jednu odgovornost*
- Odgovornosti modula odgovaraju razlozima za promjenu (veza 1:1)
- Ukoliko svi moduli imaju jedinstvenu odgovornost, sve promjene rezultiraju promjenom samo jednog modula. U suprotnom, imamo krhkost i nepokretnost

Potrebno je oblikovati *ortogonalan* sustav u kojemu razdioba poslova odgovara intrinsičnoj strukturi problema.

- Odgovornost modula je kvant funkcionalnosti iz domene aplikacije
- Svaki modul bi trebao imati jednu, samo jednu odgovornost
- Često teško pogoditi isprve
- Nužna je *analiza domene* – svodi se na određivanje odgovornosti
- Ukoliko moduli nemaju jedinstvenu odgovornost, podsustav treba *prekrojiti* (*refactoring*) što ranije

Načelo izdvajanja sučelja (NIS)

Ne tjerati klijente da ovise o onom što ne koriste. Nekoherentnim konceptima (ako ih baš moramo imati) klijenti trebaju *pristupati preko izvedenih sučelja*. Nužno je napraviti kompromis s NJO.

Fizička načela

Najzgodnije provoditi nad *datotekama izvornog koda*.

- Rano *automatsko testiranje* (regresijsko, inkrementalno) – testiranje u izolaciji
- Plitka i nepovezana struktura ovisnosti – *aciklička ovisnost*
- *Grupirati* komponente u pakete – koherencija
    1. Grupiranje prema *korelaciji korištenja* – komponente koje se ne koriste zajedno ne pripadaju istom paketu
    2. Grupiranje prema *zajedničkom izdavanju* – grupiramo komponente koje se zajedno izdaju i održavaju
    3. Grupiranje prema *odgovornosti* – komponente osjetljive na promjene iz istog skupa
- Urediti odnose među paketima

Primijeniti *načela*:

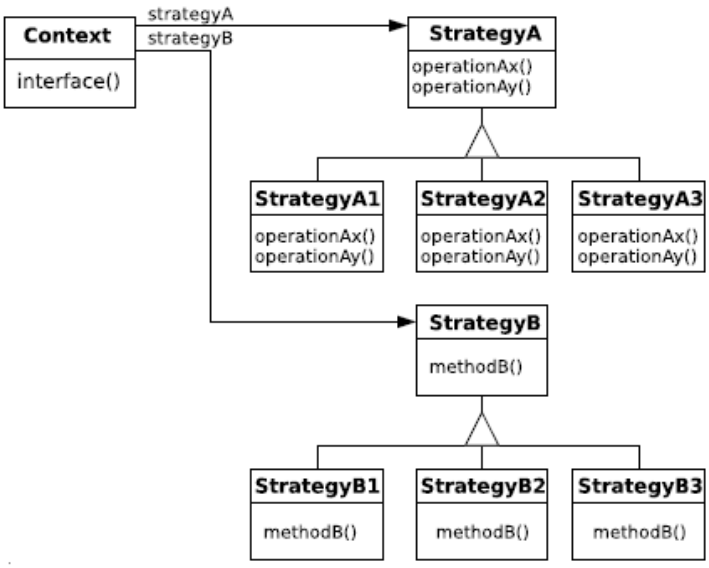
1. Načelo *acikličke ovisnosti*
2. Načelo *stabilne ovisnosti* – usmjeravanje ovisnosti prema inertnijim paketima
3. Načelo *primjerene apstrakcije* – paket treba biti tim apstraktniji što je više stabilan

Stabilnost – otpornost na promjene

Oblikovni obrasci

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

Strategija



Koristimo kada

- Trebamo *dinamički mijenjati ponašanje* neke komponente (konteksta)
- Ponašanje zadajemo odabirom postupka*, bez potrebe za mijenjanjem izvornog koda konteksta*
- Posebno korisno ako imamo *više nezavisnih obitelji postupaka* (više strategija unutar konteksta)

Sudionici

- Strategija
  - Deklarira *zajedničko sučelje* svih podržanih postupaka
  - Preko tog sučelja *kontekst poziva konkretne postupke*
- Konkretna Strategija
  - Implementira postupak preko *zajedničkog sučelja*
  - Često je jedna od strategija null-objekt koji ne radi ništa (korisno za testiranje)
- Kontekst
  - Konfigurira se preko *pokazivača na Konkretnu Strategiju*
  - Može definirati sučelje preko kojeg Strategije mogu pristupiti *njegovim podatcima*

Suradnja

- Prijenos parametara između *Konteksta* i *Strategije*
  - Izravno slanje parametara (push)
  - Neizravno slanje (pull) – kontekst šalje sebe, Strategija uzima ono što joj treba – međuovisnost!

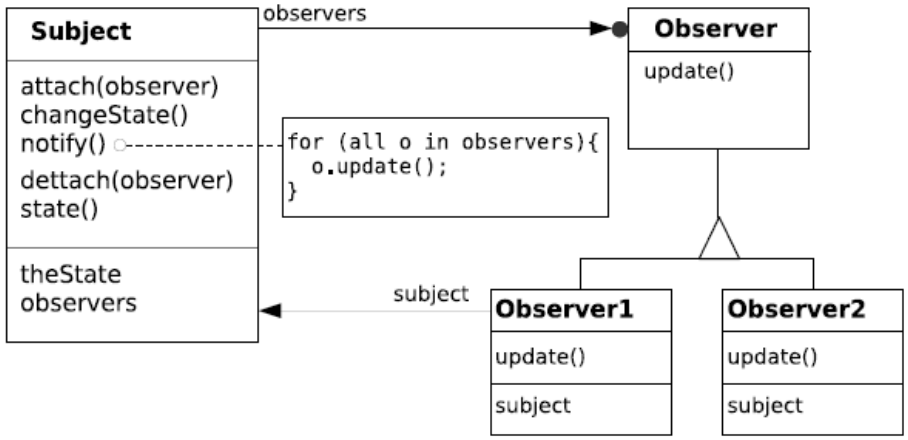
Posljedice

- Modeliramo obitelji opcionalnih postupaka (zajednička funkcionalnost se može izdvojiti nasljeđivanjem)
- Podatnija alternativa nasljeđivanju - razdvajanje konteksta od ponašanja, ortogonalnost
- Eliminacija uvjetnih izraza
  - Veća fleksibilnost i složenost u odnosu na alternative

Primjeri upotrebe

- Grafičke biblioteke – različiti načini prikaza grafičkih elemenata
- Optimizirajući prevodioci – različite tehnike za dodjeljivanje registara, redoslijed instrukcija, …
- Logička sinteza – postupci provlačenja vodova između elektroničkih elemenata
- STL – način alociranja memorije standardnih spremnika
- GUI biblioteke – validacija korisničkog unosa u dijalozima

Promatrač



Koristimo kada

- *Promjena jednog objekta zahtijeva promjene u drugim objektima* koji nisu unaprijed poznati
- Objekt treba *obavještavati druge objekte*, uz uvjet da bude *što neovisniji o njima*
- Potrebno *prikazivati različite aspekte nekog dinamičkog stanja*, a želimo *izbjeći cikličku ovisnost* između stanja i pogleda
- Postoje dva programska entiteta, jedan ovisan o drugome, a zgodno ih je razdvojiti

Sudionici

- Subjekt (izdavač, izvor informacija)
  - Poznaje *promatrače* preko *apstraktnog sučelja*
  - Pruža *sučelje* za *prijavu* novih i *odjavu* postojećih promatrača
  - Sadrži *original stanja* koje je od interese promatračima (pruža *sučelje* za pristup tom stanju)
  - *Obavještava* promatrače o promjenama stanja
- Promatrač (pretplatnik, element prikaza)
  - Pruža *sučelje* za *obavještavanje* o promjenama u subjektu
- Konkretan promatrač (tablica, histogram, grafikon)
  - Sadrži *referencu* na subjekt
  - Obično ima *kopiju podskupa stanja* subjekta
  - Implementira sučelje za *obavještavanje*

Suradnja

- Promatrači se *dinamički* prijavljuju i odjavljuju kod Subjekta
- Subjekt *obavještava* promatrače kad god se dogodi *promjena stanja*
- Nakon primanja obavijesti, Promatrači *pribavljaju* novo stanje i poduzimaju odgovarajuće *aktivnosti*
- Promjene mogu biti inicirane i od strane Promatrala – promatrač koji je inicirao promjenu također obnavlja stanje tek *nakon* službene obavijest
- Obavještavanje može biti inicirani ili od strane *Subjekta* ili od strane *njegovih klijenata*

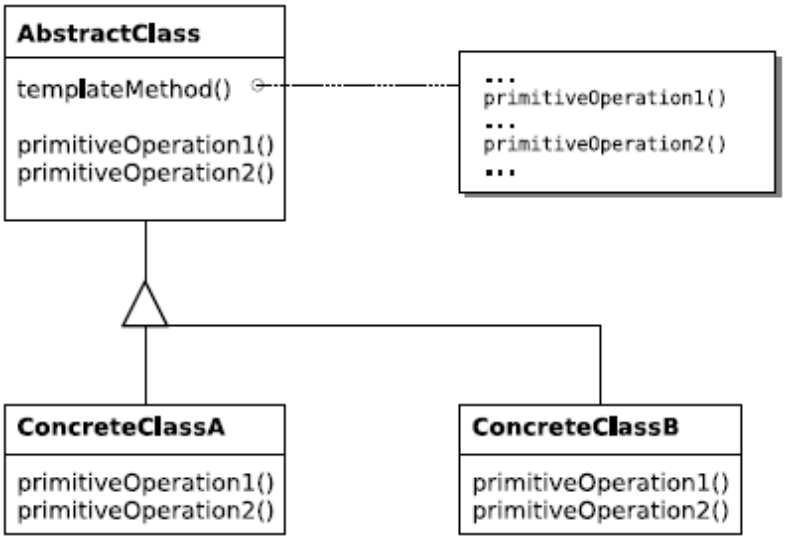
Posljedice

- *Ukidanje ovisnosti* između Subjekta i konkretnih Promatrača! Subjekt *ne ovisi o složenoj implementaciji* Promatrača.
- Mogućnost *dinamičke* prijave i odjave Promatrača
- Potreba za složenijim protokolom obavještavanja (sve promjene se ne tiču svih promatrača)
- Pospješuju se *NBP* (dodavanje promatrača), *NIO* (subjekt ovisi o apstraktnom sučelju) te *ortogonalnost* (razdvajanje promatrača i subjekta)

Primjeri upotrebe

- Sastavni dio arhitektonskog obrasca model-pogled-upravljač (MVC)
- Srodan oblikovnoj organizaciji dokument-pogled (MFC, Microsoft)
- Elementi GUI biblioteka se ponašaju kao Subjekti, dok se korisnički kod prijavljuje kao Promatrač (listeners)
- U C-u, poziv promatrača izvodi se tzv. call-back funkcijama čiji prvi argument obično identificira kontekst subjekta

Okvirna metoda



Koristimo kada

- Stalne dijelove algoritma valja *prikupiti na jednom mjestu*, dok izvedeni razredi definiraju *promjenljive korake* – istu grubu strukturu postupaka (okvir) dijeli više izvedenih razreda
- *Zajedničko ponašanje* izvedenih razreda želimo *izdvojiti* da izbjegnemo *dupliciranje* koda
- Izvedenim razredima treba omogućiti *proširivanje funkcionalnosti* – okvirna metoda poziva tzv. metode proširenja (hooks); osnovni razred pruža praznu podrazumijevanu implementaciju

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 postupaka
  - 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 postupaka u Apstraktnom razredu

Posljedice

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

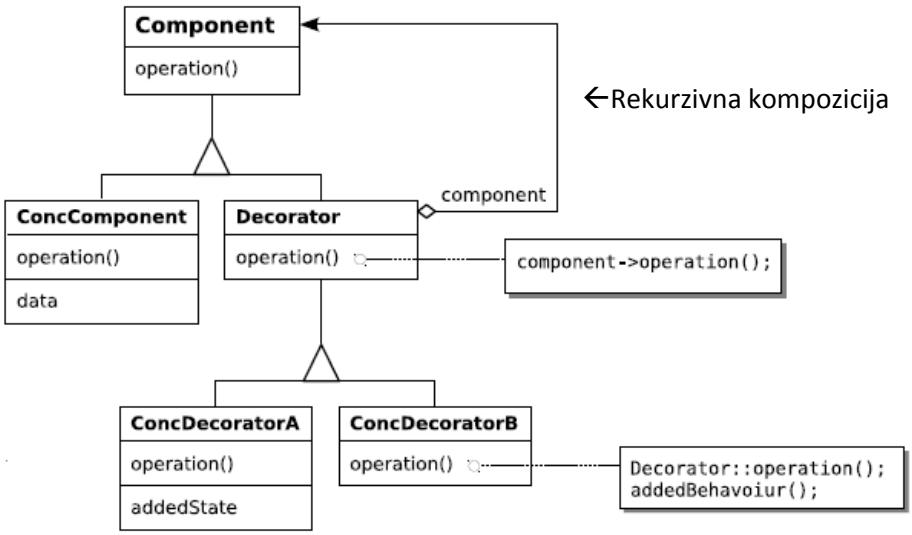
Primjeri upotrebe

- Vrlo raširena u praksi, kao *najjednostavniji* OO *oblik višestrukog korištenja*
- Biblioteka pythona: cmd, http.server.BaseHTTPRequestHandler

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*

Dekorator



Koristimo kada

- Pojedinim objektima je potrebno pridodati odgovornosti *dinamički* i *transparentno* (*bez utjecaja na druge komponente*)
- Odgovornosti je potrebno moći *povući*
- Nasljeđivanje konkretne nepraktično – želimo dinamički konfiguraciju, imamo više osnovnih razreda, nemamo višestruko nasljeđivanje
- Dodatke nije prikladno implementirati unutar osnovnog razreda jer *nisu primjenljivi* na *sve* objekte ili su *preglomazni* i *često se mijenjaju*

Sudionici

- Komponenta (java.io.OutputStream)
  - Definira *sučelje* za objekte kojima se *odgovornosti* mogu *dinamički* konfigurirati
- Konkretna komponenta (java.io.FileOutputStream)
  - Definira *izvedbu* objekta s *dinamičkim* odgovornostima
- Dekorator (java.io.FilterOutputStream)
  - Nasljeđuje *sučelje* Komponente
  - Sadrži *referencu* na *umetnutu* komponentu (*rekurzivna kompozicija*)
- Konkretni dekorator (java.io.BufferedOutputStream)
  - Izvodi *dodatne odgovornosti* komponente

Suradnja

- Tipično, Dekoratori prosljeđiuju zahtjeve sučelja Komponente sadržanim Komponentama
- Prije i/ili poslije proslijeđenog poziva, dekoratori obavljaju *dodatne operacije* za koje su odgovorni
- Klijenti koji trebaju *konfigurabilnost* po volji dekoriraju konkretnu komponentu dodatnim odgovornostima
- Klijenti koji ne trebaju konfigurabilnost *transparentno* koriste dekoriranu komponentu preko apstraktnog sučelja

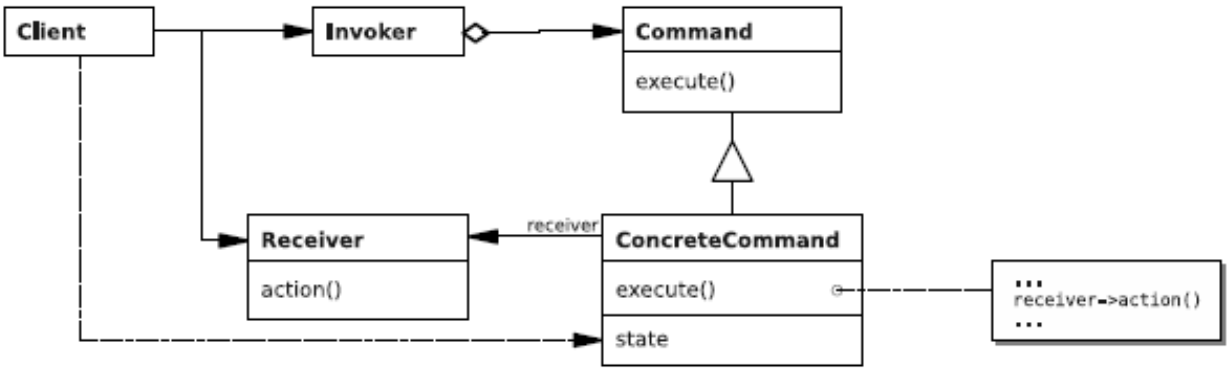
Posljedice

- U odnosu na konfigurabilnu konkretnu komponentu:
  - Klijenti ovise o *apstraktnom* razredu preko minimalnog sučelja (NJO)
  - Aplikacija ne moda održavati svojstva koja se *ne koriste*
- Oprez – dekorirana komponenta nije identična originalu!
- Povećana *složenost*, puno malih objekata

Primjeri upotrebe

- Transparentno dodavanje grafičkih efekata u bibliotekama GUI elemenata
- Konfiguriranje dodatnih funkcionalnosti tokova podataka (streams)
- Konfiguriranje pristupa bazi podataka – sa ili bez logiranja; signalizacija pogrešaka povratnim kodovima i iznimkama
- Python – Modificirati funkciju iz eksterne biblioteke; Debuggirati/logirati/profilirati više funkcija, ali bez ponovljenog koda

Naredba



Koristimo kada

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

Sudionici

- Naredba (Command, listić s narudžbom)
  - Deklarira jednostavno *sučelje* za *izvođenje* operacije
- Konkretna naredba (CmdPaste, ćevapi s lukom)
  - Definira vezu između Pozivatelja i Primatelja
  - Implementira *sučelje* Naredbe *pozivajući* Primatelja
- Pozivatelj (MenuItem, konobar)
  - Inicira zahtjev preko sučelja Naredbe
- Primatelj (MyDocument, kuhar)
  - Zna koje *operacije* mora obaviti kako bi se *zadovoljio zahtjev*
- Klijent (MyApp)
  - Kreaira Konkretnu naredbu i *predaje* je Pozivatelju

Suradnja

- Klijent kreira *konkretnu* naredbu, navodi njenog *primatelja*, te registrira *naredbu* kod Pozivatelja
- Pozivatelj poziva *konkretnu* Naredbu preko *osnovnog sučelja*; Naredba po potrebi sprema stanje kako bi se omogućio njen *naknadni opoziv* (*undo*)
- Konkretne Naredbe *pozivaju* metode Primatelja da *zadovolje zahtjev*
- Pozivatelj i Primatelj se *ne poznaju*

Posljedice

- Naredba *odvaja* inicijatora zahtjeva (Pozivatelja) od objekta koji zna izvršiti potrebnu operaciju (Primatelja)
- Zahtjevnije Naredbe mogu se *složiti od jednostavnih*, kao u slučaju Makro-naredbe. Makro-naredbe odgovaraju obrascu Kompozitu.
- Dodavanje novih Naredbi je lako jer *ne zahtijeva mijenjanje postojećih razreda*
- Ako je interesantno da Pozivatelj može inicirati više Naredbi – dobivamo strukturu sličnu obrascu Promatraču
- Nedostatak – *veliki* broj *malih* razreda (može se ublažiti predlošcima)

Primjeri upotrebe

- Raspoređivanje poslova u stvarnom vremenu (thread pool unutar web servera)
- Bilježenje operacija u cilju omogućavanja oporavka nakon kvara sustava
- Podržavanje opozivih akcija, uključujući transakcijsko poslovanje
- Korisnička sučelja, npr. Java Swing