



Fakulteta za  
informacijske študije  
Faculty of information studies

# Načrtovalski vzorci v programiranju

Seminar pri predmetu Uvod v programiranje

# Avtor seminarja

Gregor Grajzar

- Začel programiranje kot hobi
- Prva zaposlitev kot programer 2002
- 2004 obdelava pravnih dokumentov lusinfo (C#, C++ in drugo)
- 2008 del mednarodne razvojne ekipe 24ur, Kmetija in vrsto drugih portalov v lasti CME Media
- Od leta 2009 delam samostojno.
- Trenutno tudi študent na FIŠ-u

# Agenda

- Kratek uvod
- Zgodovina
- Pogled čez plot oz. kje vse najdemo vzorce
- Izrazoslovje
- Še nekaj o objektih
  - Interface in abstraktni razredi
  - Zelo kratek uvod v UML

Pregled in razdelitev tipičnih vzorcev

- 12 uporabnih vzorcev z uporabo v znanih primerih
- Zaključek
- Kritike pristopa
- Vprašanja



# Opomba

- Programska koda je v nekaterih primerih poenostavljena z namenom jasnosti
- Na prosojnicah je predstavljen le bistven del kode ali strukture
- Poenostavljeni so tudi UML diagrami objektne strukture
- Predstavljeni vzorci so le ena od možnih implementacij te rešitve
- Uporabljeni izrazi v nekaterih primerih niso prevedeni, pogosto uporabljam angleške izraze ob slovenskem prevodu (kar olajša brskanje za tematiko)
- Namen teh prosojnic je predstavitev problematike na čim bolj poenostavljen in jasen način


# Moja inspiracija

## Dude, are you still programming using if...then...else?

14,493,201 members

**CODE PROJECT**  
For those who code

MACHINE LEARNING, CODING HACKS, & MORE  
Learn tips & tricks from our developers.  
[READ ON](#)

Sign in

[home](#) [articles](#) [quick answers](#) [discussions](#) [features](#) [community](#) [help](#)

Search for articles, questions, tips

Articles » Development Lifecycle » Design and Architecture » Design Patterns

Article

[Browse Code](#)

[View Stats](#)

[Revisions](#)

[Comments \(15\)](#)

Posted 21 Dec 2005

Tagged as

C#

Windows

.NET

Visual-Studio

Stats

123.7K views

497 downloads

95 bookmarked

### Dude, are you still programming using if...then...else?

Maxim Astafev

21 Dec 2005

Rate this: ★★★★★ 4.09 (46 votes)

This article shows a concrete example of the true advantages of using design patterns when implementing software.

 [Download source - 18.3 Kb](#)

#### Introduction

Do you hear a lot about software design, software architecture, and design patterns? But you hardly see any striking advantage to use them in your projects? Or you think of them like they are unusable academic nonsense? Or you simply don't have the time to cope with them? What a pity! And that is for many reasons.

This article will show you a concrete example of why you definitely should have a closer look at design patterns again and again. Consider that the complexity of software is steadily increasing. So all of us need methods for keeping our code easy readable, highly maintainable, and easily extensible without having to give up the flexibility of modern programming languages. Design patterns are the very basics which provide us exactly this. Unfortunately, most articles describe design patterns without really pointing out their advantages.

This article is intended to change this. It will show you on a concrete example how you can keep your projects easy extensible and maintainable by using a single design pattern. After reading it, you will know, what the visitor pattern is intended for, where and why you should use it, and what advantages it gives to your projects. In short – you will know what the true meaning of such keywords like maintainability, extensibility, and reusability of code is, and how you can easily add these valuable issues to your own projects.

#### Visitor Pattern

This pattern is a robust and highly scalable way for implementing case distinction in your code. Let us construct some very simple example here. Let us assume that we need to implement a simple insurance software. We have an insurance policy which is related to some person. The policy fee is dependent on the gender of a person. Let us assume that women have an initial fee discount of 20%.

  
**ImageGear**  
Integrate a dev library for rendering, annotating, and manipulating files.  
[Try ImageGear .NET.](#)  
[Download Your Trial](#)





Kako pa drugače, če ne  
if ... then ... else?



# Zakaj objektno programiranje in alternativa

## Temeljni principi OOP:

- Enkapsulacija (izolacija)
- Abstrakcija
- Dedovanje
- Polimorfizem
- Modularnost
- Odvisnost
- ...

## Proceduralno programiranje z uporabo Tabel (array):

```
$rsi_array = array(  
    "name" =>$name,  
    "date"  =>$date,  
    "value" =>$val,  
    "type"  =>gettype ($val));
```





.... in objektno programiranje  
se tu šele začne.







# Kratek pogled v preteklost



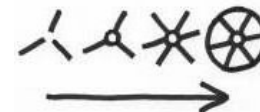
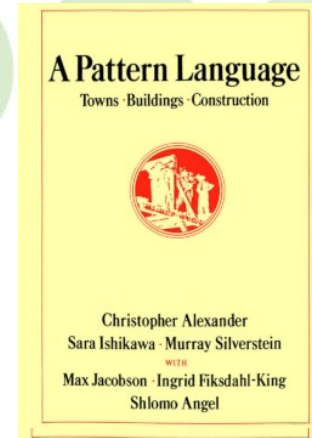
# Inspiracija in začetki objektnih vzorcev

Arhitekt Christopher Alexander napiše dve revolucionarni knjigi, ki opisujeta vzorce v arhitekturi in urbanizmu:

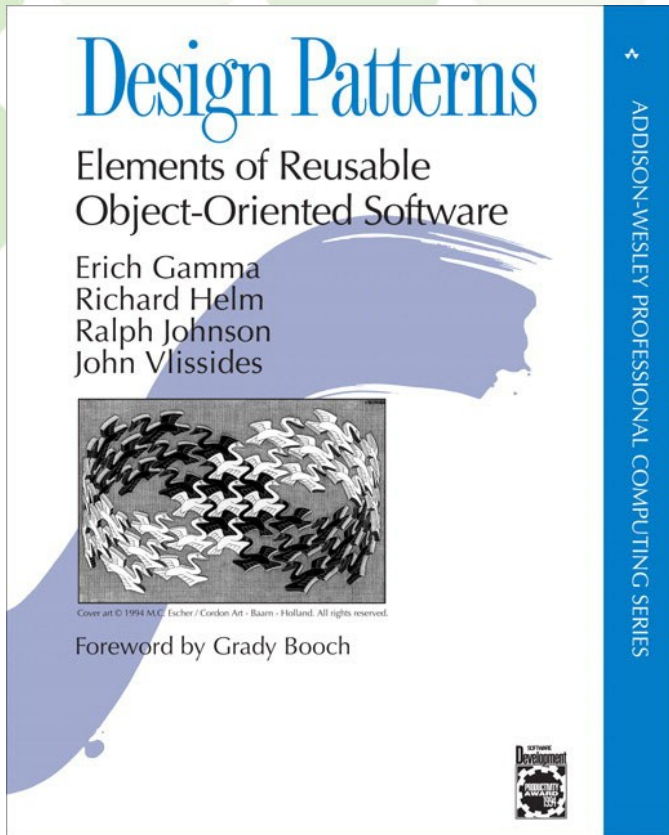
- Pattern Language: Towns, Buildings, Construction (1977)
- The Timeless Way of Building (1979)

Ward Cunningham in Kent Beck napišeta članek:

Using Pattern Language for Object-Oriented Programs (1987)



# Gang of Four Design Patterns



Ali popularno GOF iz leta 1994 je temeljno delo programiranja z objektnimi vzorci in je eno najbolj vplivnih ter citiranih del v svetu računalništva.

Primeri kode: Smalltalk, C++

Urednik knjige profesor Douglas C. Schmidt predava tudi na POSA seminarjih Coursere



# Kratek pogled čez plot

Vzorke najdemo danes v praktično vseh jezikih, ki podpirajo objektno programiranje:

C++, C#, Objective C,  
Java, JavaScript,  
PHP, Python, Swift....

Pri sicer proceduralnem programskem jeziku C so vzorci prav tako izvedljivi:

<https://github.com/huawenyu/Design-Patterns-in-C>

Knjiga: »Object-oriented Programming in ANSI-C«



# Vzorci pri JavaScript-u

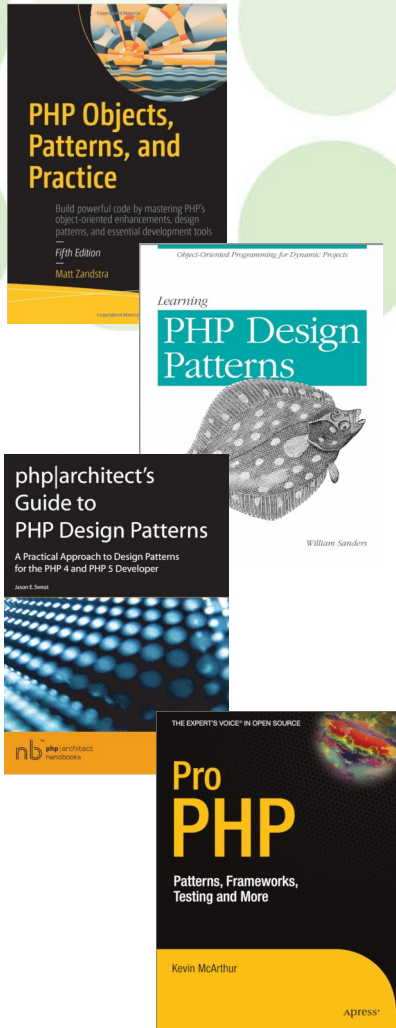
Vzorci so temeljni princip razvoja kode v ozadju trenutno najbolj popularnih JS ogrodij (frameworks):



- Vue.js, Angular.js, React.js, Node.js
- Meteor.js , Backbone.js, Redux.js, Express, GreenSock, D3.js,



# Vzorci pri PHP-ju



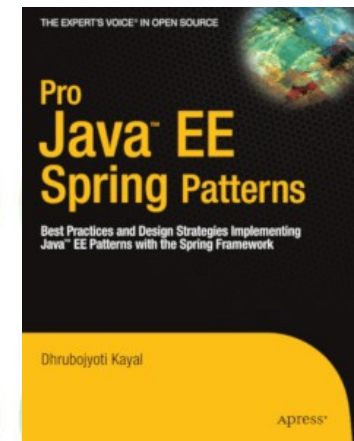
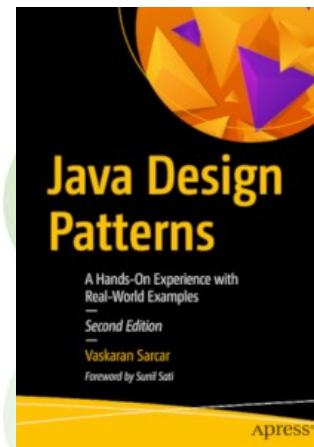
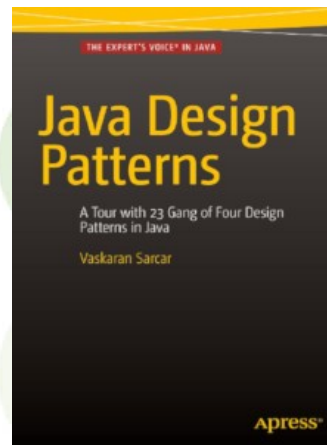
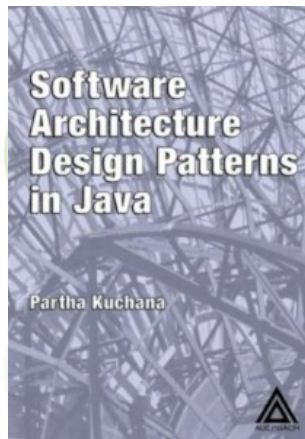
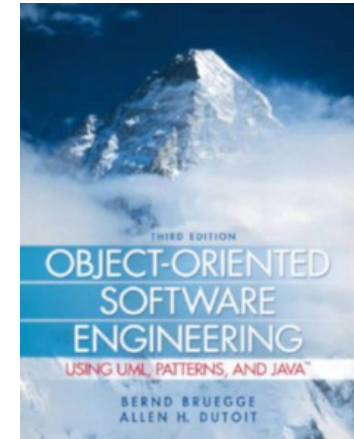
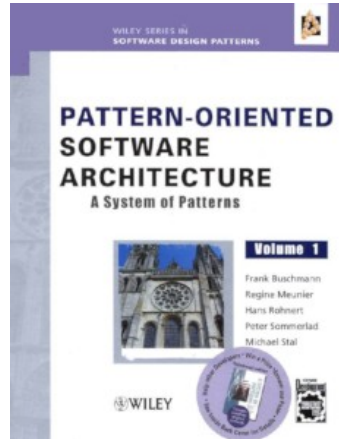
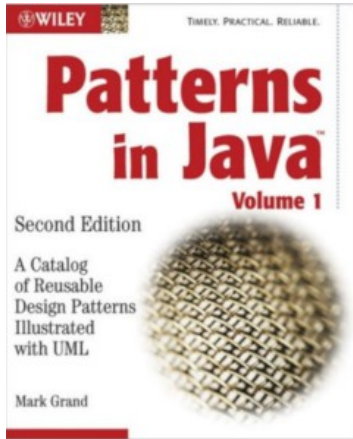
- Veliko PHP projektov (WordPress, Drupal, Joomla...) je nastala v pred PHP5 eri in so do nedavnega upoštevali podporo kode »za nazaj« (večna dilema pri posodobitvah) ter še vedno vsebujejo ogromno kode s proceduralnim pristopom. Vendar:
- PHP5 (2004) bistveno posodobljen objektni model po sodobnih standardih
- PHP7 (2014) do 3x hitrejši od Python-a

Desno: 1.315 vrstic »posodobljene« verzija proceduralne kode neke precej obiskane slov. strani





# Java





# Kratek vpogled v načrtovalske vzorke

In pregled 12 uporabnih vzorcev

V industriji so ji iznašli v trenutku, ko je  
bilo objektno programiranje über-hot.



# Izrazi, ki jih srečamo

In jih uporabljajo v slovenski literaturi

- Načrtovalski vzorci (Design patterns)
- Vzorci načrtovanja objektnih rešitev
- Vzorci oblikovanja (Design Patterns)
- Vzorci oblikovanja programske opreme

Lahko bi rekli tudi: vzorci objektnih struktur, vzorci konstruiranja objektov programiranja, objektni vzorci (object patterns).



# Izrazi v angleškem svetu

Pattern Orientated Software Architecture (POSA)

Na kratko: Design patterns



# Vmesnik (interface)

Vmesnik (interface) **je predpis** metod objekta, ki sam sicer **ne vsebuje kode**. Gre za popolnoma abstrakten objekt brez vsakršne kode.

*// primer objekta interface, ki predpisuje neko poljubno funkcijo*

```
public interface IAvtoUpgrade{
```

```
    public void nekaSuperFunkcija(String poljubniParametri);
```

```
}
```

*// v razredu Avto določimo uporabo funkcije*

```
class Avto implements IAvtoUpgrade, IseEnVmesnik{
```

```
    public void nekaSuperFunkcija(String poljubniParametri) {
```

```
        ..
```

```
    }
```

```
    ....
```



# Abstraktni razred

**Nedokončan razred**, ki vsebuje določene lastnosti in funkcije. Del teh funkcij je nedokončanih in se deklarira na enak način, kot z vmesnikom (interface) a so lahko deli kode že implementirani.

*// v razredu Avto določimo uporabo funkcije*

```
public abstract class OgrodjeAvta {
```

```
    public void nekaPredvidenaFunkcija(String poljubniParametri);
```

```
    // ostal del razreda je izdelan
```

```
class Avto extends OgrodjeAvta{
```

```
    - public void nekaPredvidenaFunkcija(String poljubniParametri) {
```

```
        // implementacija te funkcije
```

```
    }
```

```
    ....
```



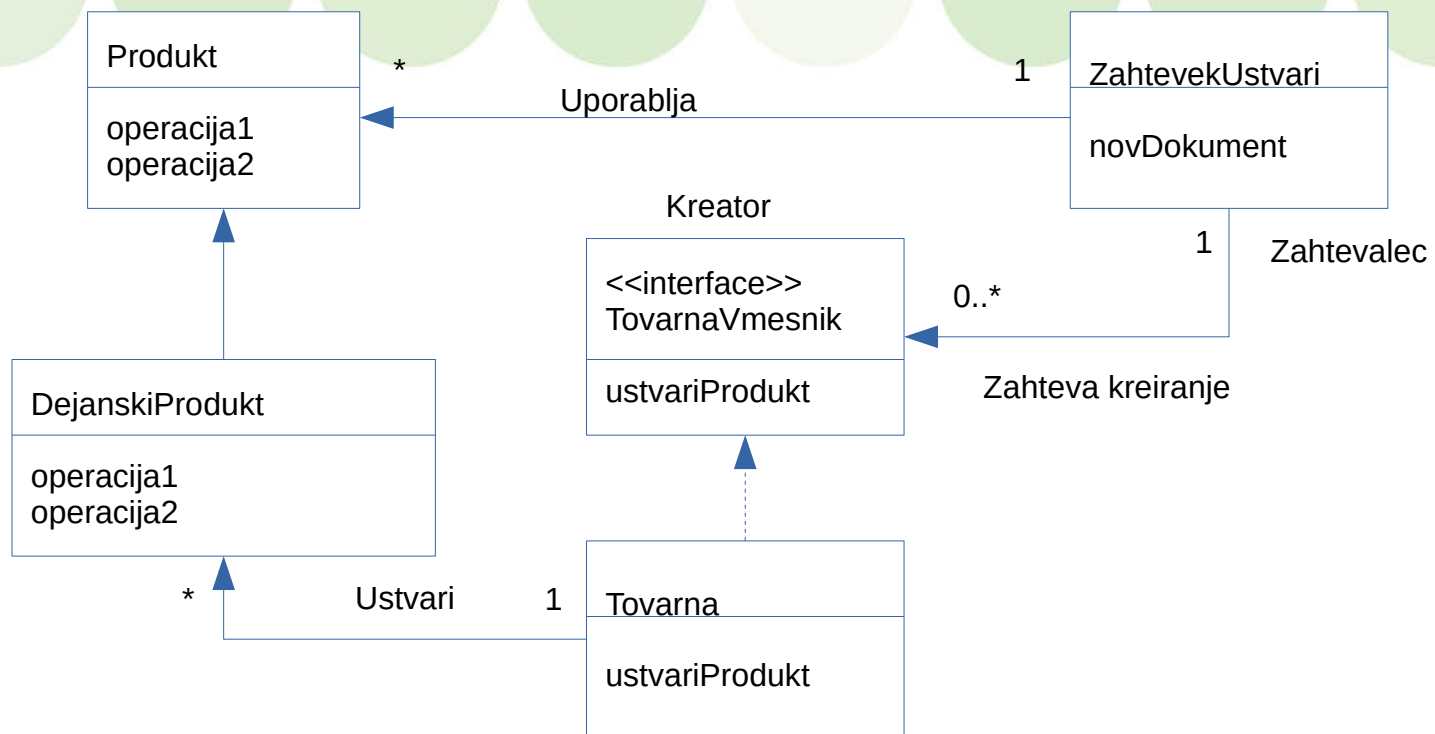
# Uporaba abstraktnih razredov in vmesnikov

```
class Avto extends OgradjeAvta implements IAvtoUpgrade, IPotnik, IPregledovanec {  
    public void nekaPredvidenaFunkcija(String poljubniParametri) {  
        // implementacija te funkcije  
    }  
    public void nekaSuperFunkcija(String poljubniParametri) {  
        ..  
    }  
    ....  
}
```



# UML - Unified Modeling Language

Poenostavljen grafični princip za prikaz skupine objektov ter relacij med njimi.



\* V nekaterih primerih v nadaljevanju so tudi UML diagrami nekoliko poenostavljeni

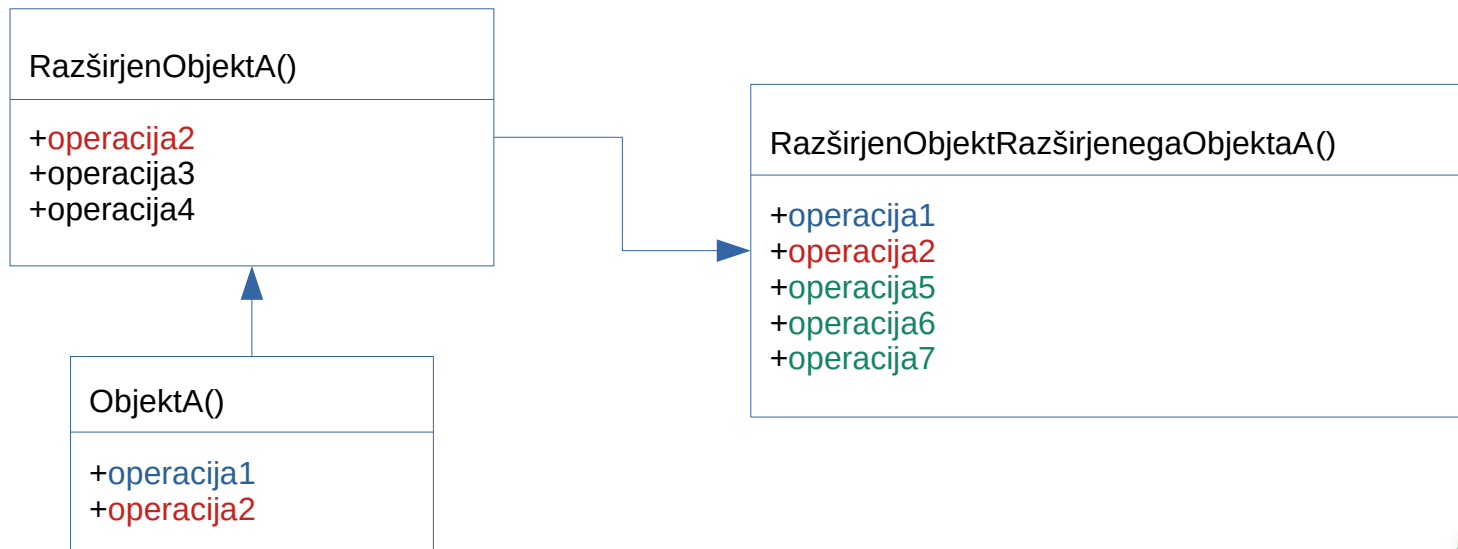




# Izzivi dedovanja razredov v večjih projektih

Poenostavljen primer dedovanja razredov, ko pripeljemo rešitev do »strela v koleno«

Kar je bistveno: **se vedno znova soočamo s podobnimi problemi**, ki smo jih v preteklosti že rešili.



# Pristop z načrtovalskimi vzorci

**Nabor preizkušenih rešitev** za znane programerske izzive.

Lahko bistveno pospešijo razvojni cikel:

- Razbijejo problem na posamezne podprobleme -> **večja abstrakcija kode**.
- Ločitev poslovne logike (business logic) od prezentacijske ter podatkov
- Omogočajo **hitrejše prilagajanje spremembam**.
- **Dinamično spreminjanje objektov med delovanjem**
- Boljši nadzor nad deli kode (**modularnost**)
- Večja **varnost**



# Kaj vzorci niso

- Algoritem
- Implementacija
- Niso omejeni na OOP
- Enaki vsakič, ko jih vidimo



# Osnovna delitev vzorcev po funkcionalnosti

Osnovni tipi programerskih izzivov, ki jih rešujejo vzorci:

- Kreacijski (ustvarjalni) vzorci (creational patterns)
- Razdelitveni (partitional patterns)
- Strukturni (structural patterns)
- Značajski (behavioral patterns)
- Vzporedenjski (concurrency patterns)
- Prezentacijski (npr.MVC)
- Bazni (Database)
- Podjetniški in taki za poslovno logiko (business logic patterns)



# Kreacijski (ustvarjalni) vzorci

## Creational patterns

Rešujejo problematiko kreiranja objektov, kadar je njihovo kreiranje pogojeno z odločitvami.

- **Singleton** (2+)
- **Factory Method** (3+)
- **Abstract Factory**
- **Builder**
- **Prototype** (JavaScript)
- **Object Pool**



# Strukturni vzorci

## Structural patterns

Opisujejo preizkušene načine medsebojne organiziranosti različnih objektov.

- **Adapter**
- **Iterator**
- **Bridge**
- **Facade** (7)
- **Decorator** (6+)
- **Virtual Proxy**
- **Cache Management**



# Značajski vzorci

## Behavioral patterns

Organizirajo, nadzorujejo in kombinirajo vedenje objektnih struktur. Implemetirajo značilne vzorce komunikacije med objekti.

- **Command**
- Chain of Responsibility
- **Interpreter**
- Command
- **Mediator**
- **Observer** (8+)
- **Strategy** (10)
- **Template Method** (12+)
- **Visitor** (9)
- State





# Vzporednostni vzorci

## Concurrency patterns

Rešujejo znane probleme večnitnega programiranja kot so komunikacija med procesi, sinhroniziranje...

- **Single Threaded Execution**
- **Lock Object**
- Guarded Suspension
- Balking
- **Scheduler**
- Read/Write Lock
- Double Buffer
- Asynchronous Processing (AsyncTask)
- Looper
- Hammer framework

\* ti vzorci niso vključeni v teh prosojnicah, ker rešujejo povsem drugačno problematiko programiranja vzporednih sistemov, a postajajo vse bolj aktualni že zaradi prihoda večjedernih procesorjev v vsako prenosno napravo.



# Razdelitveni vzorci

## Partitional Patterns

- **Composite** (4+)
- **Filter** (5+)
- Read-only

Rešujejo izive delitve problemov na manjše enote (deli in vladaj).

## Še nekaj temeljnih in pomembnih vzorcev:

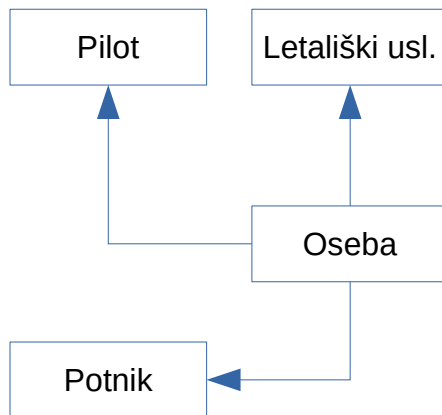
- **Delegation** (1+)
- Interface
- **Abstract Superclass**
- Immutable
- Marker Interface
- Proxy
- MVC



# 1. Delegation (prenašalec ali delegat)

Izziv:

Rešuje težave klasičnega dedovanja, ko postane dedovanje neobvladljivo zaradi količine razredov, ki imajo nekatere podobne lastnosti a so po funkcionalnosti precej drugačne.

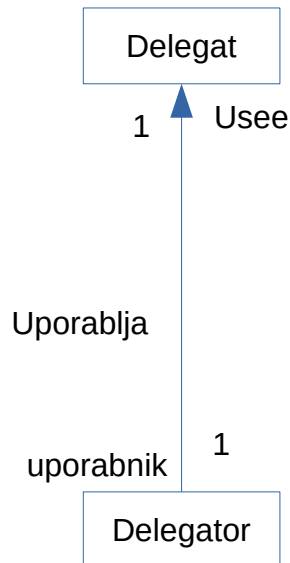


Primer potnikov na letalu, ki imajo povsem različne kompetence a potujejo na istem letu kot ostale osebe-potniki.

# 1. Delegation (prenašalec ali delegat)

rešitev z vzorcem:

Delagator razširi lastnosti objekta s tem, da ima lastnosti osnovnega objekta z instanco osnovnega objekta (delegat) in klicem teh lastnosti.



Primer potnikov na letalu, ki imajo povsem različne kompetence a potujejo na istem letu kot ostale osebe-potniki.

Delegiranje je sicer manj strukturirano kot dedovanje...



# 1. Delegation (prenašalec ali delegat)

primer kode 1.del:

```
public class Avto{  
    private IAvtoUpgrade lastnosti = new AvtoKlasik();  
    public void posodobiLastnost(IAvtoUpgrade novaLastnost) {  
        this.lastnosti = novaLastnost;  
    }  
    public void nekaLastnost() {  
        this.lastnosti.nekaLastnost();  
    }  
    ....  
  
    public interface IAvtoUpgrade{  
        public void nekaLastnost();  
    }  
}
```



# 1. Delegation (prenašalec ali delegat)

primer kode 2.del:

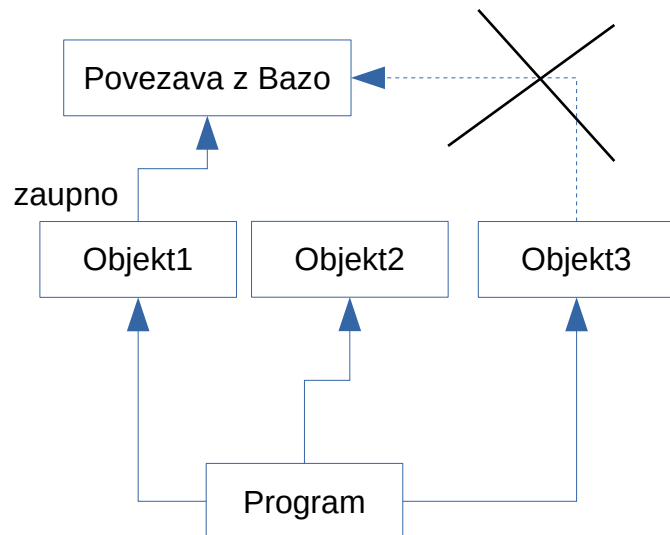
```
class AvtoKlasik implements IAvtoUpgrade{  
    public void nekaLastnost(){  
        System.out.println("Funkcije klasičnega paketa");  
    }...  
  
class SportPaket implements IAvtoUpgrade{{  
    public void nekaLastnost(){  
        System.out.println("Funkcije Športnega paketa");  
    }...  
// uporaba  
  
public static void main(String[] args) throws IOException {  
    Avto clio = new Avto();  
    clio.nekaLastnost();  
  
    // nadgradimo avto za nabavo športnega paketa  
    IAvtoUpgrade sportniPaket = new SportniPaket();  
    clio.posodobiLastnost(sportniPaket);  
    clio.nekaLastnost();  
}
```



## 2.Singleton (edinstveni)

Izziv:

V **Objektu1** uporabljamo znotraj privatne procedure **povezavo z bazo**, da lahko z njo izvedemo neko poizvedbo (query). Povezavo bi želeli uporabljati tudi v **Objektu3** in je **ne želimo vzpostavljati ponovno** (časovno zahtevna operacija).



Hkrati je povezava varnostno kritična operacija in je ne želimo izpostavljati globalno (recimo Objektu2).

Idealno bi bilo, če bi lahko uporabili isto instanco povezave (že vzpostavljene).





## 2.Singleton (edinstveni)

rešitev z vzorcem:

Singleton izkorišča javno statično proceduro znotraj katere kreira edinstveno instanco objekta, ki je ob klicu vedno ista.

```
class SamostojenObjekt{  
    private mojaInstanca = SamostojenObjekt();  
  
    public static SamostojenObjekt getInstance(){  
        if(this.mojaInstanca == null){  
            this.mojaInstanca = new SamostojenObjekt();  
        }  
        return this.mojaInstanca;  
    }  
}
```

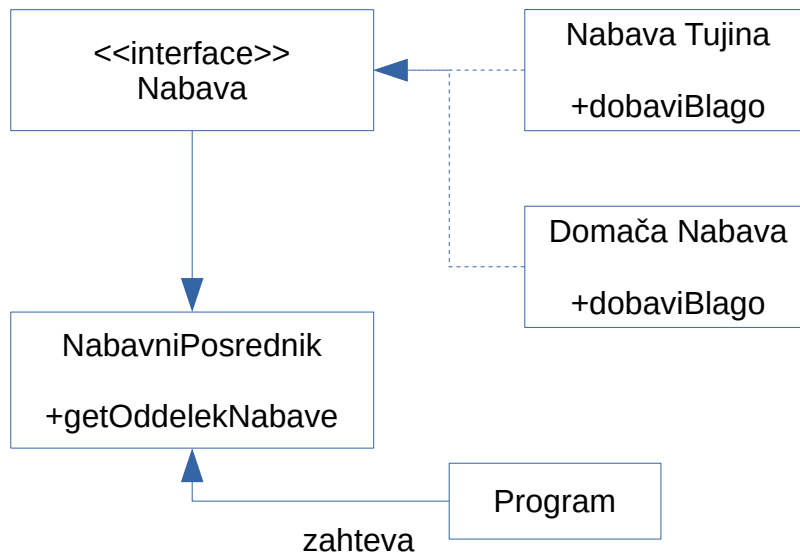


### 3.Factory method (tovarna)

Izziv:

Potrebujemo objekt, ki **dinamično kreira različne objekte** glede na vhodne parametre.

Primer je povezava do baze, ki se glede na parameter type poveže enkrat na eno od baz: MySQL, Oracle, Postgre, MongoDB...



Sama povezava je enostavna a so funkcionalnosti baz toliko različne, da potrebujemo različne načine za dostop do podatkov v nadaljevanju....



# 3.Factory method (tovarna)

primer kode 1.del:

```
public interface Nabava {  
    public void dobaviBlago();  
}
```

```
public class DomacaNabava implements Nabava {  
  
    @Override  
    public void dobaviBlago() {  
        /// funkcije domače dobave  
    }  
}
```

```
public class NabavaTujina implements Nabava {  
  
    @Override  
    public void dobaviBlago() {  
        /// funkcije dobave iz tujine  
    }  
}
```



# 3. Factory method (tovarna)

primer kode 2.del:

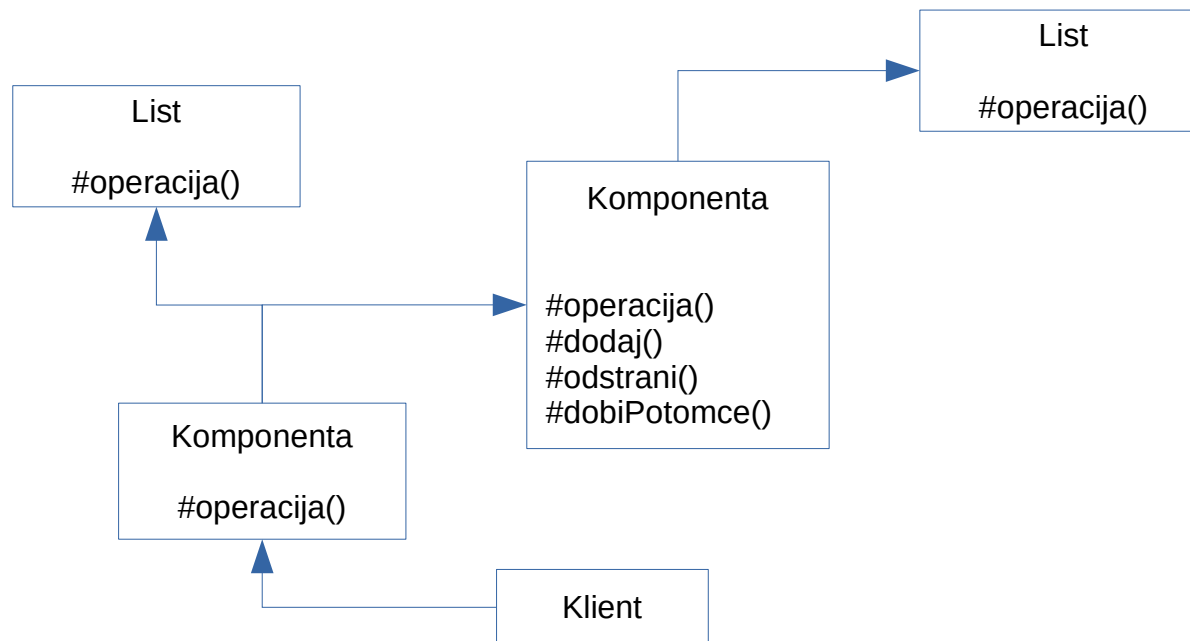
```
class NabavniPosrednik {  
    // tovarniška metoda  
    public Nabava getOddelekNabave(String tip) {  
  
        if(tip.equals("uvoz")) {  
            return new NabavaTujina();  
        } else {  
            return new DomacaNabava();  
        }  
    }  
}  
  
public static void main(String[] args) {  
  
    NabavniPosrednik poisciOddelek = new NabavniPosrednik();  
    // kupujemo domač izdelek  
    Nabava izdelek1 = poisciOddelek.getOddelekNabave("domace");  
    izdelek1.dobaviBlago();  
  
    // kupujemo izdelek iz tujine  
    Nabava izdelek2 = poisciOddelek.getOddelekNabave("uvoz");  
    izdelek2.dobaviBlago();  
}  
}
```



# 4. Composite Design pattern (mešanica)

Izziv:

Vzorec omogoča podobno obravnavo objektov, grupe objektov in njihovih delov, na podoben način. Omogoča postavitev objektov v drevesno strukturo. Recimo, če želimo izdelati hierarhično strukturo oddelkov v podjetju, ki zahteva drevesno predstavitev.



## 4. Composite Design pattern

primer code:

```
class CompanyDirectory implements Employee {  
  
    private List<Employee> employeeList = new ArrayList<Employee>();  
  
    ...  
    public void addEmployee(Employee emp) {  
        employeeList.add(emp);  
    }  
  
    public void removeEmployee(Employee emp) {  
        employeeList.remove(emp);  
    }  
}  
  
class Manager implements Employee{  
    ...  
}  
  
class Developer implements Employee{  
    ...  
}
```



# 5. Filter Pattern

Izziv + prvi del kode:

Filter omogoča **enostavno filtriranje seznamov** (List) glede na različne kriterije.

```
// predpišemo metodo filtra
interface Kriterij {
    public List<Usluzbenec> ustrezaKriteriju(List<Usluzbenec> oseba);
}

// ustvarimo ustrezen filter
class KriterijMoski implements Kriterij {
    @Override
    public List<Usluzbenec> ustrezaKriteriju(List<Usluzbenec> osebe) {

        List<Usluzbenec> moskeOsebe = new ArrayList<Usluzbenec>();
        for (Usluzbenec oseba : osebe) {
            if (oseba.getSpol().equalsIgnoreCase("Moski")) {
                moskeOsebe.add(oseba);
            }
        }
        return moskeOsebe;
    }
}
```

Opomba:

Uporaba je primerna za seznam nepremičnin (seminar), naš primer predstavlja filtriranje seznama bivših in sedanjih uslužbencev glede na status in spol.





# 5.Filter Pattern

drugi del kode:

```
class Usluzbenec {
    private String ime;
    private String spol;
    private String upokojenStatus;

    public Usluzbenec(String ime, String spol, String r) {
        this.ime = ime;
        this.spol = spol;
        this.upokojenStatus = r;
    }
    // ....
}

class DodajKriterij implements Kriterij {

    private Kriterij kriterij;
    private Kriterij drugKriterij;

    public DodajKriterij(Kriterij enKriterij, Kriterij drugKriterij) {
        this.kriterij = enKriterij;
        this.drugKriterij = drugKriterij;
    }
    @Override
    public List<Usluzbenec> ustrezaKriteriju(List<Usluzbenec> osebe) {
        List<Usluzbenec> prviKriterijOsebe = kriterij.ubrezaKriteriju(osebe);
        return drugKriterij.ubrezaKriteriju(prviKriterijOsebe);
    }
}
```



# 5.Filter Pattern

## uporaba kode:

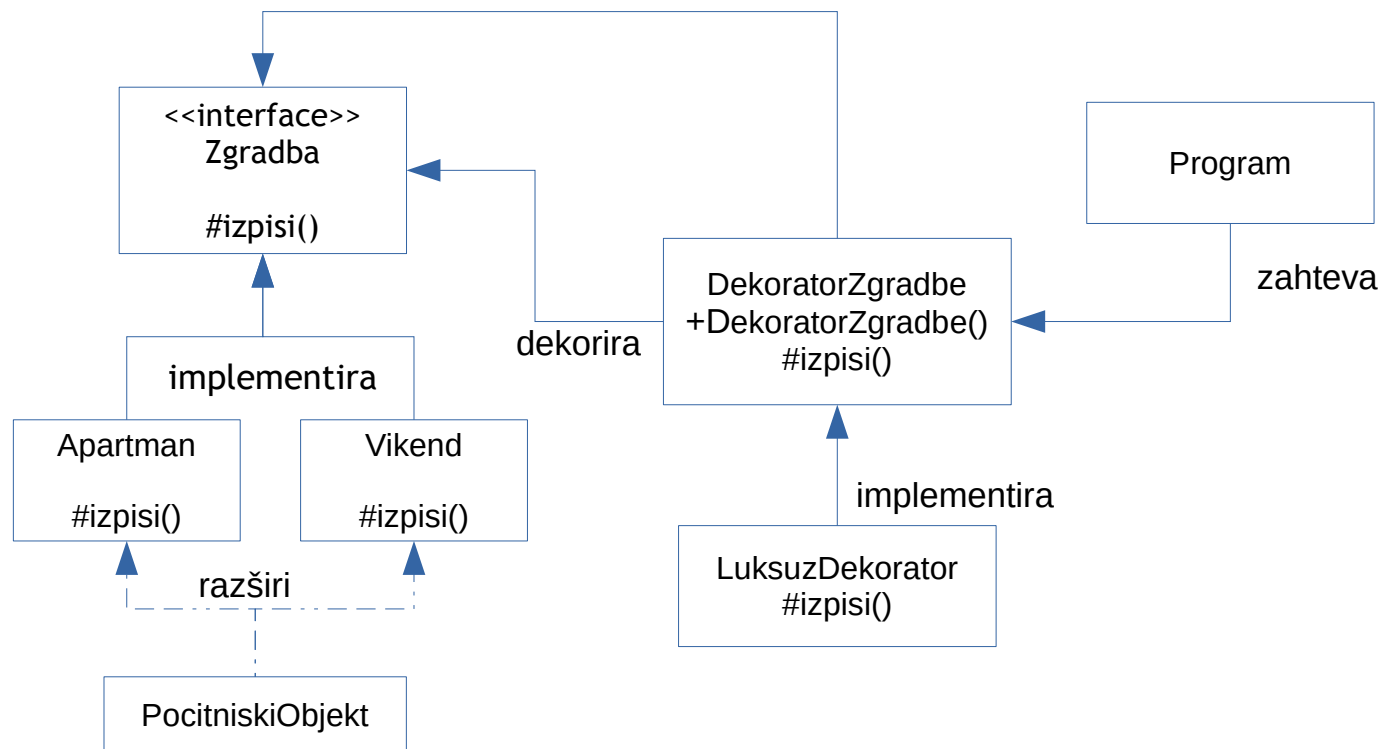
```
public static void main(String[] args) {  
    List<Usluzbenec> osebe = new ArrayList<Usluzbenec>();  
  
    osebe.add(new Usluzbenec("Tone", "Moski", "DA"));  
    osebe.add(new Usluzbenec("Janez", "Moski", "NE"));  
    osebe.add(new Usluzbenec("Ana", "Zenska", "NE"));  
    osebe.add(new Usluzbenec("Kristina", "Zenska", "DA"));  
    osebe.add(new Usluzbenec("Marko", "Moski", "NE"));  
    osebe.add(new Usluzbenec("Brane", "Moski", "DA"));  
  
    Kriterij moski = new KriterijMoski();  
    Kriterij zenske = new KriterijZenske();  
    Kriterij upokojenci = new KriterijUpokojenec();  
    Kriterij upokojeniMoski = new DodajKriterij( upokojenci, moski);  
    Kriterij upokojenAliZenska= new AliKriterij( upokojenci, zenske);  
  
    System.out.println("Moški: ");  
    izpisiOsebe(moski.ustrezaKriteriju(osebe));  
    System.out.println("Ženske: ");  
    izpisiOsebe(zenske.ustrezaKriteriju(osebe));  
    System.out.println("Upokojeni moški: ");  
    izpisiOsebe( upokojeniMoski.ustrezaKriteriju(osebe));  
}  
  
public static void izpisiOsebe(List<Usluzbenec> osebe) {  
    for (Usluzbenec oseba : osebe) {  
        System.out.println(oseba);  
    }  
}
```



# 6.Decorator

## Izziv in UML rešitve:

Decorator je oblika vzorca , ki »**okrasi**« **objekt z novimi lastnostmi**, ki jih dodajamo dinamično. Na primer: počitniški objekt, ki je lahko apartma ali vikend, a ga dekoriramo z lastnostmi luksuzne vile.



## 6.Decorator rešitev:

```
public interface Zgradba {  
    void izpisi();  
}
```

```
public class Apartman extends PocitiniskiObjekt implements Zgradba {  
    @Override  
    public void izpisi() {  
        System.out.println("Zgradba: Apartman");  
    }  
}
```

```
public abstract class DekoratorZgradbe implements Zgradba {  
    protected Zgradba dekoriranaZgradba;  
  
    public DekoratorZgradbe(Zgradba dekoriranaZgradba){  
        this.dekoriranaZgradba = dekoriranaZgradba;  
    }  
  
    public void izpisi(){  
        dekoriranaZgradba.izpisi();  
    }  
}
```



## 6.Decorator

### rešitev in uporaba:

```
public class LuksuzDecorator extends DekoratorZgradbe {  
  
    public LuksuzDecorator(Zgradba dekoriranaZgradba) {  
        super(dekoriranaZgradba);  
    }  
  
    @Override  
    public void izpisi() {  
        dekoriranaZgradba.izpisi();  
        setVilla(dekoriranaZgradba);  
    }  
  
    private void setVilla(Zgradba dekoriranaZgradba){  
        System.out.println("Objekt je: Villa");  
    }  
}
```

```
/// uporaba  
Zgradba appartma1= new Apartman();  
Zgradba vikend1 = new Vikend();  
Zgradba luksuznaVila = new LuksuzDecorator(new Vikend());  
appartma1.izpisi();
```



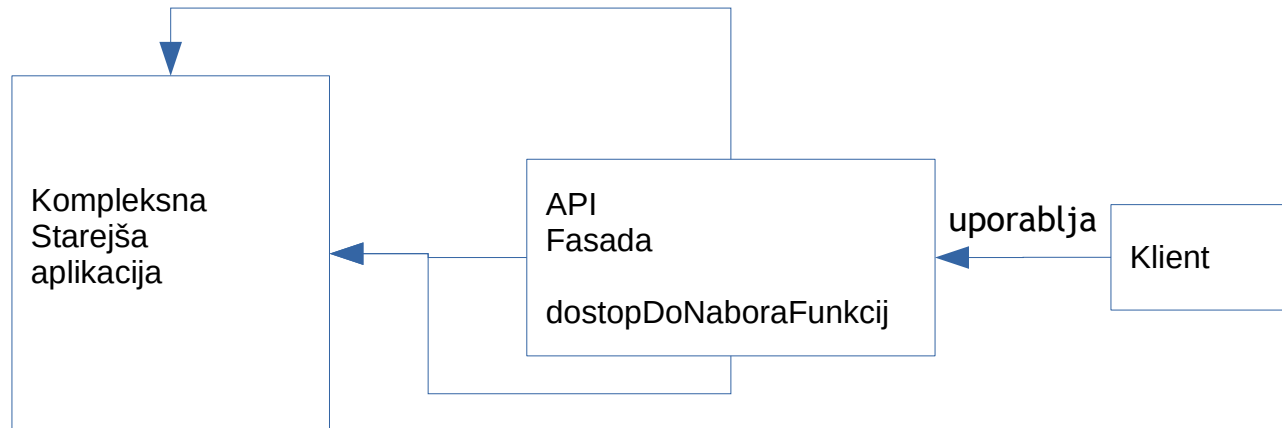
# 7.Facade (fasada)

Izziv:

Vzorec »fasada« odpre posamezne metode objekta izza njega nov vmesnik (enkapsulira).

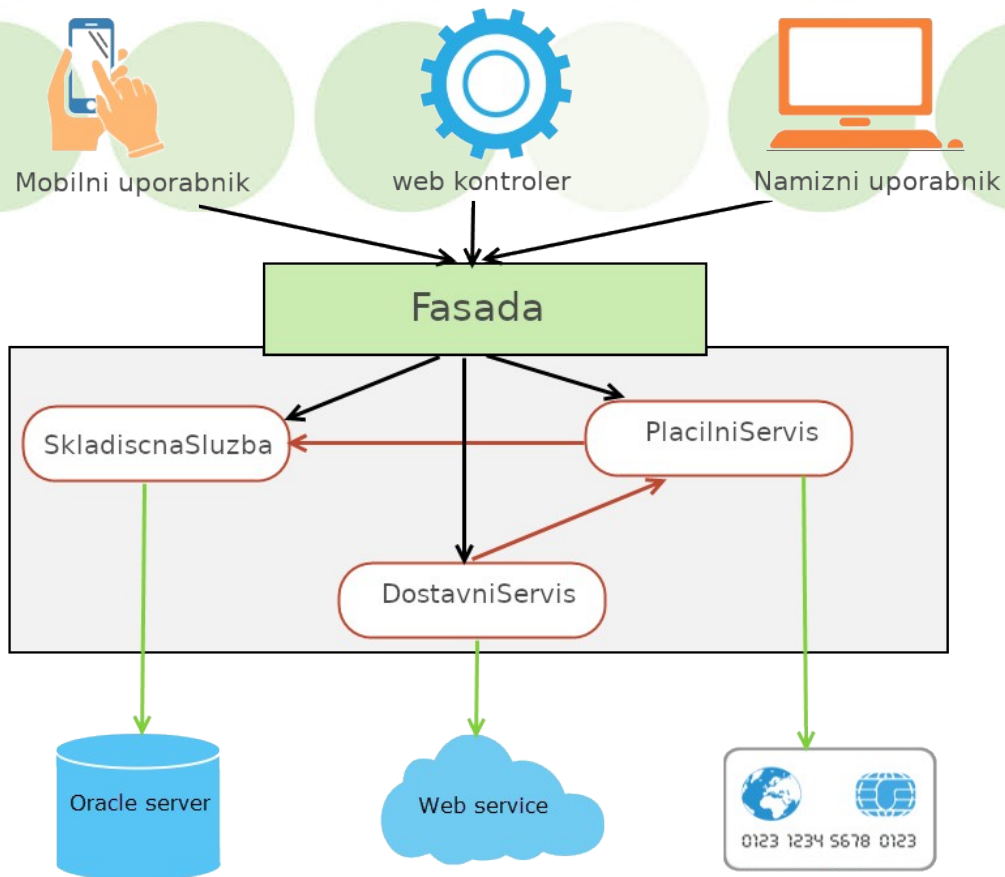
Uporablja se lahko za:

- izdelavo univerzalnega API-ja za dostop do kompleksnejših sistemov
- izdelavo varnostnega ovoja s podobno funkcijo, kot jo izvaja vse popularnejši Web Application Firewall



# 7.Facade (fasada)

Rešitev:



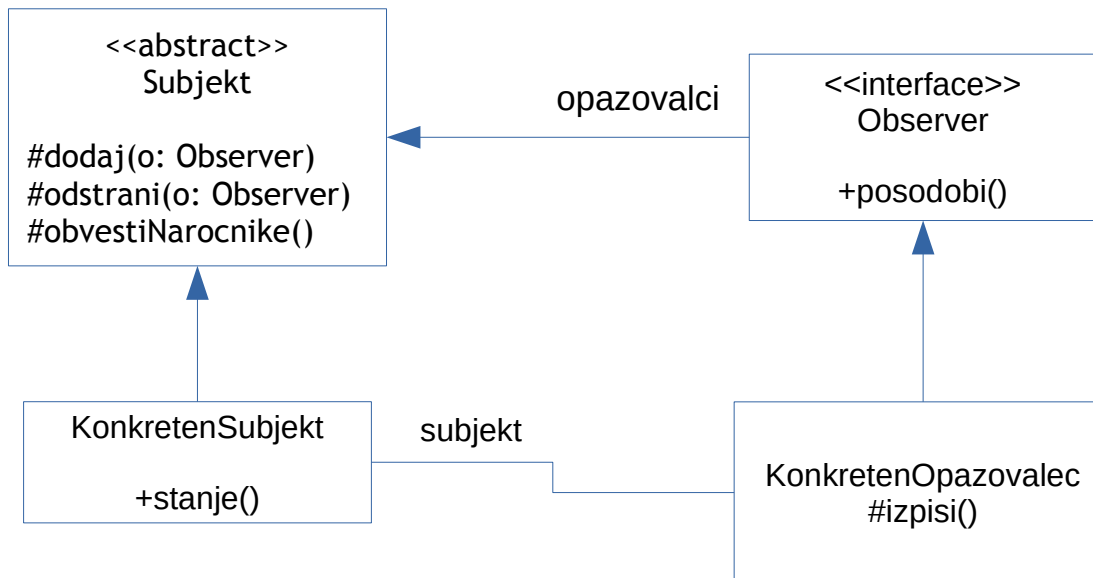


# 8.Observer pattern (opazovalec)

Izziv:

Zelo poznan in široko uporaben ter dobro dokumentiran značajski vzorec. Vzorec vzpostavlja komunikacijo med dvema objektoma: **opazovalec** in **opazovanec**. **Opazovanec obvešča opazovalce** o dogodku ali spremenjenem stanju.

**Opazovalce dinamično dodajamo in odstranjujemo z opazovanja.**



# 8.Observer pattern (opazovalec)

Implementacija kode:

```
public interface Observer{  
    public void posodobi(Sporocilo m);  
}  
  
public interface Subjekt {  
    public void dodaj(Observer o);  
    public void odstrani(Observer o);  
    public void obvestiNarocnike(Sporocilo m);  
}  
  
public class Sporocilo {  
    final String vsebinaSporocila;  
  
    public Sporocilo (String m) {  
        this.vsebinaSporocila = m;  
    }  
  
    public String getVsebinaSporocila() {  
        return this.vsebinaSporocila;  
    }  
}
```



# 8.Observer pattern (opazovalec)

## Implementacija kode 2:

```
public class NarocnikEna implements Observer {  
    @Override  
    public void posodobi(Sporocilo m) {  
        System.out.println("NarocnikEna :: " + m.getVsebinaSporocila());  
    }  
}
```

```
public class Obvescevalec implements Subjekt {  
  
    private List<Observer> observers = new ArrayList<>();  
    @Override  
    public void dodaj(Observer o) {  
        observers.add(o);  
    }  
  
    @Override  
    public void odstrani(Observer o) {  
        observers.remove(o);  
    }  
  
    @Override  
    public void obvestiNarocnike(Sporocilo m) {  
        for(Observer o: observers) {  
            o.posodobi(m);  
        }  
    }  
}
```



# 8.Observer pattern (opazovalec)

Uporaba:

```
NarocnikEna n1 = new NarocnikEna();  
NarocnikDva n2 = new NarocnikDva();  
NarocnikTri n3 = new NarocnikTri();
```

```
//MessagePublisher  
Obvescevalec p = new Obvescevalec();
```

```
p.dodaj(n1);  
p.dodaj(n2);
```

```
//n1 in n2 bosta prejela posodobitev  
p.obvestiNarocnike(new Sporocilo("Prvo sporočilo"));
```

```
System.out.println("Odstranim naročnika Ena in dodam Tri");  
p.odstrani(n1);  
p.dodaj(n3);
```

```
//n2 in n3 bosta prejela posodobitev  
p.obvestiNarocnike(new Sporocilo("Drugo sporočilo"));
```



## 9. Visitor pattern (obiskovalec)

Izziv:

Če Observer obvešča druge objekte o svojih spremembah pa je **visitor objekt, ki ga objektu dinamično pripnemo** (ga obišče) ter ga s tem nadgradimo z novo funkcionalnostjo.

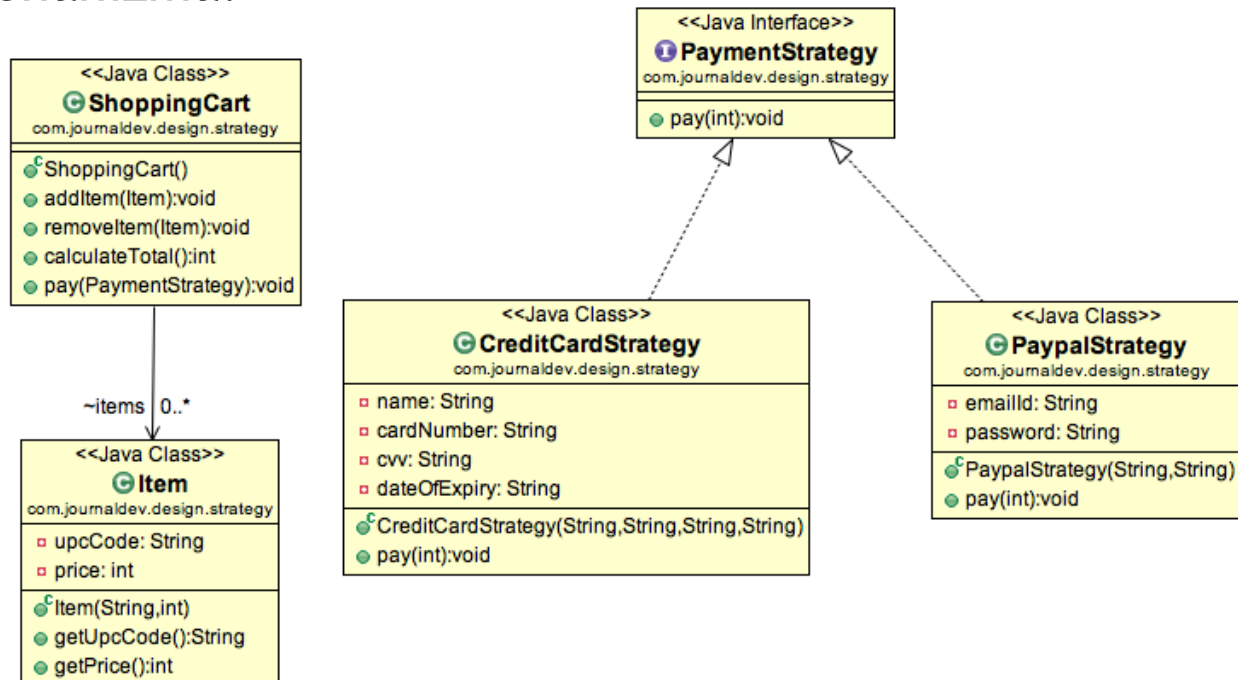


# 10.Strategy pattern (strategija)

## Izziv:

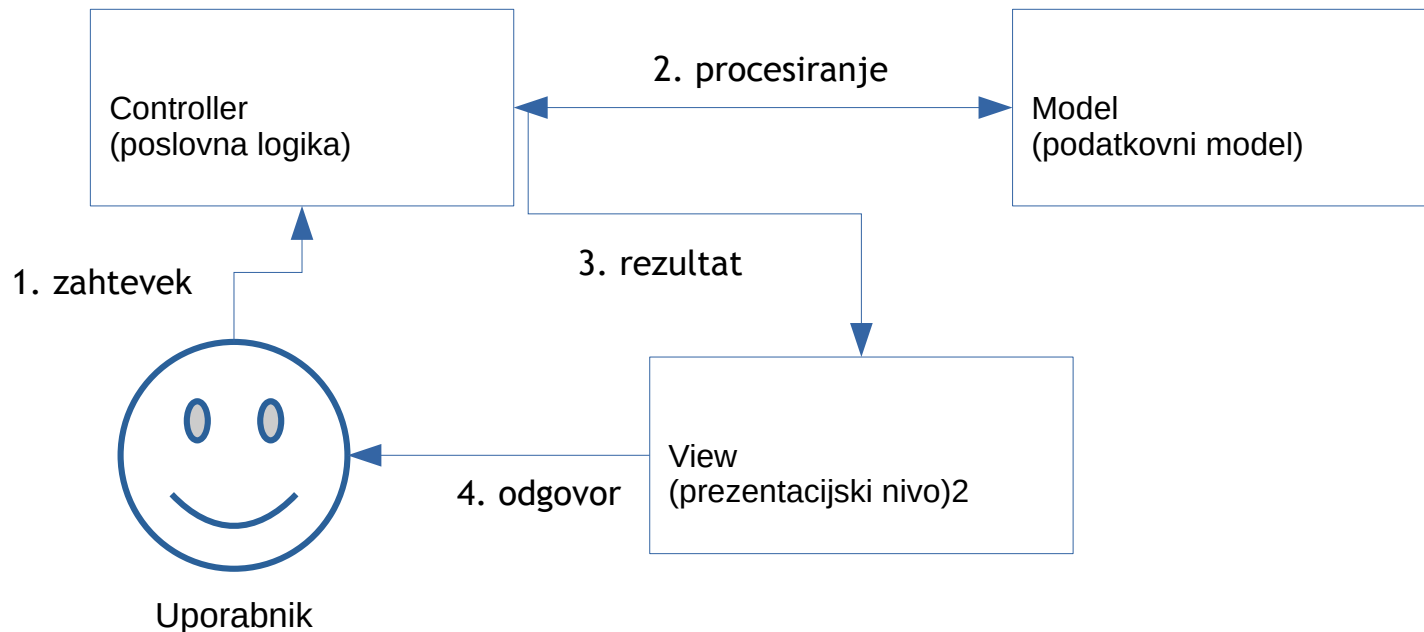
Še en značilen značajski vzorec je »strateški vzorec«, ki omogoča implementacijo različnih strategij glede na vhodne parametre.

Spodaj je kratek UML primer strategije plačilnega mehanizma:



# 11.MVC pattern

Splošno poznan je tudi **MVC** model (Model View Controler), ki omogoča ločitev prezentacijskega nivoja (template za prikaz -> view), podatkovnega modela (model) in poslovne logike, ki jo v tem modelu prevzema Controller.





# 12.Template Method 1. del

Kadar imamo standarden sistem postopkov, jih lahko zapakiramo...

```
public class TemplateGof {  
  
    public static abstract class AbstractResourceManipulatorTemplate {  
        protected Resource resource;  
  
        private void openResource() {  
            resource = new Resource();  
        }  
  
        protected abstract void doSomethingWithResource();  
  
        private void closeResource() {  
            resource.dispose();  
            resource = null;  
        }  
  
        public void execute() {  
            openResource();  
            try {  
                doSomethingWithResource();  
            } finally {  
                closeResource();  
            }  
        }  
    }  
}
```



# 12.Template Method 2.del

```
public static class ResourceUser extends AbstractResourceManipulatorTemplate {  
    @Override  
    protected void doSomethingWithResource() {  
        resource.useResource();  
    }  
}  
  
public static class ResourceEmployer extends AbstractResourceManipulatorTemplate {  
    @Override  
    protected void doSomethingWithResource() {  
        resource.employResource();  
    }  
}  
  
public static void main( String[] args ) {  
    new ResourceUser().execute();  
    new ResourceEmployer().execute();  
}
```



# Povzetek

1. Načrtovalski vzorci nam pri programiranju omogočajo večjo abstrakcijo kode in **delitev posameznih logičnih sklopov (modularnost)**.

2. Ko imamo vzorec enkrat zasnovan, je dodajanje funkcionalnosti relativno preprosto (**dinamičnost in nadgradljivost**).

3. **Dinamično spreminjanje objektov med delovanjem**

4. Omogočajo večjo separacijo občutljivih delov kode, kar pomeni večjo **varnost**.

5. **Hitrejše razvojne cikle**

Sama izvedba vzorcev zahteva večjo **izkušnost** programerjev.



# Kritike

Čeprav najdemo danes vzorce v mnogoterih aplikacijah, frameworkih in so v splošnem precej uporabni, ima ta pristop tudi precej kritike... Če navedem samo nekaj razlogov:

- **Paternitis** – pretirana uporaba vzorcev vodi v posebno vrsto obsedenosti in preobilja (redundency)
- **Neberljivost** – nepoznavalcem je koda zaradi visoke stopnje abstrakcije precej neberljiva
- **Podobnost posameznih vzorcev** – znanih vzorcev je danes že ogromno in se mnogokrat razlikujejo le v niansah
- Oblikovanje vzorcev ti sicer daje idejo a ne vedno tudi rešitve.
- Zaradi hitrih sprememb v industriji, **zahteva razvoj hitrejšre rešitve**
- **Drago vzdrževanje** zaradi prve alineje
- Brisanje starih rešitev s kompleksnimi novimi **ni lahka in poceni rešitev**
- ...



*Designing a software is basically an art. And there is no definition or criteria for best art.*

*Ena od izjav kritikov iz knjige: Java Design Patterns*





*Vprašanja?*



*Hvala lepa!*

*Vsem želim uspešno programiranje  
ter odlično zaključeno izpitno obdobje.*

