Objektno Orijentisano Programiranje 1

Design patterns

Design patterns

- Šabloni za rešavanje tipskih problema
- Knjiga: **"Design Patterns: Elements of Reusable Object-Oriented Software"**, Ralph Johnson, John Vlissides, Richard Helm i Erich Gamma
- Vrste:
  - za kreiranje (creational)
  - strukturni (structural)
  - za ponašanje (behavioral)

ŠABLONI ZA KREIRANJE

Šabloni za kreiranje

- **Singleton**
- **Factory**
- Abstract factory
- Builder
- Prototype

Singleton

- Garantuje da će postojati samo jedna instanca neke klase
- Ideja:
  - da bismo sprečili da neko pozove konstruktor, napravićemo da bude **private**
  - napravićemo **public static** metodu koja će vratiti tu jedinu instancu

Primer

```
public class Singleton {
  private Singleton() {
  }
  private static Singleton instance = null;
  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Primer

```
public class Test {

  public static void main(String[] args) {
    Singleton s1 = Singleton.getInstance();
    s1.hello();

    Singleton s2 = Singleton.getInstance();
    s2.hello();
  }

}
```

Factory

- Predstavlja "fabriku" objekata neke klase
- Klijent ne instancira direktno objekte neke klase, već ih dobija iz fabrike
  - SocketFactory, itd.

Primer

```
public class FabrikaVozila {
  public static final int AUTOMOBIL = 0;
  public static final int KAMION    = 1;
  public static Vozilo kreirajVozilo(int tip) {
    switch (tip) {
    case AUTOMOBIL: return new Automobil();
    case KAMION:    return new Kamion();
    default:        return null;
    }
  }
}
```

Primer

```
public class Test {
  public static void main(String[] args) {
    Vozilo auto = FabrikaVozila.kreirajVozilo(
                            FabrikaVozila.AUTOMOBIL);
    auto.vozi();
    Vozilo kamion = FabrikaVozila.kreirajVozilo(
                             FabrikaVozila.KAMION);
    kamion.vozi();
  }
}
```

Abstract factory

- Predstavlja apstraktnu "fabriku" objekata neke klase
- Klijent prvo mora da dobije odgovarajuću fabriku, a onda da iz nje dobije traženi objekat

Builder

- Koristi se kada je potrebno napraviti instancu klase, ali i postaviti neke atribute ili pozvati odgovarajuće metode
- Umesto da klijent kreira objekat, pa ga onda podešava, Builder fabrika će to uraditi za njega
- Ako postoji više različitih klasa, koje se podešavaju na isti način, onda se koristi ovaj šablon

Prototype

- Umesto da se klasa instancira, njen objekat se klonira
  - izbegava se komplikovana inicijalizacija
  - ako postoji više objekata iste klase, a različitog internog stanja, ovako se lakše prave
    - naprave se unapred u fabrici prototipova, pa se onda kloniraju po potrebi

Clone u Javi

- Ako klasa implementira interfejs **Cloneable**, onda mora da redefiniše metodu **clone()**
- Metoda **clone()** postoji u klasi **Object**, ali je **protected**
- Metoda **clone()** vraća kopiju originalnog objekta bajt po bajt
  - primitivni atributi se kopiraju bajt-po-bajt
  - neprimitivni atributi (reference) u kopiji ukazuju na isti objekat kao i kod originala – shallow copy

STRUKTURNI ŠABLONI

Strukturni šabloni

- **Fasada (Facade)**
- **Adapter**
- Most (Bridge)
- *Kompozitni (Composite)*
- Dekorator (Decorator)
- Flyweight
- Proksi (Proxy)

Fasada

- Sakriva od klijenta kompleksnu strukturu klasa i njihovih poziva
- Ako je potrebno da se instancira veći broj klasa i/ili da se pozove veći broj metoda tih klasa, to može da uradi fasada
  - klijent samo pozove metodu fasade, a ona odradi kompleksan posao

Primer

```
public class Racunar {
  final int BOOT_ADRESA = 1024;
  Procesor procesor = new Procesor();
  Memorija memorija = new Memorija();
  Disk     disk     = new Disk();
  public void start() {
    disk.zavrti();
    memorija.ucitajBootSektor(
                  disk.vratiBootSektor());
    procesor.skociNa(BOOT_ADRESA, memorija);
  }
}
```

Primer

```
public class Test {
  Racunar racunar;
  public Test() {
    racunar = new Racunar();
    racunar.start();
  }

  public static void main(String args[]) {
    new Test();
  }
}
```

Adapter

- Spaja dve klase
  - ne možemo da menjamo ni jednu klasu
- Realizuje se kao klasa koja sa jedne strane izgleda kao jedna klasa (implementira interfejs ili nasleđuje tu klasu), a sa druge strane nasleđuje ili sadrži drugu klasu
- Dve realizacije:
  - preko atributa (asocijacija i kompozicija)
  - upotrebom nasleđivanja (u slučaju jave: nasleđivanje i implementacija)

Adapter

Adapter preko atributa

- Adapter implementira interfejs sa jedne strane, a drugu stranu sadrži kao atribut
  - poziv metode definisane interfejsom prevodi u poziv odgovarajuće metode atributa

Primer-atribut

```
public interface USBTastatura {
  public int vratiTaster();
}
public class Racunar {
  USBTastatura tastatura;
  public void testTastature() {
  System.out.println(
     tastatura.vratiTaster());
  }
}
```

Primer-atribut

```
public class PS2Tastatura {
  public int vratiTaster() {
    System.out.println(
          "PS2 tastatura vraća taster.");
    return 65;
  }
}
public class Test {
  public static void main(String[] args) {
    Racunar racunar = new Racunar();
    racunar.tastatura = new PS2ToUsbAdapter(
                                    ps2tastatura);
    racunar.testTastature();
  }
}
```

Primer-atribut

```
public class PS2ToUsbAdapter implements USBTastatura
{
  PS2Tastatura tastatura;
  public PS2ToUsbAdapter(PS2Tastatura
                         tastatura) {
    this.tastatura = tastatura;
  }
  @Override
  public int vratiTaster() {
    return tastatura.vratiTaster();
  }
}
```

Adapter preko nasleđivanja

- Adapter implementira interfejs sa jedne strane, a nasleđuje klasu sa druge strane
  - nema višestrukog nasleđivanja
- Poziv metode definisane interfejsom prosleđuje na poziv metode roditelja

Primer-nasleđivanje

```
public interface USBTastatura {
  public int vratiTaster();
}
public class Racunar {
  USBTastatura tastatura;
  public void testTastature() {
  System.out.println(
     tastatura.vratiTaster());
  }
}
```

Primer-nasleđivanje

```
public class PS2Tastatura {
  public int vratiTaster() {
    System.out.println(
          "PS2 tastatura vraća taster.");
    return 65;
  }
}
public class Test {
  public static void main(String[] args) {
    Racunar racunar = new Racunar();
    racunar.tastatura = new PS2ToUsbAdapter();
    racunar.testTastature();
  }
}
```

Primer-nasleđivanje

```
public class PS2ToUsbAdapter extends
PS2Tastatura implements USBTastatura {
  @Override
  public int vratiTaster() {
    return super.vratiTaster();
  }
}
```

Bridge

- Za razdvajanje specifikacije i implementacije
- I specifikacija i implementacija mogu da se dodatno nasleđuju i redefinišu, ali su nezavisni jedno od drugog

Composite

- Predstavlja kompozitnu komponentu koja ima rekurzivnu hijerarhijsku internu strukturu
- Poput stabla
  - svaki element je ili list, ili sadrži svoje podelemente

Decorator

- Dodavanje novih atributa ili metoda u nekoj klasi, ali ne preko nasleđivanja
  - ovako je moguće dinamički (run-time) proširiti funkcionalnost neke klase
- Nova klasa ne nasleđuje klasu koju proširuje, već nasleđuje zajedničkog pretka, a sadrži u sebi (kao atribut) objekat klase koju proširuje
- Primeri: DataInputStream, BufferedInputStream, JScrollPane u Javi

FlyWeight

- Koristi se kada imamo potrebu za velikim brojem objekata, ali je skupo da ih sve napravimo (potrošnja memorije, pre svega)
- Umesto ručnog pravljenja velikog broja objekata, koristićemo već napravljene objekte (iz nekog repozitorijuma)
  - moguće samo ako se ne koriste svi istovremeno
  - ovih objekata ima značajno manje

Proksi

- Jednostavan predstavnik željene (kompleksne) klase
- Klijent radi sa proksijem, a zapravo se izvršavaju metode u kompleksnom objektu
- Nekad se kreiranje kompleksnog objekta odlaže, pa se koristi jednostavan predstavnik
- Nekad su jednostavan i kompleksan predstavnik razdvojeni na različitim računarima
- Primer: Java RMI (Remote Method Invocation) se zasniva na ovom konceptu

ŠABLONI ZA PONAŠANJE

Šabloni za ponašanje

- **Komanda (Command)**
- **Observer**
- **Iterator**
- **State**
- Lanac odgovornosti (Chain of Responsibility)
- Interpreter
- Mediator
- Uspomena (Memento)
- Strategy
- Template Method
- Visitor

Command

- Predstavlja sistem za izvršavanje komandi, gde je svaka komanda predstavljena odgovarajućom klasom
- Sve komande nasleđuju apstraktnu komandu, koja ima metodu execute()
- Ako želimo da obezbedimo undo/redo funkcionalnost, moramo da implementiramo jednu od dve dodatne funkcionalnosti:
  - pre izvršenja svake komande, zapamtiti trenutno stanje (ponekad nepraktično – obrada slike ili filma)
  - zapamtiti izvršenu komandu u nekakvoj kolekciji i implementirati inverznu funkciju ove komande

Primer

```
public abstract class Command {
  public File currentDir;
  public List<String> parameters;
  public Command() {
    currentDir = new File(".");
    parameters = new ArrayList<String>();
 }
  /** Izvršava komandu.
   * Ako vrati <b>false</b>, program će završiti.
   * @return da li da ostane u programu ili ne
   */
  public abstract boolean execute();
}
```

Primer

```
public class DirCommand extends Command {
  @Override
  public boolean execute() {
    for(File f : currentDir.listFiles()) {
      System.out.println(f.getName());
    }
    return true;
  }
}
```

Primer

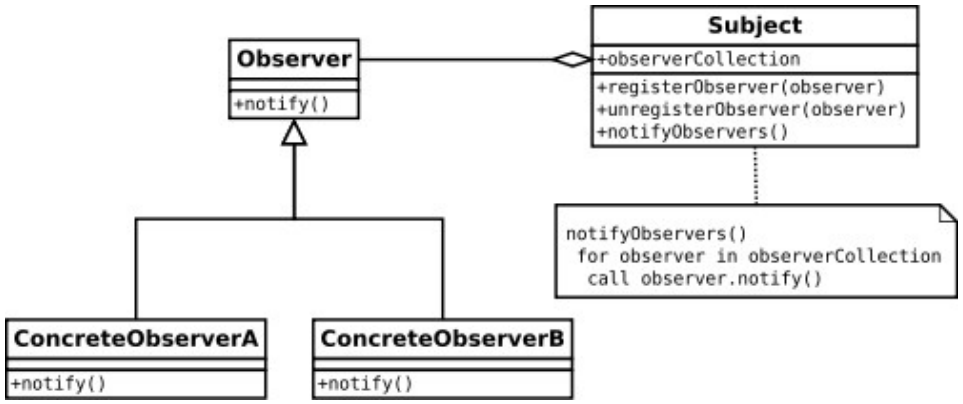
```
public class ExitCommand extends Command {
  @Override
  public boolean execute() {
    //System.exit(0);
    return false;
  }
}
```

Primer

```
Parser parser = new Parser();
Scanner sc = new Scanner(System.in);
while (true) {
  System.out.print(">");
  String line = sc.nextLine();
  Command cmd = parser.parse(line);
  if (cmd != null) {
    if (cmd.execute() == false)
      break;
  } else
    System.out.println("Unknown command.");
}
sc.close();
```

Observer

- Za nadgledanje promene stanja objekta
- Kada god se stanje objekta promeni, obaveštava se posmatrač, koji se prethodno registrovao
- Primer:
  - Swing GUI – svaka akcija korisničkog interfejsa ima odgovarajući observer, koji se zove *Listener*



Observer

- Objekat koji se nadgleda može da implementira Observable interfejs
  - sadrži metode za dodavanje/uklanjanje posmatrača
- Objekat koji posmatra implementira interfejs Observer
  - sadrži metode koje bivaju pozvane kada se posmatra

Primer

```
public interface Observable {
  public void register(Observer observer);
  public void unregister(Observer observer);
}

public interface Observer {
  public void update(int newState);
}
```

Primer

```
public class Timer implements Observable {
  List <Observer> observers;
  public Timer() {
    observers = new ArrayList<Observer>();
  }
  @Override
  public void register(Observer observer) {
    observers.add(observer);
  }
  @Override
  public void unregister(Observer observer) {
    observers.remove(observer);
  }
...
```

Primer

```
...
  @SuppressWarnings("static-access")
  public void start() {
    int seconds = 0;
    while(true) {
      try {
        Thread.currentThread().sleep(1000);
        seconds++;
        for (Observer o : observers)
          o.update(seconds);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Primer

```
public class Clock implements Observer {

  @Override
  public void update(int newState) {
    System.out.println(newState);
  }

}
```

Iterator

- Podržan u samom jeziku:

  for (Stavka s : stavke) ...
- Kolekcije sadrže iteratore

  lista.iterator()

Primer

```
Automobil[] niz = new Automobil[] {
new Automobil("Audi", "A8"),
new Automobil("Renault", "Laguna"),
new Automobil("Fiat", "Punto") };

for (Automobil a : niz)
  System.out.println(a);
```

Primer

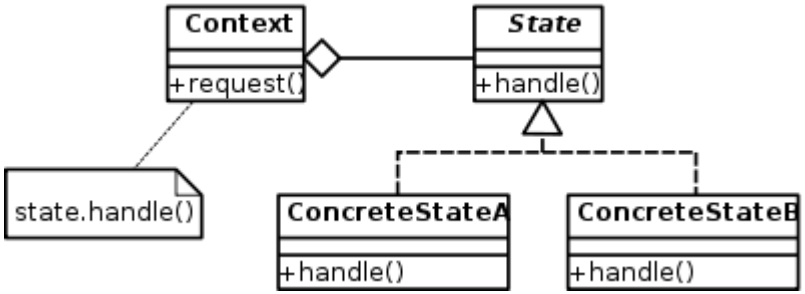
```
Collection<Automobil> kolekcija = new
ArrayList<Automobil>();
kolekcija.add(new Automobil("Audi", "A8"));
kolekcija.add(new Automobil("Renault", "Laguna"));
kolekcija.add(new Automobil("Fiat", "Punto"));
Iterator<Automobil> iterator = kolekcija.iterator();
while (iterator.hasNext())
  System.out.println(iterator.next());

kolekcija.add(new Automobil("VolksWagen", "Passat"));

for (Automobil a : kolekcija)
  System.out.println(a);
```

State

- Programska realizacija automata stanja
- Svako stanje je modelovano klasom
  - zajednički predak je ili klasa, ili interfejs
- Svako stanje "zna" koje je sledeće stanje, u zavisnosti od ulaza



Primer

```
public class Test {

  public static void main(String[] args) {
    Semafor semafor = new Semafor();
    semafor.ukljuci();
  }

}
```

Primer

```
public class Semafor {
  SemaforState stanje;
  public Semafor() {
    stanje = TrepcuceZuto.getInstance();
  }
  public void ukljuci() {
    sleep();
    stanje = stanje.ukljucio();
    sleep();
    for (int i = 0; i < 10; i++) {
      stanje = stanje.tajmerSeAktivirao();
      sleep();
    }
    stanje = stanje.iskljucio();
    sleep();
  }
```

Primer

```
public abstract class SemaforState {
  /** Kada se aktivira tajmer, vraća novo stanje. */
  public abstract SemaforState tajmerSeAktivirao();
  /** Ako se uključio u ovom stanju,
   *  vraća novo stanje.
   */
  public abstract SemaforState ukljucio();
  /** Ako se u bilo kom stanju iskljuci,
   *  prelazi u trepcuce zuto.
   */
  public SemaforState iskljucio() {
    return TrepcuceZuto.getInstance();
  }
  public abstract void ispisStanja();
}
```

Primer

```
public class TrepcuceZuto extends SemaforState {
...
  @Override
  public SemaforState tajmerSeAktivirao() {
    // sledece stanje je zeleno
    return Zeleno.getInstance();
  }
  @Override
  public SemaforState ukljucio() {
    return Zeleno.getInstance();
  }
  @Override
  public void ispisStanja() {
    System.out.println("TREPCUCE ZUTO");
  }
}
```

Primer

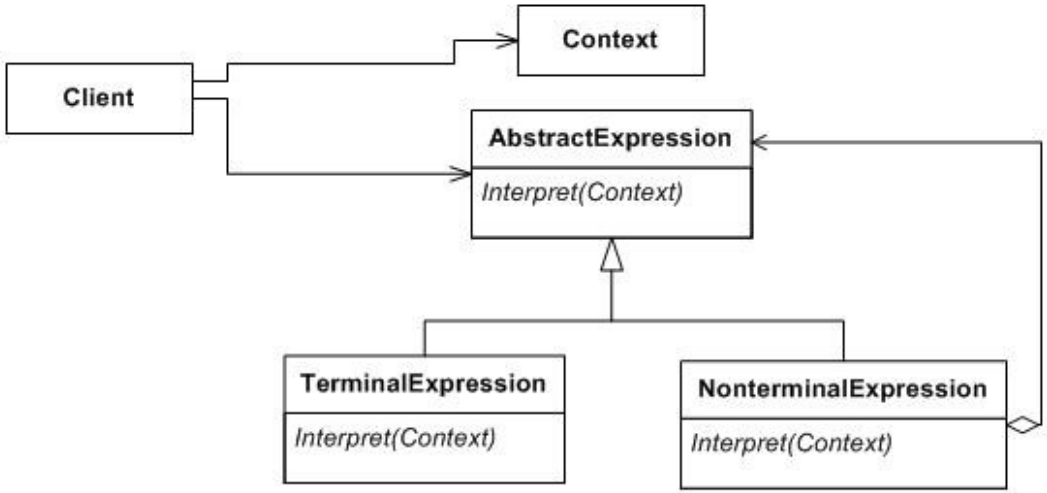
```
public class Zeleno extends SemaforState {
...
  @Override
  public SemaforState tajmerSeAktivirao() {
    // sledece stanje je trepćuće zeleno
    return TrepcuceZeleno.getInstance();
  }
  @Override
  public SemaforState ukljucio() {
    return this;
  }
  @Override
  public void ispisStanja() {
    System.out.println("ZELENO");
  }
}
```

Chain of Responsibility

- Koristi se kada je potrebno da jedan zahtev obradi više učesnika
- Svaki učesnik prosleđuje urađen posao do sledećeg učesnika
- Učesnici ne moraju da budu poznati unapred
- Primer:
  - digitalno potpisivanje i kriptovanje teksta

Interpreter

- Kao što mu ime kaže, interpretira neki niz izraza u skladu sa gramatikom
- Nije parser (prevodi tekst u neku memorijsku reprezentaciju), već izvršava izraze koji su već kreirani kao objekti odgovarajućih klasa (memorijska reprezentacija)
  - svaki objekat poseduje internu logiku izvršavanja



Mediator

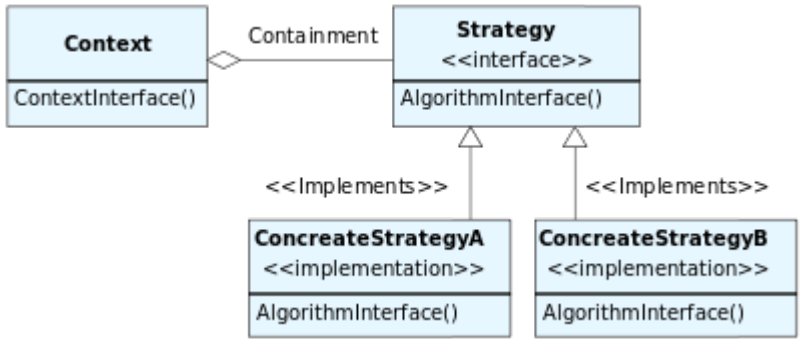
- Ako je potrebno da veći broj objekata međusobno komunicira, ponekad je jednostavnije/isplativije da svi komuniciraju kroz centralno čvorište – mediator.
- Primer:
  - dijalog za učitavanje datoteke: dugme Open je disable-ovano dok ne odaberemo datoteku
  - dijalog za odabir fonta: polje za prikaz teksta se podešava na osnovu odabranog fonta i njegovih svojstava

Memento

- Obezbeđuje mehanizam za pamćenje i restauraciju stanja aplikacije
  - osnova za undo/redo mehanizam
- Kada je potrebno da se uradi "snapshot" stanja aplikacije, kreira se memento
  - to se obično radi prilikom menjanja stanja
- Dve vrste implementacije:
  - pre izvršenja svake komande, zapamtiti trenutno stanje (ponekad nepraktično – obrada slike ili filma)
  - zapamtiti izvršenu komandu u nekakvoj kolekciji i implementirati inverznu funkciju ove komande

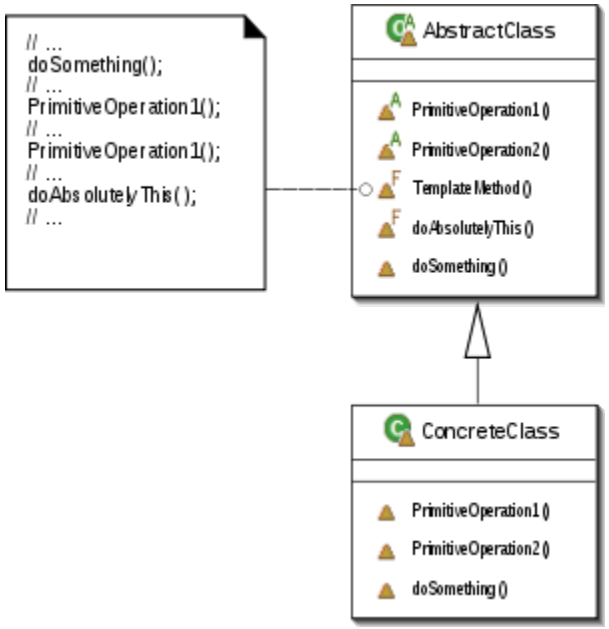
Strategy

- Više različitih algoritama je implementirano u više klasa
  - moguće je koristiti bilo koji od njih, u toku rada programa
- Primer:
  - snimanje u više različitih formata (Save As)
  - više različitih načina kriptovanja



Template method

- Definiše nacrt algoritma, gde se određeni delovi algoritma realizuju u klasama naslednicama, u redefinisanim metodama



Visitor

- Ako imamo operaciju koja bi trebalo da se izvede nad nekim objektom, Visitor omogućuje da se dodaju nove operacije nad takvim objektima
  - to se postiže tako što se napravi metoda accept, koja prima instancu Visitor klase