Nabolje prakse
u OO razvoju softvera

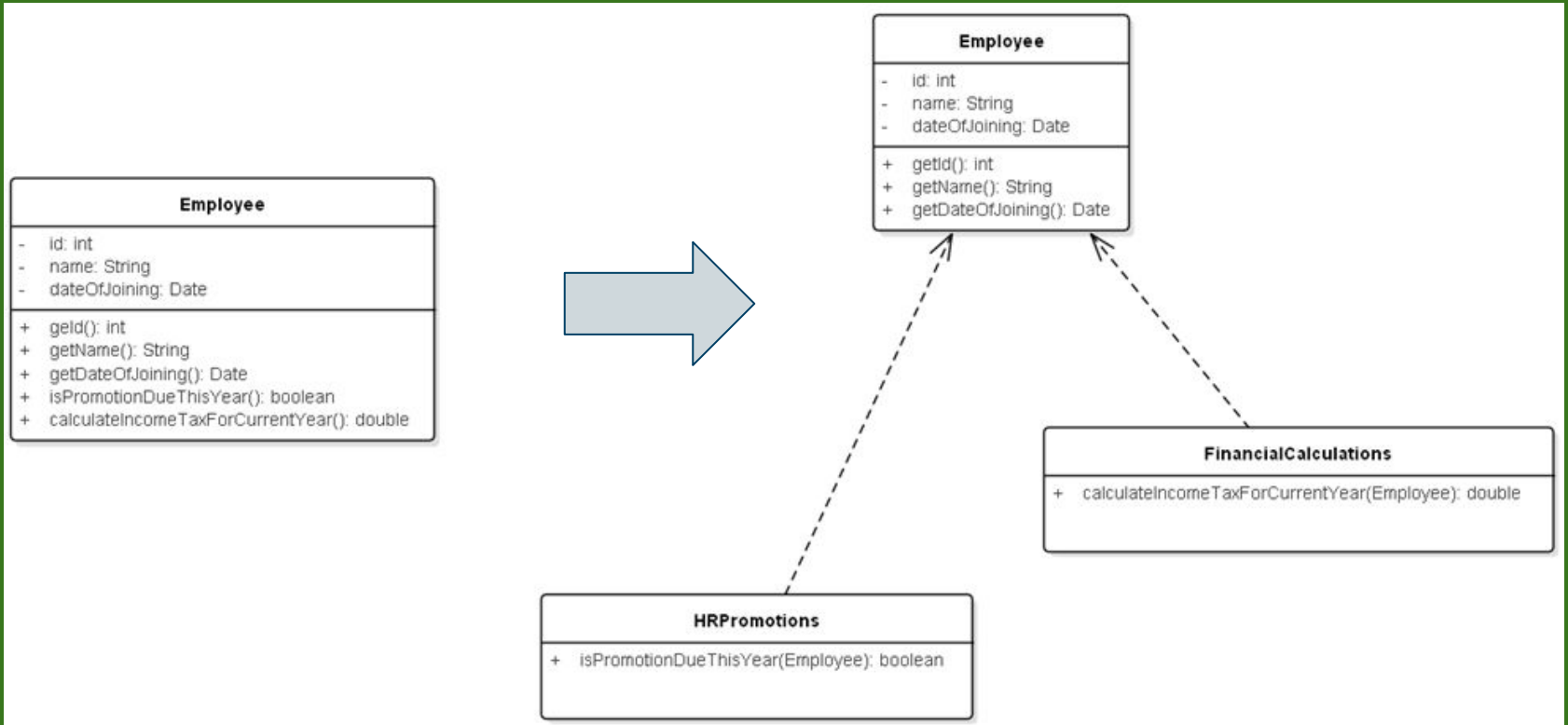
Sadržaj

- SOLID principi
- Kohezija i sprezanje
- Dizajnerski obrasci

SOLID principi

- *Single Responsibility:* “Klasa treba da ima samo jednu odgovornost.”
- *Open - closed:* “Softverske komponente treba da budu otvorene za proširivanje, ali zatvorene za modifikaciju.”
- *Liskov substitution:* “Metode pozvane za refence tipa osnovne klase moraju biti u stanju da koriste objekte izvedenih klasa.”
- *Interface segregation:* “Klijenti ne treba da budu zavisni od interfejsa koje ne koriste.”
- *Dependency inversion:* “Moduli visokog nivoa ne treba da zavise od modula niskog nivoa. I jedni i drugi treba da budu zavisni od apstrakcija.”

Single responsibility



Open - closed (loš pristup)

```
public class AreaCalculator {

  public double calculateCircleArea(Circle c) {
    return c.getRadius()*c.getRadius()*Math.PI;
  }

  public double calculateRectangleArea(Rectangle r) {
    return r.getWidth()*r.getHeight();
  }
}
```

Open - closed (dobar pristup)

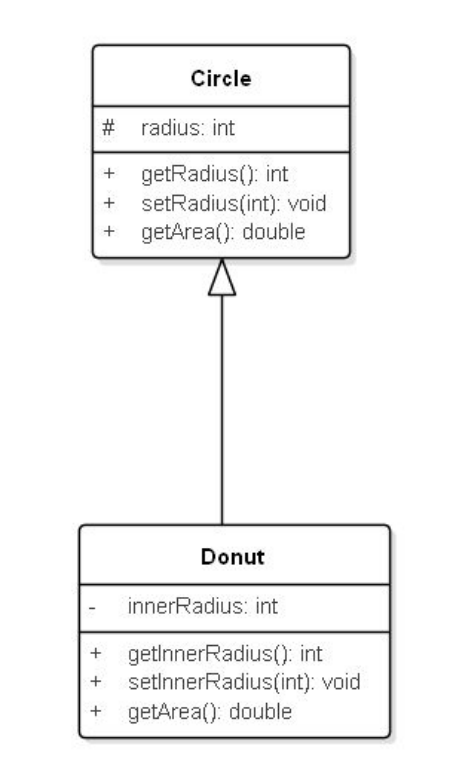
```
public abstract class Shape {
  public abstract double getArea();
}
```

```
public class Circle extends Shape
{
  ...
}
```

```
public class Rectangle extends Shape
{
  ...
}
```

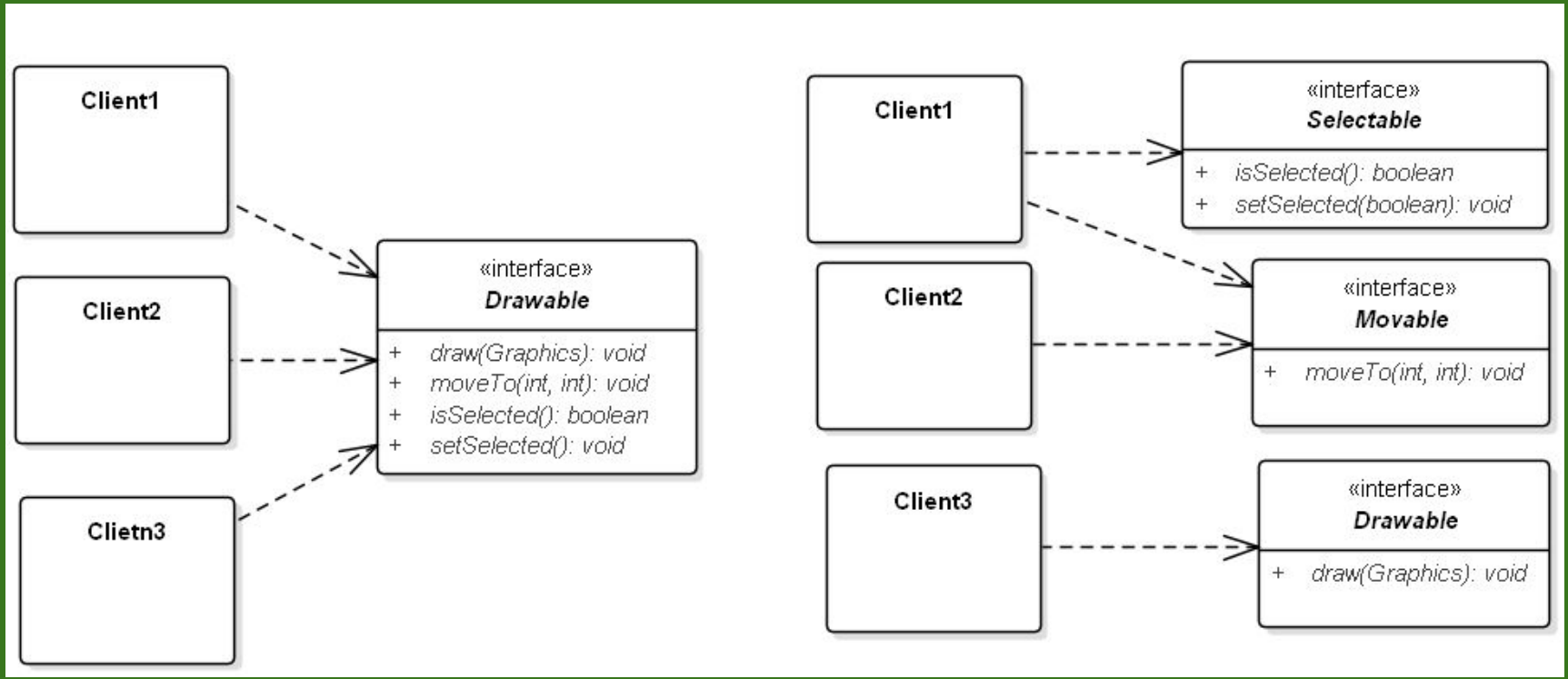
```
public class AreaCalculator {
  public double calculateArea(Shape s) {
    return s.getArea();
  }
}
```

Liskov substitution

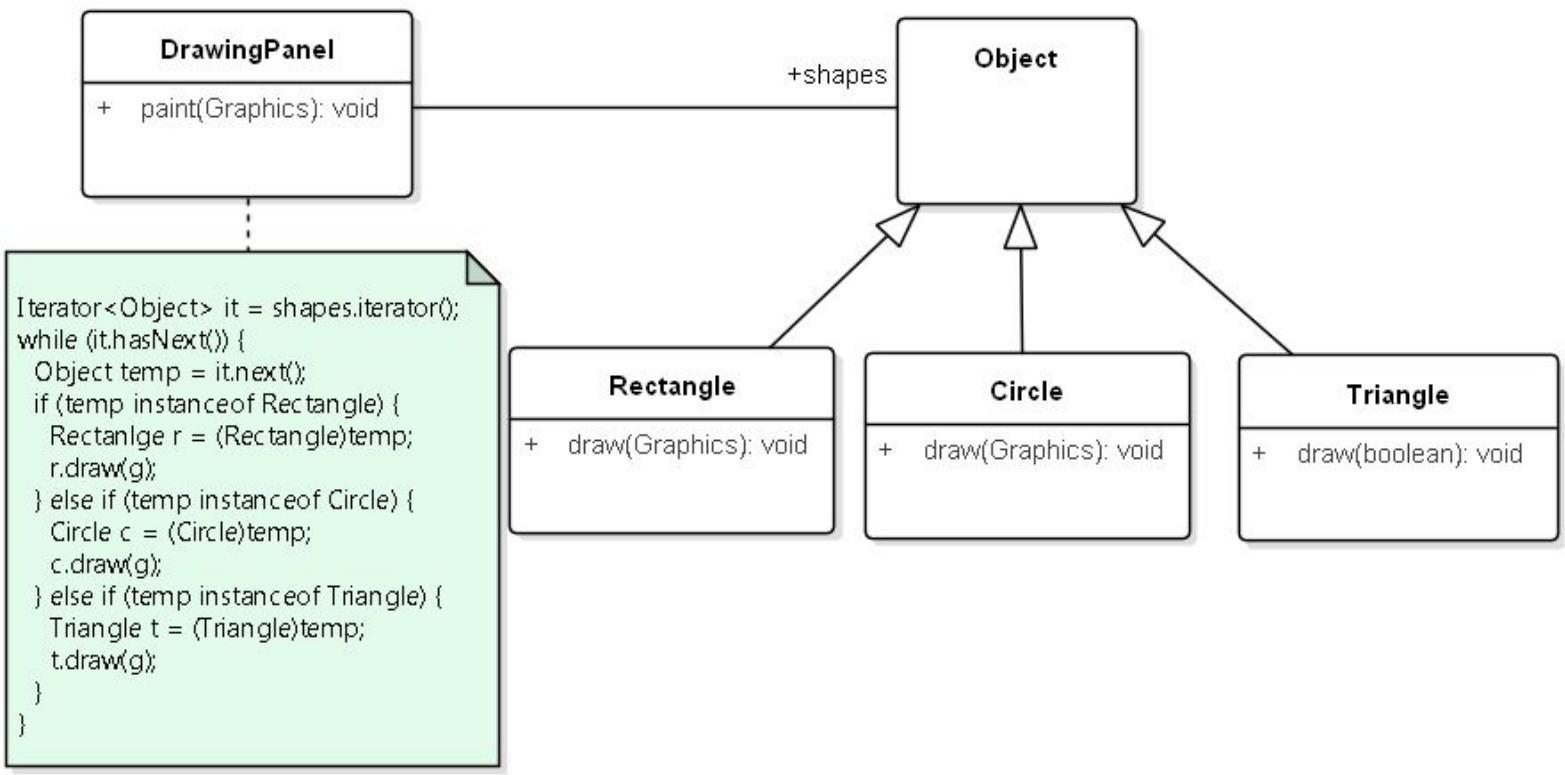



```
Circle c = new Circle(...);
double circleArea = c.getArea();
…
c = new Donut(...);
double donutArea = c.getArea();
```

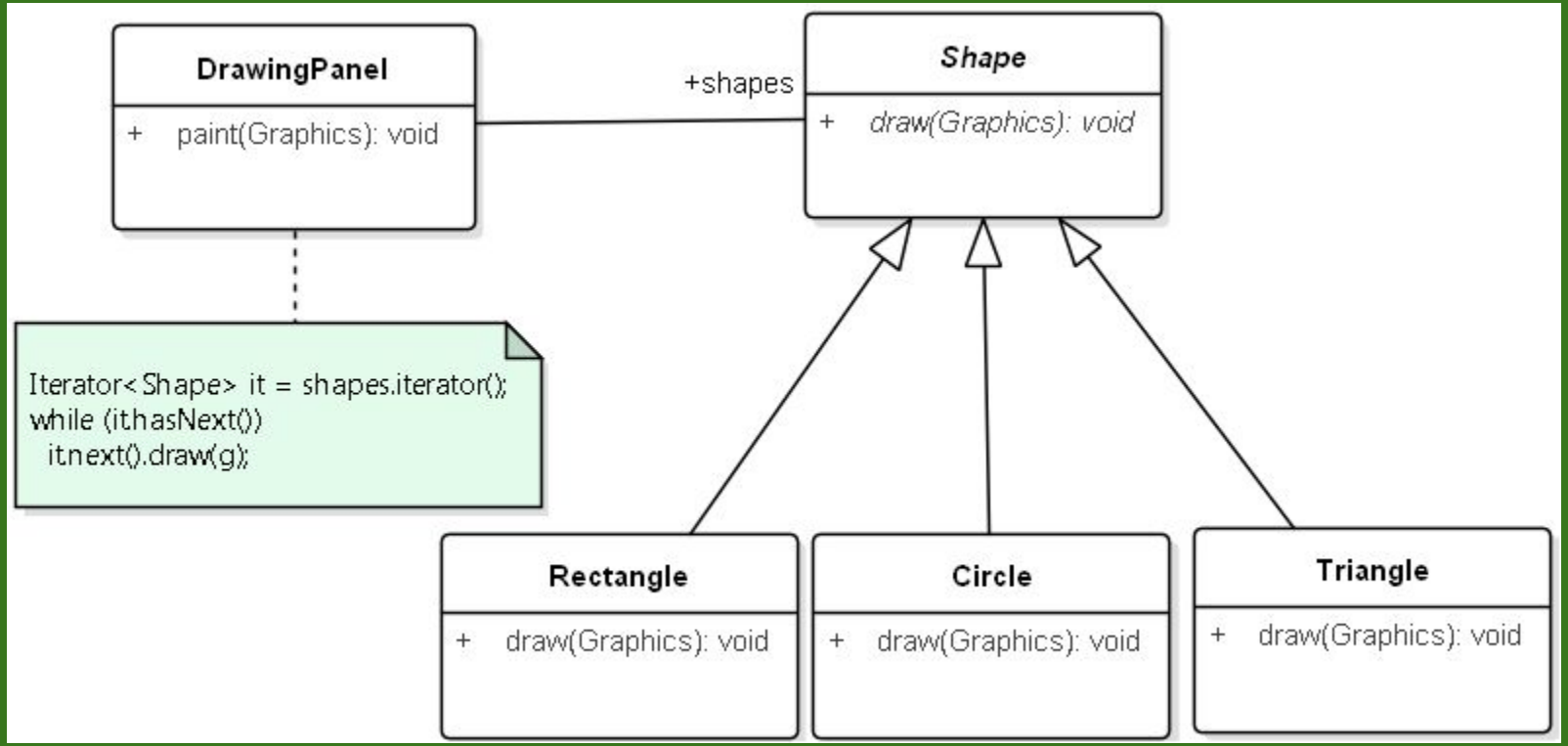
Interface segregation



Dependency inversion (loš pristup)



Dependency inversion (dobar pristup)



Kohezija i sprezanje

- Niska kohezija (engl. *low cohesion*)
  - Klasa ima više nepovezanih odgovornosti
- Visoka kohezija (engl. *high cohesion*)
  - Klasa ima jednu odgovrnost
- Čvrsto sprezanje (engl. *tight coupling*)
  - Obično posledica niske kohezije
  - Nefleksibilan sistem koji se teško prilagođava promenama zahteva
- Labavo sprezanje (engl. *loose coupling*)
  - Obično posledica visoke kohezije
  - Fleksibilan sistem koji se lako prilagođava promenama zahteva

Kompozicija nasuprot proširivanju

```
public class Donut extends Circle {
  private int innerRadius;
  …
  public double getArea() {
    return super.getArea() - innerRadius * innerRadius * Math.PI;
  }
}
```

```
public class Donut {
  private Circle outer;
  private Circle inner;
  …
  public double getArea() {
    return outer.getArea() - inner.getArea();
  }
}
```

I kompozicija i proširivanje

```
public class Donut extends Circle {
  private Circle hole;
  …
  public double getArea() {
    return super.getArea() - hole.getArea();
  }
}
```

Pojam dizajnerskog obrasca

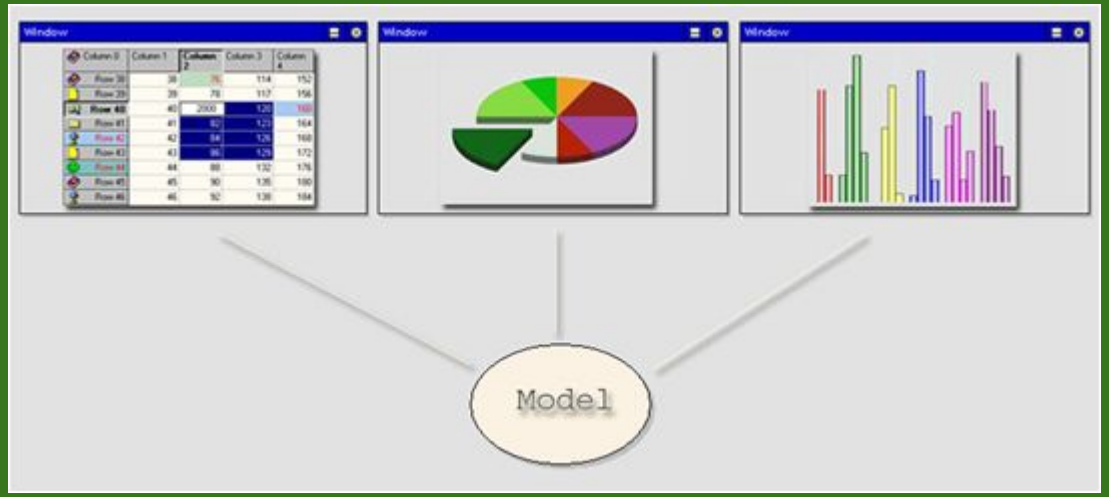
- “Each pattern describes a problem which occurs over and over again in out environment, and then describes core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same twice.”
  Cristopher Alexander (o obrascima u arhitekturi)

Elementi opisa obrasca

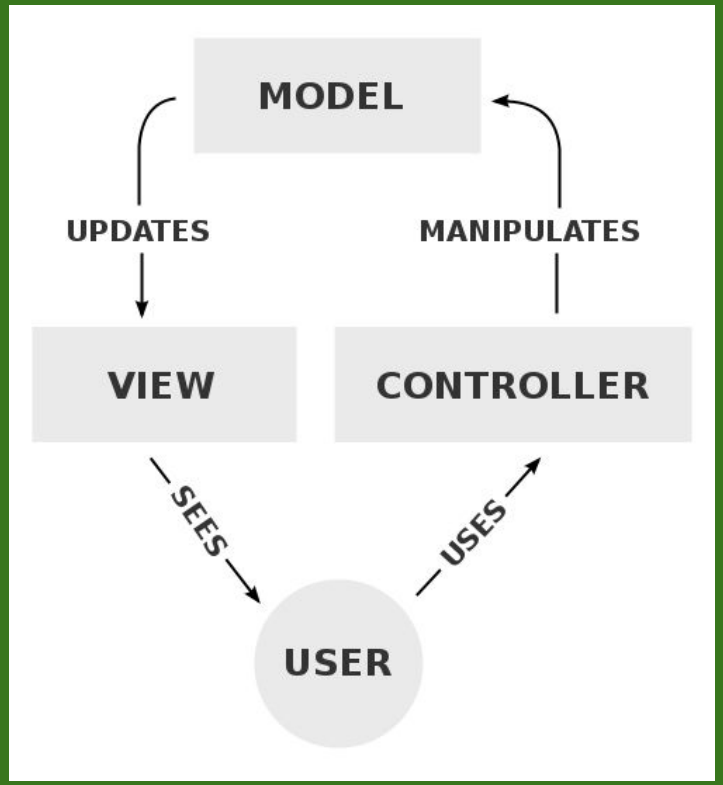
- Naziv obrasca - opis problema u reč ili dve
- Problem - opisuje problem kada se primenjuje obrazac
- Rešenje - opisuje elemente koji čine dizajn, njihove veze, odgovornosi i saradnju
- Posledice - rezultati, dobre i loše strane primene obrasca

Model View Controller (MVC)

- Model - podaci aplikacije
- View - prikaz podataka
- Controller - logika aplikacije

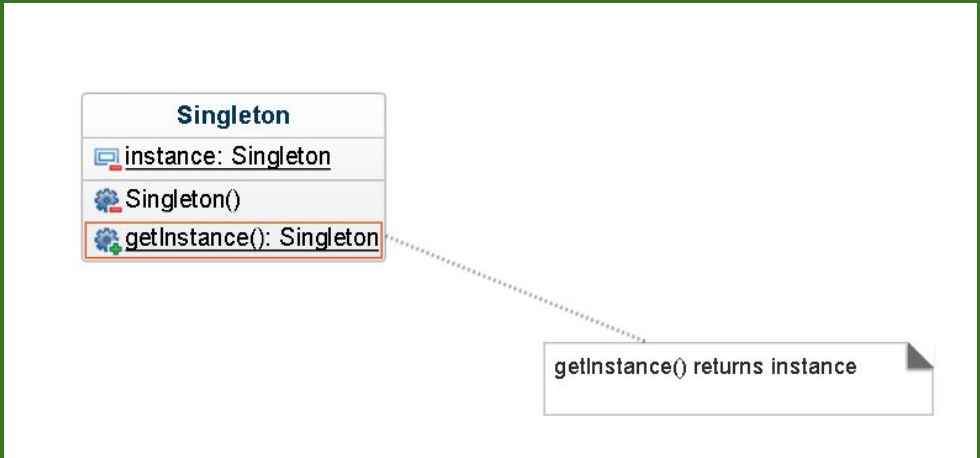


Interakcija MVC komponenti



Singleton

- Osigurava da klasa ima samo jednu instancu

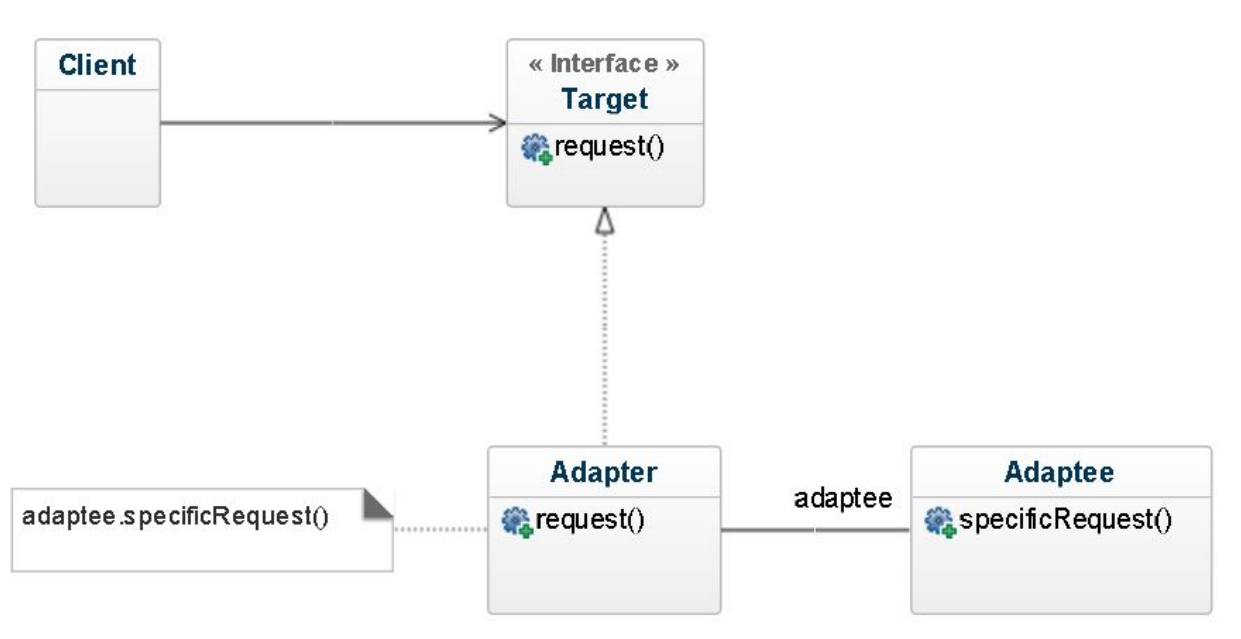


Singleton kod

```
public class MySingleton {

  private static MySingleton instance;

  private MySingleton() {
  }

  public static MySingleton getInstance() {
    if (instance == null) {
      instance = new MySingleton();
    }
    return instance;
  }
}
```

Adapter

- Konvertuje interfejs klase u drugi interfejs koji očekuje klijent



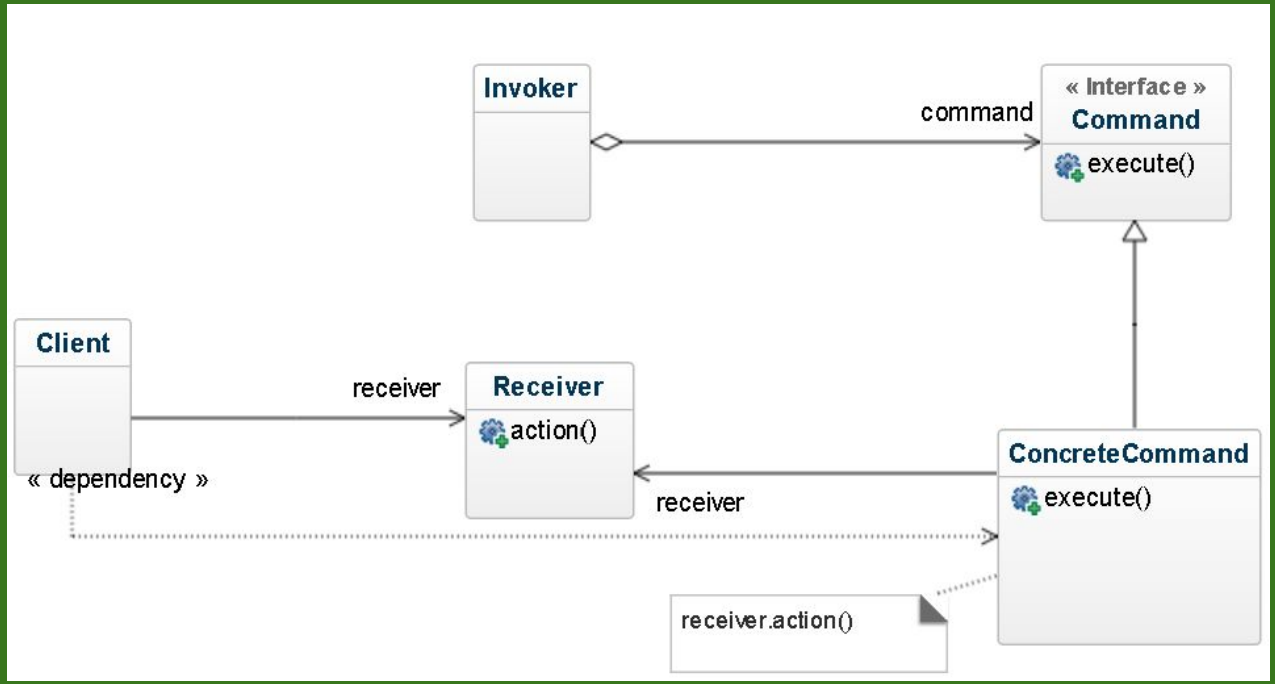
Adapter kod

```
public class Hexagon {
  …
  public void paint(Graphics g) {
    …
  }
  public boolean doesContain(int x, int y) {
    …
  }
}
```

```
public class HexagonAdapter extends Shape {
  private Hexagon hex;
  …
  public void draw(Graphics g) {
    this.hex.paint(g);
  }
  public boolean contains(int x, int y) {
    return this.hex.doesContain(x, y);
  }
}
```

Command

- Undo i redo funkcionalnost

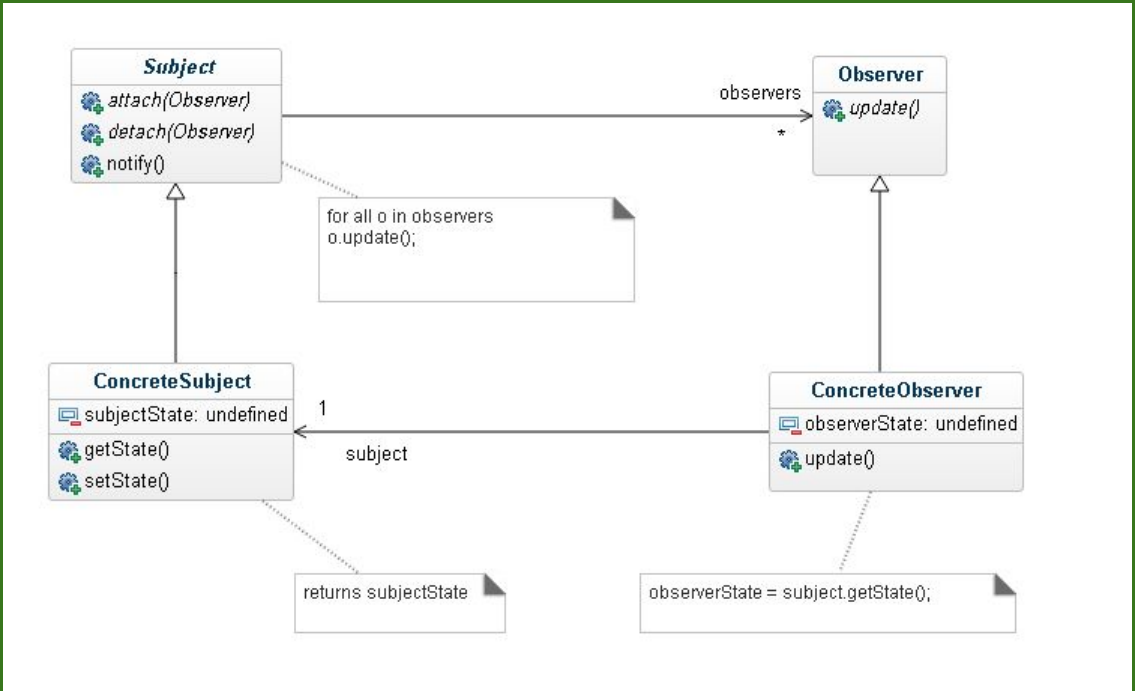


Command kod

```
public class CmdAddPoint implements Command {
  private Model model;
  private Point point;

  public CmdAddPoint(Model model, Point point) {
    this.model = model;
    this.point = point;
  }

  public void execute() {
    this.model.add(this.point);
  }

  public void unexecute() {
    this.model.remove(this.point);
  }
}
```

Observer

- Pretplaćivanje na događaje koji se dešavaju na posmatranom subjektu



Observer kod

```
public class Subject {
  private List<Observer> observers;
  private State state;
  …
  public void attach(Observer observer) {
    this.observers.add(observer);
  }
  …
  public void setState(State state) {
    this.state = state;
    notifyAllObservers();
  }

  public void notifyAllObservers() {
    for (Observer observer : this.observers) {
      observer.update(this.state);
    }
  }
}
```

Builder obrazac

- Koristi se kada se želi konstruisati kompleksan Product objekat
- Ukoliko Product ima veliki broj obeležja od koji su neka opcionalna, to zahteva definisanje većeg broja konstruktora ili prosleđivanje null vrednosti što može biti izvor grešaka
- Builder klasa obezbeđuje metode za postepeno konstruisanje Product objekta
- Postoji više varijacija ovog obrasca

Builder obrazac (1)

```
public class House {
  public static class Builder {
    private int id;
    private String foundations;
    private String walls;
    private String roof;

    public Builder(int id) {
      this.id = id;
    }

    public Builder foundations(String foundations) {
      this.foundations = foundations;
      return this;
    } …
```

Builder obrazac (2)

```
    public House build() {
      House house = new House();
      house.id = this.id;
      house.foundations = this.foundations;
      house.walls = this.walls;
      house.roof = this.roof;
      return house;
    }
  }

  // obelezja i metode pristupa za House

  private House() {
  }
}
```

Builder obrazac (3)

```
House solidHouse = new House.Builder(123)
        .foundations(“Concrete”)
        .walls(“Bricks”)
        .roof(“Tiles”)
        .build();

House ecoHouse = new House.Builder(124)
        .foundations(“Timber”)
        .walls(“Logs”)
        .roof(“Cane”)
        .build();
```

Rezime

- Poštovanje SOLID principa prilikom razvoja rezultuje kvalitetnijim softverom
- Visoka kohezija komponenti utiče labavo sprezanje komponenti što softver čini fleksibilnijim
- Dizajnerski obrasci nude rešenja problema koji se često javljaju u razvoju softvera