Föreläsning 9

Designmönstrer

Faktorisering

Identifiering av återkommande kodavsnitt – upptäcka möjliga återanvändbara komponenter:

- identifiera kodavsnitt som gör samma uppgift fast på olika platser
- implementera denna uppgift i en återanvändbar komponent
- omorganisera det ursprungliga programmet.

Minskar risken för inkonsistent hanterande

- ett eventuell fel behöver bara åtgärdas på en plats (annars behövs upprepade ändringar)

Dubblerad kod i samma klass

```
public class Computation {
   public void method1(...) {
      //...
      computeStep1();
      computeStep2();
      computeStep3();
      //...
   }//method1
   public void method2(...) {
      //...
      computeStep1();
      computeStep2();
      computeStep3();
      //...
   }//method2
}//Computation
```

Faktorisera via ny metod:

```
public class Computation {
   private void computeAll(...) {
      computeStep1();
      computeStep2();
      computeStep3();}
   }//computeAll
   public void method1(...) {
      //...
      computeAll();
      //...
   }//method1
   public void method2(...) {
      //...
      computeAll();
      //...
   }//method2
}//Computation
```

Dubblerad kod i olika klasser

```
public class ComputationA {
   public void method1(...) {
      //...
      computeStep1();
      computeStep2();
      computeStep3();
      //...
   }//method1
}//ComputationA
```

```
public class ComputationB {
   public void method2(...) {
      //...
      computeStep1();
      computeStep2();
      computeStep3();
      //...
   }//method2
}//ComputationB
```

Faktorisera via implementationsarv:

```
public class Common {
   protected void computeAll(...) {
      computeStep1();
      computeStep2();
      computeStep3();
   }//computeAll
}//Common

public class ComputationA extends Common {
   public void method1(...) {
      //...
      computeAll();
      //...
   }//method1
}//ComputationA

public class ComputationB extends Common {
   public void method2(...) {
      //...
      computeAll();
      //...
   }//method2
}//ComputationB
```

Dubblerad kod i olika klasser

```
public class ComputationA {
   public void method1(...) {
      //...
      computeStep1();
      computeStep2();
      computeStep3();
      //...
   }//method1
}//ComputationA
```

```
public class ComputationB {
   public void method2(...) {
      //...
      computeStep1();
      computeStep2();
      computeStep3();
      //...
   }//method2
}//ComputationB
```

Faktorisera via delegering:

```
public class Helper {
   protected void computeAll(...) {
      computeStep1();
      computeStep2();
      computeStep3();
   }//computeAll
}//Helper

public class ComputationA {
   Helper helper = . . . ;
   public void method1(...) {
      //...
      helper.computeAll();
      //...
   }//method1
}//ComputationA

public class ComputationB {
   Helper helper = . . . ;
   public void method2(...) {
      //...
      helper.computeAll();
      //...
   }//method2
}//ComputationB
```

Dubblerad kod mixad med omgivningsberoende kod

Antag att vi har en grafisk plotter för att rita ut en sinus-kurva:

```
import  java.awt.*;
import  javax.swing.*;
public class Plotter extends JPanel{
   private int xorigin;
   private int yorigin;
   private int xratio = 50;
   private int yratio = 50;

   public Plotter() {
     setBackground(Color.WHITE);
   }//constructor
```

```
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      drawCoordinates(g);
      plotFunction(g);
   }// paintComponent
```

```
   protected void drawCoordinates(Graphics g) {
      g.setColor(Color.BLACK);
      g.drawLine(0, getHeight()/2, getWidth(), getHeight()/2);
      g.drawLine(getWidth()/2,0, getWidth()/2, getHeight());
   }//drawCoordinates
```

```
   protected void plotFunction(Graphics g) {
      g.setColor(Color.RED);
      xorigin = getWidth()/2;
      yorigin = getHeight()/2;
      for (int px = 0; px < getWidth(); px++)  {
         double x = (px - xorigin) / (double) xratio;
         double y = Math.sin(x);
         int py = yorigin - (int) (y * yratio);
         g.fillOval(px - 1, py - 1, 3, 3);
      }
   }//plotFunction
}//Plotter
```

Dubblerad kod mixad med omgivningsberoende kod

För att rita ut en cosinus-kurva i stället för en sinus-kurva behövs endast en liten ändring i metoden `plotFunction`:

Metoden `plotFunction` för sinus-kurva:

```
protected void plotFunction(Graphics g) {
   g.setColor(Color.RED);
   xorigin = getWidth()/2;
   yorigin = getHeight()/2;
   for (int px = 0; px < getWidth(); px++)  {
      double x = (px - xorigin) / (double) xratio;
      double y = Math.sin(x);
      int py = yorigin - (int) (y * yratio);
      g.fillOval(px - 1, py - 1, 3, 3);
   }
}//plotFunction
```

Metoden `plotFunction` för cosinus-kurva:

```
protected void plotFunction(Graphics g) {
   g.setColor(Color.RED);
   xorigin = getWidth()/2;
   yorigin = getHeight()/2;
   for (int px = 0; px < getWidth(); px++)  {
      double x = (px - xorigin) / (double) xratio;
      double y = Math.cos(x);
      int py = yorigin - (int) (y * yratio);
      g.fillOval(px - 1, py - 1, 3, 3);
   }
}//plotFunction
```

Designmönstret Template Method

Problem: Hur definiera en algoritm där vissa delar skiljer sig åt i olika sammanhang.

Lösning: Beskriv algoritmen i en överordnad klass som en `templateMethod()`, som delar upp arbetet i olika deloperationer som vid behov kan omdefinieras i ärvda klasser.

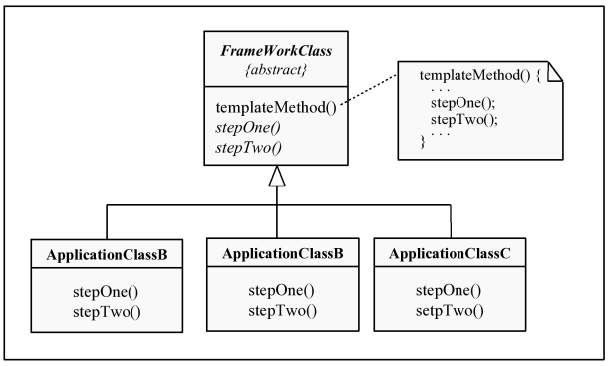
Syfte: Definiera ett skelett av en algoritm i en metod samt skjut upp vissa steg till subklasserna. Låt subklasserna definiera dessa steg.

Användbarhet: Används till att implementera invarianta delar i en algoritm, för att undvika dubblering av kod.

Enheter: En superklass som implementerar skal (*template*) för en eller flera metoder, vilka nyttjar abstrakta metoder, s.k. *hook*-metoder.

Konkreta klasser som implementerar *hook*-metoderna.

Designmönstret Template Method



Hollywood principen: *Don't call us, we'll call you!*

Refaktorering av klassen `Plotter`

Vi skriver om klassen `Plotter` enligt template-mönstret:

```
import  java.awt.*;
import  javax.swing.*;
public abstract class Plotter extends JPanel{
   private int xorigin;
   private int yorigin;
   private int xratio = 50;
   private int yratio = 50;
   public Plotter() {
     setBackground(Color.WHITE);
   }//constructor
   public void paintComponent(Graphics g) {
      //som tidigare
   }// paintComponent
   protected void drawCoordinates(Graphics g) {
      //som tidigare
   }//drawCoordinates
```

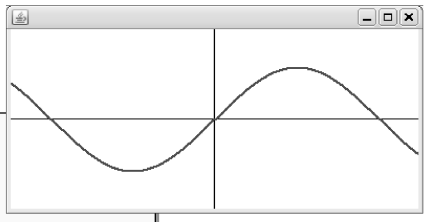
```
//template method
   protected void plotFunction(Graphics g) {
      g.setColor(Color.RED);
      xorigin = getWidth()/2;
      yorigin = getHeight()/2;
      for (int px = 0; px < getWidth(); px++)  {
         double x = (px - xorigin) / (double) xratio;
         double y = func(x);
         int py = yorigin - (int) (y * yratio);
         g.fillOval(px - 1, py - 1, 3, 3);
      }
   }//plotFunction
//hook-method
   public abstract double func(double x);
}//Plotter
```

Konkret plotter för sinus-funktionen

För att skapa en plotter för att rita ut en sinus-funktionen behöver vi

a) implementera en konkret klass där vi specificerar hook-metoden:

```
public class PlotSine extends Plotter {
   public double func(double x) {
      return Math.sin(x);
   }//func
}//PlotSine
```



b) samt en main-metod :

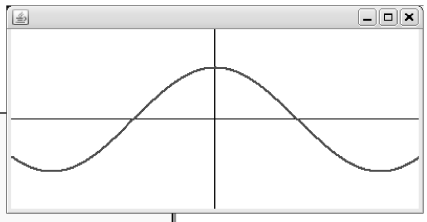
```
import javax.swing.*;
public class Main {
   public static void main(String[] args) {
      JFrame w = new JFrame();
      Plotter ps = new PlotSine();
      w.add(ps);
      w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      w.setSize(400,200);
      w.setVisible(true);
   }//main
}//Main
```

Konkret plotter för cosinus-funktionen

För att skapa en plotter för att rita ut en cosinus-funktionen behöver vi

a) implementera en konkret klass där vi specificerar hook-metoden:

```
public class PlotCosine extends Plotter {
   public double func(double x) {
      return Math.cos(x);
   }//func
}//PlotCosine
```



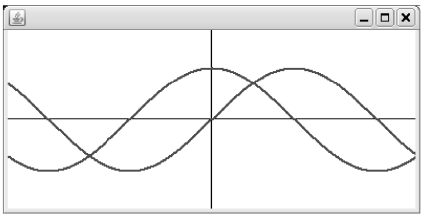
b) samt en main-metod :

```
import javax.swing.*;
public class Main {
   public static void main(String[] args) {
      JFrame w = new JFrame();
      Plotter ps = new PlotCosine();
      w.add(ps);
      w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      w.setSize(400,200);
      w.setVisible(true);
   }//main
}//Main
```

Hur rita flera funktioner med samma plotter?

Om vi vill rita ut två funktioner samtidig med samma plotter måste vi skapa en plotter med två hook-metoder:

```
public abstract class TwoPlotter extends JPanel{
   . . .
   // template-method
   protected void plotFunction(Graphics g) {
      g.setColor(Color.RED);
      xorigin = getWidth()/2;
      yorigin = getHeight()/2;
      for (int px = 0; px < getWidth(); px++)  {
         double x = (px - xorigin) / (double) xratio;
         double y = funcA(x);
         int py = yorigin - (int) (y * yratio);
         g.fillOval(px - 1, py - 1, 3, 3);
         y = funcB(x);
         py = yorigin - (int) (y * yratio);
         g.fillOval(px - 1, py - 1, 3, 3);
      }
   }//plotFunction
```



```
   // hook-methods
   public abstract double funcA(double x);
   public abstract double funcB(double x);
}//TwoPlotter
```

Hur rita flera funktioner med samma plotter?

De konkreta klasserna måste således implementera två hook-metoder:

```
public class PlotSinCos extends TwoPlotter {
   public double funcA(double x) {
      return Math.sin(x);
   } //funcA
   public double funcB(double x) {
      return Math.cos(x);
   }//funcB
}//PlotSinCos
```

För att rita tre funktioner med samma plotter måste vi skapa en template-klass:

```
public abstract class TripplePlotter {
   . . .
   // tre olika hook-metoder:
   public abstract double func1(double x);
   public abstract double func2(double x);
   public abstract double func3(double x);
}//TripplePlotter
```

- inte särskilt elegant
- ej flexibelt – endast ett fixt antal funktioner
- önskvärt att ha en `MultiPlotter` som ritar ut godtyckligt antal funktioner.

Designmönstret Strategy

Problem: Hur designa för att kunna välja mellan olika men relaterade algoritmer.

Lösning: Kapsla in de enskilda algoritmerna i en `Strategy`-klass.

Syfte: Göra algoritmerna utbytbara.

Används när:

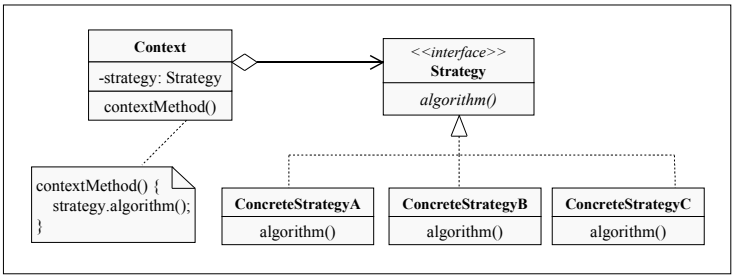
- flera relaterade klasser skiljer sig endast i beteende, inte i hur de används.
- algoritmerna som styr dessa beteenden använder information som klienten inte behöver känna till.
- olika varianter av en algoritm behövs.

Enheter: Ett interface (eller en abstrakt klass) `Strategy` som definierar strategin.

Konkreta klasser som implementerar olika strategier.

Klassen `Context` som handhar referenser till strategiobjekten.

Designmönstret Strategy



MultiPlotter med användning av Strategy-mönstret

Lösning:

Frigör funktionen som skall ritas ut från mekanismen som ritar ut funktionen.

I stället för att representera varje funktion som en metod representeras varje funktion som ett objekt.

```
public interface Function {
   double apply(double x);
}//Function

public class Sine implements Function {
   public double apply(double x) {
      return Math.sin(x);
   }//apply
}//Sine

public class Cosine implements Function {
   public double apply(double x) {
      return Math.cos(x);
   }//apply
}//Cosine
```

Nu kan vi utnyttja dynamisk bindning genom att alltid anropa `apply` i MultiPlotter:n på objekt av olika typ (t.ex. Sine eller Cosine).

abstract Multiplotter

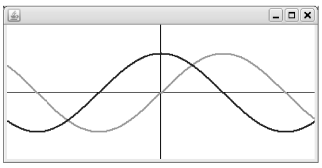
```
import java.awt.*;
import javax.swing.*;
public abstract class MultiPlotter extends JPanel{
   private int xorigin;
   private int yorigin;
   private int xratio;
   private int yratio;
   protected ArrayList<Function> functions = new ArrayList<Function>();
   protected ArrayList<Color> colors = new ArrayList<Color>();

   abstract public void initMultiPlotter();

   public MultiPlotter() {
      initMultiPlotter();
      setBackground(Color.WHITE);
   }//initMultiPlotter

   public void addFunction(Function f, Color c) {
      if (f != null) {
         functions.add(f);
         colors.add(c);
      }
   }//addFunction

   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      drawCoordinates(g);
      plotFunctions(g);
   }//paintComponent
```

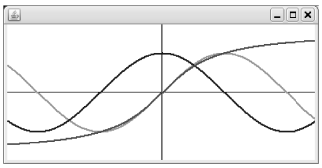
abstract Multiplotter

```
    protected void drawCoordinates(Graphics g) {
        g.setColor(Color.BLACK);
        g.drawLine(0, getHeight()/2, getWidth(), getHeight()/2);
        g.drawLine(getWidth()/2,0, getWidth()/2, getHeight());
    }//drawCoordinates
    protected void plotFunctions(Graphics g) {
        xorigin = getWidth()/2;
        yorigin = getHeight()/2;
        for (int i = 0; i < functions.size(); i++) {
            g.setColor(colors.get(i));
            for (int px = 0; px < getWidth(); px++)  {
                double x = (px - xorigin) / (double) xratio;
                double y = functions.get(i).apply(x);
                int py = yorigin - (int) (y * yratio);
                g.fillOval(px - 1, py - 1, 3, 3);
            }
        }
    }//plotFunctions
}//MultiPlotter
```

Två konkreta MultiPlotters

```
import java.awt.Color;
public class DoublePlotter extends MultiPlotter {
    public void initMultiPlotter() {
        addFunction(new Sine(), Color.GREEN);
        addFunction(new Cosine(), Color.BLUE);
    }//initMultiPlotter
}//DoublePlotter
```

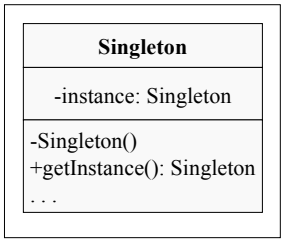




```
import java.awt.Color;
public class TrippelPlotter extends MultiPlotter {
    public void initMultiPlotter() {
        addFunction(new Sine(), Color.GREEN);
        addFunction(new Cosine(), Color.BLUE);
        addFunction(new Atan(), Color.RED);
    }//initMultiPlotter
}//TripplePlotter
```

Designmönstret Singleton

En singleton-klass är en klass som det får finnas *endast en instans* av och denna instans är *globalt åtkomlig*.



Den globala instansen av klassen hålls som en privat klassvariabel i klassen själv (variabeln `instance` i figuren ovan).

Konstruktorn i klassen är privat!

Användare erhåller instansen av klassen via metoden `getInstance()`.

En singleton klass är i övrigt som en vanlig klass och kan självklart ha fler variabler och metoder.

Designmönstret Singleton

```
public class Singleton {
   private static final Singleton instance = new Singleton();
   // Other useful instance variables here
   private Singleton() {}
   public static Singleton getInstance() {
      return instance;
   }//getInstance()
   // Other useful methods here
}//Singleton
```

Statiska attribut i en klass initieras när Javas Viritual Machine laddar in klassen. Eftersom en klassen måste finnas inladdad innan någon av dess metoder anropas kommer alltså attributet `instance` att initieras innan metoden `getInstance` anropas första gången.

Designmönstret Singleton

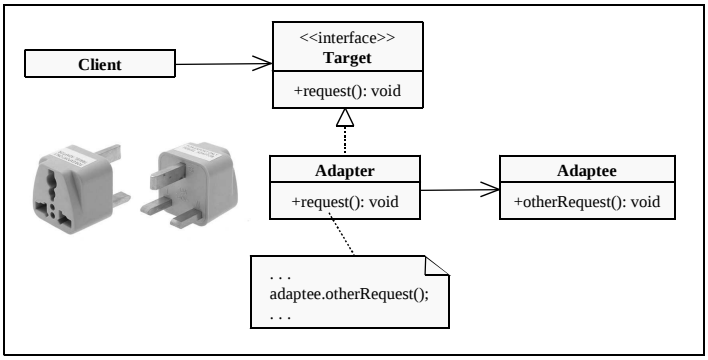
Instansieringen av kan skjutas upp tills metoden `getInstance` anropas för första gången. Detta innebär att denna metod måste göras trådsäker (konstruktionen **synchronized**).

```
public class Singleton {
   private static Singleton instance;
   // Other useful instance variables here
   private Singleton() {}
   public static synchronized Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton();
      }
      return instance;
   }//getInstance()
   // Other useful methods here
}//Singleton
```

Vi återkommer till trådsäkerhet i en senare föreläsning

Designmönstret Adapter

Designmönstret Adapter Pattern konverterar gränsnittet hos en klass till ett annat gränssnitt så att inkomplatibla klasser kan samarbeta.



`Target` beskriver gränssnittet såsom `Client` vill ha det. I klassen `Adapter` sker konverteringen av anropen.

Designmönstret Adapter

```
public interface SquarePeg {
    public void insert(String str);
}//SquarePeg
```

```
public class ConcreteSquarePeg implements SquarePeg{
    public void insert(String str) {
      System.out.println("SquarePeg insert: " + str);
    }//insert
}//SquarePeg
```

```
public class RoundPeg {
    public void insertIntoHole(String msg) {
        System.out.println("RoundPeg insertIntoHole: " + msg);
    }//InsertIntoHole
}//RoundPeg
```

Designmönstret Adapter

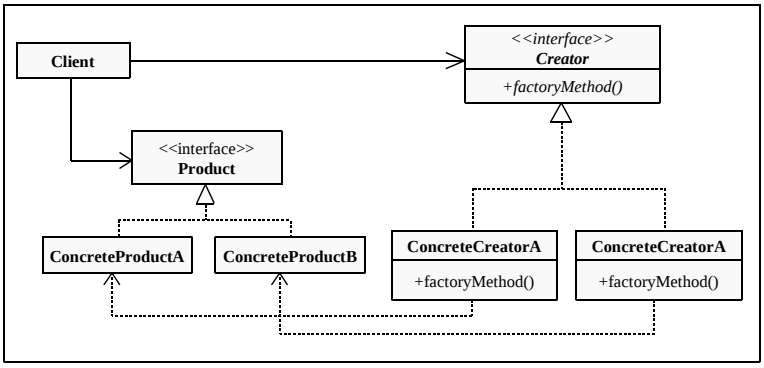
```
public class PegAdapter implements SquarePeg {
    private RoundPeg roundPeg;
    public PegAdapter(RoundPeg peg) {
        this.roundPeg = peg;
    }//constructor
    public void insert(String str) {
        roundPeg.insertIntoHole(str);
    }//insert
}//PegAdapter
```

```
public class TestPegs {
    public static void main(String[] args) {
        RoundPeg roundPeg = new RoundPeg();
        SquarePeg squarePeg = new ConcreteSquarePeg();
        SquarePeg adapter = new PegAdapter(roundPeg);
        squarePeg.insert("Inserting square peg!");
        adapter.insert("Inserting round peg!");
    }//main
}//TestPegs
```

Designmönstret Factory Method

Factory method används när man

- vid exekveringen måste avgöra vilken typ av objekt som skall skapas
- vill undvika beroenden av konkreta klasser.



Designmönstret Factory Method

Produkter

```
public interface Product {
    String shipFrom();
}
```

```
public class ProductA implements Product {
    public String shipFrom () {
        return " from South Africa";
    }
}
```

```
public class ProductB implements Product {
    public String shipFrom () {
        Return " from Spain";
    }
}
```

```
public class DefaultProduct implements Product {
    public String shipFrom () {
        return " not available";
    }
}
```

Designmönstret Factory Method

Fabrik

```
public class Creator {
   public Product factoryMethod(int month) {
      if (month >= 4 && month <=11)
         return new ProductA();
      else  if (month == 1 || month == 2 || month == 12)
         return new ProductB();
      else
     return new DefaultProduct();
   }
}
```

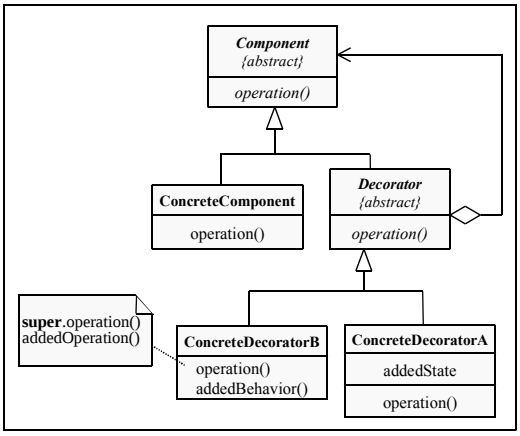
Klient

```
public class Main {
   public static void main(String[] args) {
      Creator c = new Creator();
      Product product;
       for (int i=1; i<=12; i++) {
          product = c.factoryMethod(i);
          System.out.println("Avocados " + product.shipFrom());
      }
   }
}
```

Körningsexempel:
Avocados from Spain
Avocados from Spain
Avocados not available
Avocados from South Africa
Avocados from South Africa
Avocados from South Africa
Avocados from South Africa
Avocados from South Africa
Avocados from South Africa
Avocados from South Africa
Avocados from South Africa
Avocados from Spain

Designmönstret Decorator

Designmönstret Decorator används för att lägga till funktionalitet på individuella objekt utan att använda arv. Mönstret används ofta då arv skulle ge alldeles för många nivåer i arvshierarkin, eller då arv inte är möjligt.
Rekommenderad struktur för Decorator-mönstret är enligt:



Component: En klass som definierar gränssnittet för de objekt som dynamiskt kan ges ytterligare funktionalitet.

ConcreteComponent: En klass som definierar objekt till vilka ytterligare funktionalitet kan ges.

Decorator: Har en referens till ett Component-objekt och definierar ett gränssnitt som överensstämmer med gränssnittet för Component.

ConcreteDecorator: Lägger till funktionalitet på objektet.

Exempel på användning av Decorator

Antag att vi har en smörgåsbutik. Kunden kan kombinera sin smörgås efter eget önskemål. Det finns ett antal brödsorter att välja mellan och ett antal olika pålägg. Vi startar med att skapa en abstrakt klass `Sandwich` enligt:

```
public abstract class Sandwich {
    protected String name;
    public String getName() {
        return name;
    }//getName

 public abstract double getPrice();

}//Sandwich
```

Exempel på användning av Decorator

Sedan skapar vi en klass för var och en av de brödsorter vi har. Dessa klasser ärver klassen `Sandwich`. Säg att vi har panini och baguetter:

```
public class Panini extends Sandwich {
    public Panini(){
        name = "Panini";
    }//constructor
    public double  getPrice() {
        return 10.5;
    }//getPrice
}//Panini
```

```
public class  Baguette extends Sandwich {
    public Baguette() {
        name = "Baguette";
    }//constructor
    public double getPrice() {
        return 8.5;
    }//getPrice
}//Baguette
```

Exempel på användning av Decorator

Sedan inför vi vår Decorator-klass, `SandwichDecorator`, vilken är en abstrakt klass som ärver från `Sandwich`-klassen och som har en instansvariabel av klassen `Sandwich`:

```
public abstract class SandwichDecorator extends Sandwich {
    protected Sandwich decoratedSandwich;
   public abstract double getPrice();
}//SandwichDecorator
```

Nu kan vi skapa konkreta klasser, t.ex. för att lägga ost, skinka, salami eller tomater på smörgåsen.

Exempel på användning av Decorator

Nu kan vi skapa konkreta klasser, t.ex. för att lägga ost, skinka, salami eller tomater på smörgåsen:

```
public class Cheese extends SandwichDecorator {
    public Cheese (Sandwich sandwich) {
        decoratedSandwich = sandwich;
        name = decoratedSandwich.name + " + Cheese";
    }//constructor
    public double getPrice() {
        return decoratedSandwich.getPrice() + 5.5;
    }//getPrice
}//Cheese
```

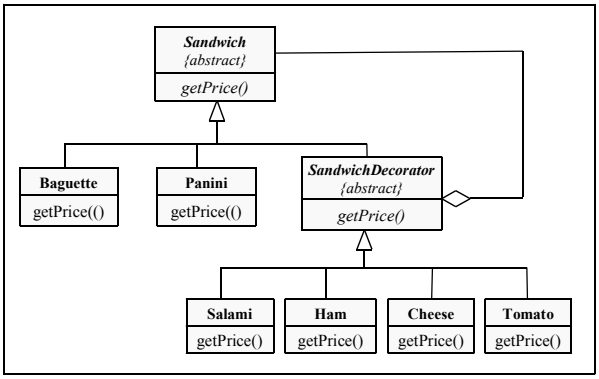
```
public class Ham extends SandwichDecorator {
    public Ham (Sandwich sandwich) {
        decoratedSandwich = sandwich;
        name = decoratedSandwich.name + " + Ham";
    }//constructor
    public double getPrice() {
        return decoratedSandwich.getPrice() + 7.5;
    }//getPrice
}//Ham
```

Exempel på användning av Decorator

```
public class Salami extends SandwichDecorator {
   public Salami (Sandwich sandwich) {
      decoratedSandwich = sandwich;
      name = decoratedSandwich.name + " + Salami";
   }//constructor
   public double getPrice(){
      return  decoratedSandwich.getPrice() + 8.5;
   }//getPrice
}//Salami
```

```
public class Tomato extends SandwichDecorator {
   public Tomato (Sandwich sandwich) {
      decoratedSandwich = sandwich;
      name = decoratedSandwich.name + " Tomato";
   }//constructor
   public double getPrice(){
      return decoratedSandwich.getPrice() + 2.5;
   }//getPrice
}//Tomato
```

Exempel på användning av Decorator



Exempel på användning av Decorator

```
public class TestSandwich {
   public static void main (String[] args){
      //lets create a baguette with cheese, ham and tomato
      Sandwich ourBaguette= new Baguette();
      ourBaguette = new Cheese(ourBaguette);
      ourBaguette = new Ham(ourBaguette);
      ourBaguette = new Tomato(ourBaguette);
      System.out.println(ourBaguette.getName() + " = " +
                              ourBaguette.getPrice() + " kronor");
      //lets create a panini with salami and tomato.
      Sandwich ourPanini = new Panini();
      ourPanini = new Salami(ourPanini);
      ourPanini = new Tomato(ourPanini);
      System.out.println(ourPanini.getName() + " = " +
                              ourPanini.getPrice() + " kronor");
   }//main
}//TestSandwich
```

När programmet körs erhålls utskriften:

Baguette + Cheese + Ham + Tomato = 24.0 kronor
Panini + Salami + Tomato = 21.5 kronor

Designmönstret Observer

Designmönstret Observer är en teknik som tillåter ett objekt som ändrat sitt tillstånd att rapportera detta till andra berörda objekt så att de kan vidta lämpliga åtgärder.

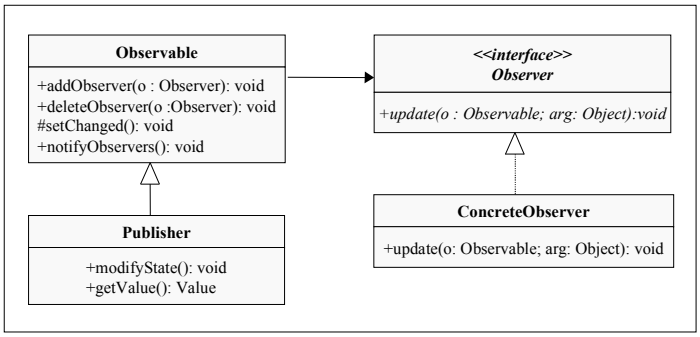
Designmönstret Observer

- definierar en-till-många relation
- minskar kopplingen mellan objekt

Java stöder designmönstret Observer genom:

- klassen `Observable`
- interfacet `Observer`

Designmönstret Observer



Det objekt som tillkännager att det ändrat sitt tillstånd är alltså observerbart (*observable*) och kallas ofta *publisher*. De objekt som underrättas om tillståndsförändringen kallas vanligtvis *observers* eller *subscribers*.

Designmönstret Observer

Klassen `Observable` har bl.a följande metoder:

void addObserver(Observer o) lägger till en ny observatör `o`.

void deleteObserver(Observer o) tar bort observatören `o`.

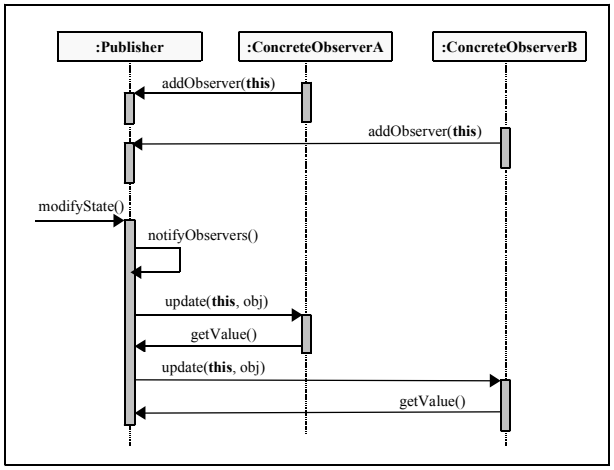
void notifyObservers() meddelar alla observatörer att tillståndet har förändrats.

protected void setChanged() markerar att tillståndet har ändrats.

Interfacet `Observer` har metoden

void update(Observable o, Object arg) som anropas av det observerbara objektet `o` när detta har förändrats.

Designmönstret Observer



Exempel – Datorintrång

```
import java.util.Observable;
public class IntrusionDetector extends Observable {
   public void someOneTriesToBreakIn() {
      //Denna metod anropas när någon försöker hacka sig in
      // i datorn. Detta meddelas alla som är intresserade
      setChanged();
      notifyObservers();
   }//someOneTriesToBreakIn
}//Observable
```

```
import java.util.Observer;
import java.util.Observable;
public class SysAdm implements Observer{
   private String name;
   private IntrusionDetector dectector;
   public SysAdm( String name, IntrusionDetector dectector) {
      this.name = name;
      this.dectector = dectector;
   }//constructor
   public void subscribe( ) {
      dectector.addObserver(this);
   }//subscriber
   public void update(Observable server, Object err ) {
      System.out.println( name + "  got the message!" );
   }//update
}//SysAdm
```

Exempel – Datorintrång

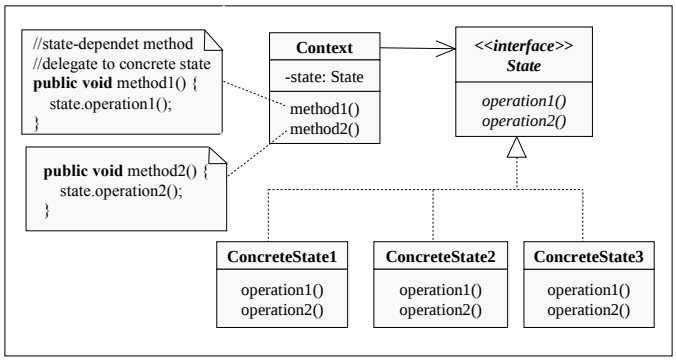
```
public class Main {
   public static void main( String[] argv ) {
      IntrusionDetector id = new IntrusionDetector();
      SysAdm kalle = new SysAdm( "Kalle", id );
      SysAdm sven = new SysAdm( "Sven", id );
      SysAdm rune = new SysAdm( "Rune", id );
      kalle.subscribe();
      rune.subscribe();
      // Nu låtsas vi att någon försöker ta sig in
      id.someOneTriesToBreakIn();
   }//main
}//Main
```

Rune got the message!
Kalle got the message!

Designmönstret State

Objekt har tillstånd och beteenden. Objekt ändrar sina tillstånd beroende på interna och externa händelser.

Om ett objekt går igenom ett antal klart identifierbara tillstånd och objektets beteende påverkas signifikant av sitt tillstånd kan det vara lämpligt att använda designmönstret `State`.



Designmönstret State

`State`-mönstret kapslar in de individuella tillstånden tillsammans med den logik som används för att byta tillstånd.

Istället för att direkt implementera beteendet för ett objekt i metoderna i klassen, så delegerar objektet vidare vad som skall göras till ett annat objekt (som är av en annan klass).

Detta innebär att man dynamiskt kan konfigurera om och byta ut beteendet hos objekt, det är som att objektet "byter klass". Vanligt arv är ju en statisk relation som inte kan förändras under runtime.

Samma strukturell uppbyggnad som designmönstret Strategy, men annat syfte.

Exempel på State-mönstret

Antag att vi har en amfibiebil.

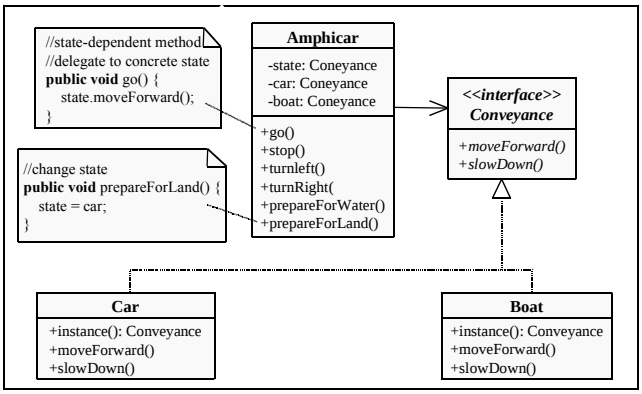
När amfibiebilen körs på land driver motorn bakhjulen och när den körs i vatten driver motorn två propellrar. Styrningen sker både på land och i vatten med framhjulen.

Låt oss säga vi vill ha följande operationer på amfibiebilen:

prepareForWater()	byt tillstånd för att köras i vatten
prepareForLand()	byt tillstånd för att köras på land
turnRight()	sväng höger, samma både på land och i vatten
turnLeft()	sväng vänster, samma både på land och i vatten
go()	kör, är olika beroende av om amfibiebilen är på land eller i vatten
stop()	stanna, är olika beroende av om amfibiebilen är på land eller i vatten

Design

```
public interface Conveyance {
   abstract public void moveForward();
   abstract public void slowDown();
}//Conveyance
```



Exempel: Klasserna Car och Boat

```
public class Car implements Conveyance {
   private Car(){};
   public void moveForward() {
      System.out.println(
            "Forward power to rear wheels");
   }//moveForward
   public void slowDown() {
      System.out.println(
            "Apply breaks on each  wheel");
   }//slowDown
}//Car
```

```
public class Boat implements Conveyance {
   private Boat(){};
   public void moveForward() {
      System.out.println(
            "Forward power to propeller");
   }//moveForward
   public void slowDown() {
      System.out.println(
            "Reverse power to propeller");
   }//slowDown
}//Boat
```

Exempel: Klassen Amphicar

```
public class Amphicar {
   private Conveyance state;
   private Conveyance car;
   private Conveyance boat;
   public Amphicar() {
      car = new Car();
      boat = new Boat();
      state = car;
   }//constructor
   public void go() {
      state.moveForward();
   }//go
   public void stop() {
      state.slowDown();
   }//stop
   // Turning doesn't depend on the state
   //so it is implemented here.
   public void turnRight() {
      System.out.println("Turn wheels right.");
   }//turnRight
   public void turnLeft() {
      System.out.println("Turn wheels left.");
   }//turnLeft
   public void prepareForWater() {
      state = boat;
   }//prepareForWater
   public void prepareForLand() {
      state = car;
   }//prepareForLand
}//Amphicar
```

Exempel: Testprogram

```
public class Main {
   public static void main(String[] args) {
      Amphicar a = new Amphicar();
      a.go();
      a.turnRight();
      a.stop();
      a.prepareForWater();
      a.go();
      a.turnRight();
      a.stop();
   }//main
}//Main
```

Testkörning:

```
Forward power to rear wheels
Turn wheels right.
Apply breaks on each wheel
Forward power to propeller
Turn wheels right.
Reverse power to propeller
```

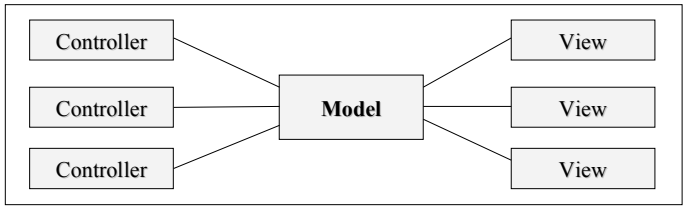
Designmönstret Model – View - Control

Designmönstret Model-View-Control (MVC) frikopplar användargränssnittet från den underliggande modellen. MVC delar upp en applikation i tre olika typer av klasser:

Model: Den datamodell som används – en abstrakt representation av den information som bearbetas i programmet. Utgörs av de domänklasser som ansvarar för tillståndet i applikationen.

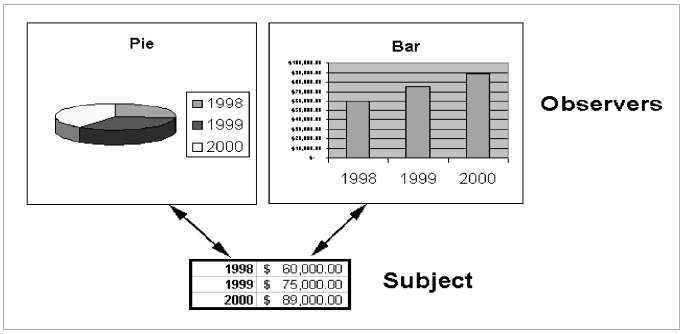
View: De klasser som ansvarar för att presentera modellen.

Control: De klasser som påverkar modellens tillstånd.



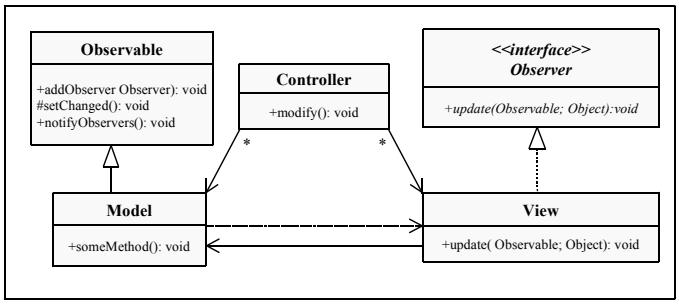
Model-View-Control

Genom att separera modellen och vyn kan man ändra i vyn och skapa flera vyer utan att behöva ändra i koden i den underliggande modellen.



När tillståndet i modellen förändras skall vyerna uppdateras, vi har alltså designmönstret `Observer`.

Model-View-Control



Flera andra varianter på MVC-mönstret är möjliga och finns beskrivna i litteraturen.

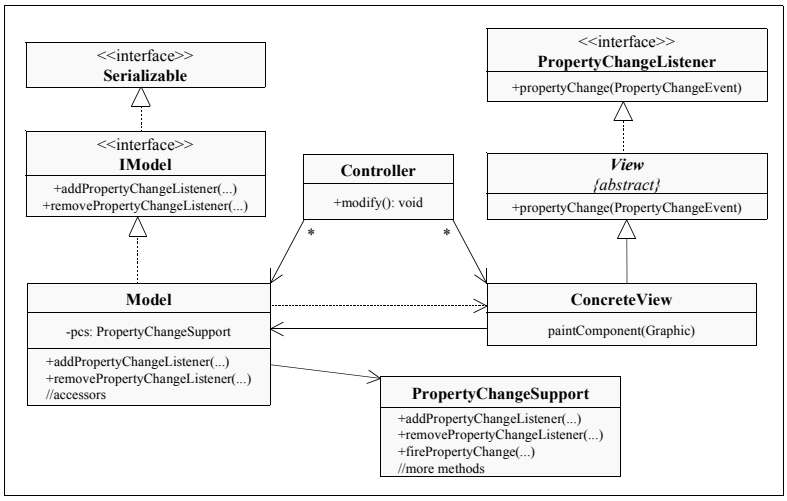
MVC och Java

När man implementerar MVC-modellen i Java rekommenderas (numera) att använda

- klassen `PropertyChangeSupport` och
- interfacet `PropertyChangeListener`

istället för `Observable` och `Observer`.

MVC och Java



I ovanstående design är inte `Model` en subklass till `PropertyChangeSupport`, utan istället har delegering används för att möjliggöra att `Model` skall kunna ärva från någon annan klass.

`PropertyChangeSupport` och `PropertyChangeListener`

I klassen `PropertyChangeSupport` finns bl.a följande metoder:

void addPropertyChangeListener(PropertyChangeListener li)
lägger till lyssnaren `li`

void removePropertyChangeListener(PropertyChangeListener li)
tar bort lyssnaren `li`

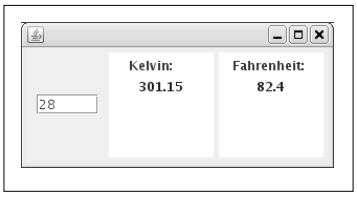
void firePropertyChange(String propertyName, Object oldValue, Object newValue)
meddelar alla lyssnare att något har hänt genom att generera en händelse av typen `PropertyChangeEvent`.

I interfacet `PropertyChangeListener` finns endast en metod specificerad:

void propertyChange(PropertyChangeEvent evt)
Denna metod anropas när en `PropertyChangeEvent` inträffar.

MVC – ett enkelt exempel

Vi visar hur MVC-mönstret kan användas genom att skriva ett mycket enkelt program som konverterar temperaturer angivna i Celsius till Kelvin respektive till Farenheit.



IModell**:**

```
import java.beans.*;
import java.io.*;
public interface IModel extends Serializable {
   void addPropertyChangeListener(PropertyChangeListener l);
   void removePropertyChangeListener(PropertyChangeListener l);
}
```

Modell:

```
import java.beans.*;
public class Model implements IModel {
   private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);
   private int degreeCelsius;
   public Model() {
      degreeCelsius = 0;
   }//constructor
   public void addPropertyChangeListener(PropertyChangeListener l) {
       pcs.addPropertyChangeListener(l);
   }//addPropertyChangeListener
   public void removePropertyChangeListener(PropertyChangeListener l) {
      pcs.removePropertyChangeListener(l);
   }//removePropertyChangeListener
   public int getValue() {
      return degreeCelsius;
   }//getValue
   public void setValue(int newValue) {
      int oldValue = degreeCelsius;
      degreeCelsius = newValue;
      pcs.firePropertyChange("value", oldValue, newValue);
   }//setValue
}//Model
```

View:

```
import java.beans.*;
import javax.swing.*;
public abstract class View extends JPanel implements PropertyChangeListener {
   protected Model model;
   public Model getModel() {
      return model;
   }//getModel
   public void setModel(Model m) {
      if (model != null)
         model.removePropertyChangeListener(this);
      model = m;
      if (model != null)
         model.addPropertyChangeListener(this);
   }//setModel
   public abstract void propertyChange(PropertyChangeEvent e);
   }//propertyChange
}//View
```

Konkreta vyer – KelvinView:

```
import java.awt.*;
import javax.swing.*;
import java.beans.*;
public class  KelvinView extends View {
   private JLabel k = new JLabel();
   public KelvinView(Model m) {
      setModel(m);
      setPreferredSize(new Dimension(100,100));
      k.setForeground(Color.BLUE);
      setBackground(Color.WHITE);
      add(new JLabel("Kelvin:     "));
      add(k);
   }//constructor
   public void propertyChange(PropertyChangeEvent e) {
      if (model != null) {
         double toK  = model.getValue()+273.15;
         k.setText(""+ toK);
      }
   }//propertyChange
}//KelvinView
```

Konkreta vyer – FarhenheitView:

```
import java.awt.*;
import javax.swing.*;
import java.beans.*;
public class  FahrenheitView extends View {
   private JLabel f = new JLabel();
   public FahrenheitView(Model m) {
      setModel(m);
      setPreferredSize(new Dimension(100,100));
      f.setForeground(Color.BLUE);
      setBackground(Color.WHITE);
      add(new JLabel("Fahrenheit: "));
      add(f);
   }//constructor
   public void propertyChange(PropertyChangeEvent e) {
      if (model != null) {
         double toK  =  model.getValue()*9.0/5.0+32;
         f.setText(""+ toK);
      }
   }//propertyChange
}//FahrenheitView
```

Controller:

```
import javax.swing.*;
import java.awt.event.*;
public class Controller extends JPanel implements ActionListener
{
   private Model model;
   private JTextField f = new JTextField(5);
   public Controller(Model m) {
      model = m;
      add(f);
      f.addActionListener(this);
   }//constructor
   public void actionPerformed(ActionEvent e) {
      if (model != null)
         model.setValue(Integer.parseInt(f.getText()));
   }//actionPerformed
}//Controller
```

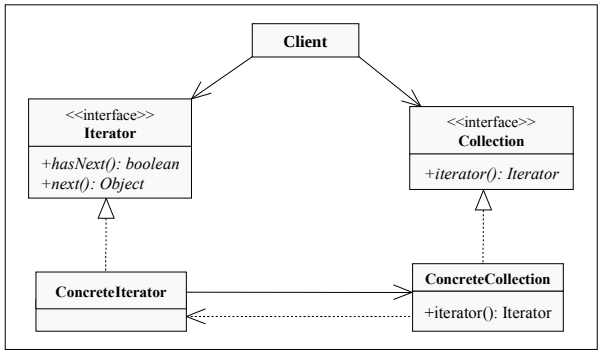
Hopkoppling av komponenterna:

```
import java.awt.*;
import javax.swing.*;
public class MVCDemo extends JFrame {
   public MVCDemo() {
      Model m = new Model();
      View kv = new KelvinView(m);
      View fv = new FahrenheitView(m);
      Controller c = new Controller(m);
      setLayout(new FlowLayout());
      add(c);
      add(kv);
      add(fv);
      pack();
      setVisible(true);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
   }//constructor
   public static void main(String[] arg){
      MVCDemo demo = new MVCDemo();
   }//main
}//MVCDemo
```

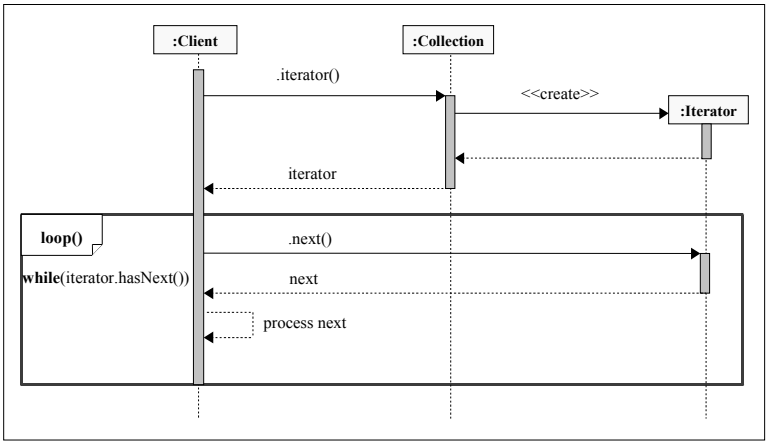
Designmönstret Iterator

Med en iterator kan man förflytta sig igenom en samling av data, utan att behöva känna till detaljer om datans interna representation.

Iterator-mönstret är användbart eftersom det tillhandahåller ett definierat sätt att genomlöpa en uppsättning av dataelement, utan att exponera vad som händer inne i klassen.

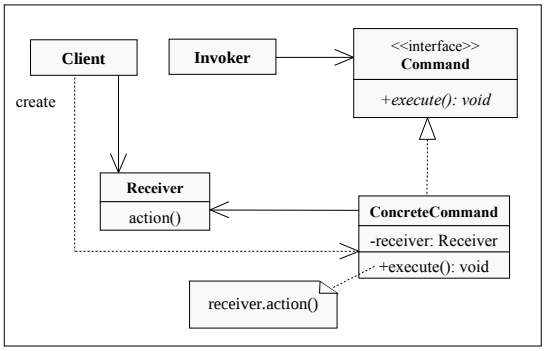


Designmönstret Iterator



Designmönstret Command

Designmönstret Command kapslar in kommandon som objekt och låter klienter få tillgång till olika uppsättningar med kommandon.



Designmönstret Command

```
//Command
public interface Order {
   public abstract void execute ( );
}//Order
```

```
// Invoker.
public class Agent {
   public Agent() {}
   public void placeOrder(Order order) {
      order.execute();
   }//placeOrder
}//Agent
```

```
// Receiver class.
public class StockTrade {
   public void buy() {
      System.out.println("You want to buy stocks");
   }//buy
   public void sell() {
      System.out.println("You want to sell stocks ");
   }//sell
}//StockTrade
```

Designmönstret Command

```
//ConcreteCommand class.
public class BuyStockOrder implements Order {
   private StockTrade stock;
   public BuyStockOrder(StockTrade st) {
      stock = st;
   }//constructor
   public void execute( ) {
      stock.buy( );
   }//execute
}//BuyStockOrder
```

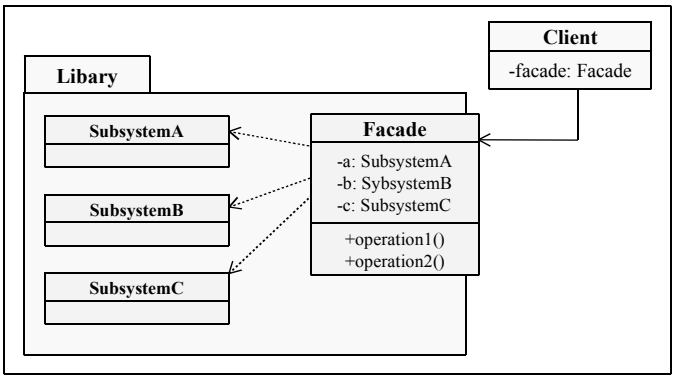
```
//ConcreteCommand class.
public class SellStockOrder implements Order {
   private StockTrade stock;
   public SellStockOrder(StockTrade st) {
      stock = st;
   }//constructor
   public void execute( ) {
      stock.sell( );
   }//execute
}//SellStockOrder
```

```
// Client
public class Client {
   public static void main(String[] args) {
      StockTrade stock = newStockTrade();
      BuyStockOrder bsc = newBuyStockOrder (stock);
      SellStockOrder ssc = newSellStockOrder (stock);
      Agent agent = newAgent();
      agent.placeOrder(bsc); // Buy Shares
      agent.placeOrder(ssc); // Sell Shares
   }//main
}//Client
```

Designmönstret Façade

Designmönstret Façade används för att dölja komplexitet för användarna.

- lättare att använda fasaden än det ursprungliga systemet
- kan byta implementation av det som fasaden gömmer
- mindre beroenden



Designmönstret Façade

```
class SubsystemA {
   String A1() {
      return "Subsystem A, Method A1\n";
   }
   String A2() {
      return "Subsystem A, Method A2\n";
   }
}

class SubsystemB {
   String B1() {
      return "Subsystem B, Method B1\n";
   }
}

class SubsystemC {
   String C1() {
      return "Subsystem C, Method C1\n";
   }
}
```

```
public class Facade {
   private SubsystemA a = new SubsystemA();
   private SubsystemB b = new SubsystemB();
   private SubsystemC c = new SubsystemC();
   public void operation1() {
      System.out.println("Operation 1\n" +
                         + a.A1()
                         + a.A2()
                         + b.B1());
   }
   public void operation2() {
      System.out.println("Operation 2\n"
                         + b.B1()
                         + c.C1());
   }
}
```

```
public class Client {
   public static void main (String[] arg) {
      Facade facade = new Facade();
      facade.operation1();
      facade.operation2();
   }
}
```

Designmönstret Composite

Designmönstret Composite används för att sätter samman objekt till trädstrukturer för att representera ”part-whole” hierarkier. Composite låter klienter hantera individuella objekt och grupper av kopplade objekt likformigt.