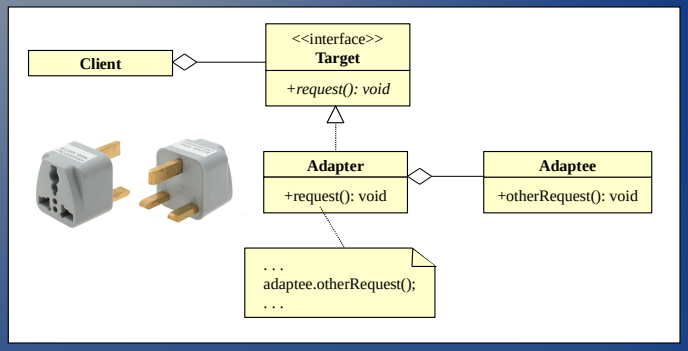
Föreläsning 10
Fler Design Patterns

Adapter, Factory Method
Decorator, Observer
Model-View-Controller
Bridge, Composite
Command

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, till gränsnittet som klassen Adaptee tillhandahåller

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
}//ConcreteSquarePeg
```

```
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
```

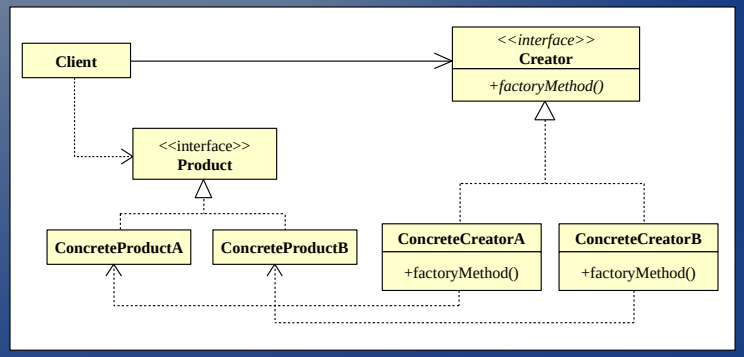
```
public class TestPegs {
   public static void main(String[] args) {
      RoundPeg roundPeg = new RoundPeg();
      SquarePeg squarePeg = new ConcreteSquarePeg();
      SquarePeg adapter = new PegAdapter(roundPeg);
      squarePeg.insert("Inserting square peg!");
      adapter.insert("Inserting round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

Utskrift:
SquarePeg insert: Inserting square peg!
RoundPeg insertIntoHole: Inserting round peg!

Designmönstret Factory Method

Factory method används när man

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



Designmönstret Factory Method

Produkter

```
public interface Product {
   abstract public 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 ProductC implements Product {
   public String shipFrom () {
      return " from Mexico";
   }
}
```

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


Designmönstret Factory Method

Fabriker

```
public interface Creator {
   abstract public Product factoryMethod(int month);
}
```

```
public class ConcreteCreatorA implements 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();
   }
}
```

```
public class ConcreteCreatorB  implements Creator {
   public Product factoryMethod(int month) {
      return new ProductC();
   }
}
```


Designmönstret Factory Method

Client

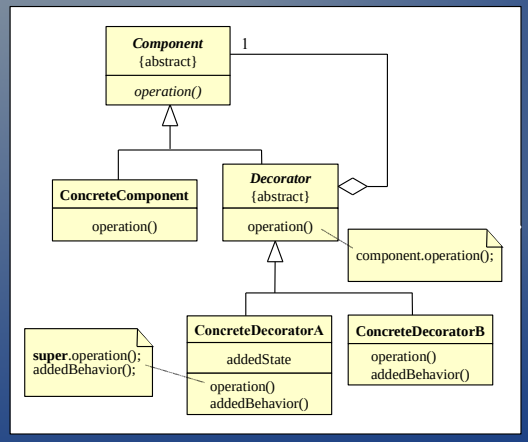
```
public class Main {
   public static void main(String[] args) {
      Creator c = new ConcreteCreatorA();
      System.out.println("From ConcreteCreatorA:");
      printYearlyDeliver(c);
      c = new ConcreteCreatorB();
      System.out.println("From ConcreteCreatorB:");
      printYearlyDeliver(c);
   }

   private static void printYearlyDeliver(Creator c) {
      for (int i = 1; i <= 12; i++) {
         Product product = c.factoryMethod(i);
         System.out.println("Avocados " + product.shipFrom());
      }
   }
}//Main
```

Körningsexempel:

```
From ConcreteCreatorA:
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

From ConcreteCreatorB:
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
Avocados  from Mexico
```

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.

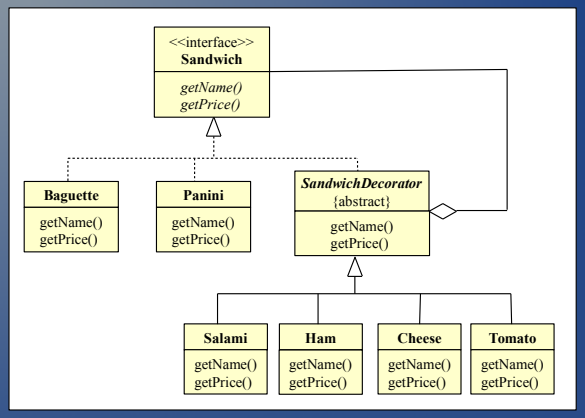
ConcreteDecorator: Lägger till funktionalitet på objektet.

Exempel på användning av Decorator

Antag att vi har en smörgåsbutik. Det finns ett antal brödsorter att välja mellan och ett antal olika pålägg.

Kunden skall kunna kombinera sin smörgås efter eget önskemål.

Exempel på användning av Decorator



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 interface Sandwich {
   public String getName();
   public 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 implements Sandwich {
   public String getName() {
      return "Panini";
   }//getName
   public double  getPrice() {
      return 10.5;
   }//getPrice
}//Panini
```

```
public class  Baguette implements Sandwich {
   public String getName() {
      return "Baguette";
   }//getName
   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:

```
abstract public class SandwichDecorator implements Sandwich {
   protected Sandwich decoratedSandwich;
   public SandwichDecorator(Sandwich sandwich) {
     decoratedSandwich = sandwich;
   }//constructor
   public String getName() {
     return decoratedSandwich.getName();
   }//getName
   public double getPrice() {
     return decoratedSandwich.getPrice();
   }//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) {
      super(sandwich);
   }//constructor
   public String getName() {
      return super.getName() + " " + Cheese";
   }//getNam
   private double chargeCheese() {
      return 5.5;
   }
   public double getPrice() {
      return decoratedSandwich.getPrice()
             + chargeCheese();
   }//getPrice
}//Cheese
```

```
public class Ham extends SandwichDecorator {
   public Ham (Sandwich sandwich) {
      super(sandwich);
   }//constructor
   public String getName() {
     return super.getName() + " " + Ham";
   }//getName
   private double chargeHam() {
      return 7.5;
   }
   public double getPrice() {
      return decoratedSandwich.getPrice()
             + chargeHam();
   }//getPrice
}//Ham
```


Exempel på användning av Decorator

```
public class Salami extends SandwichDecorator {
   public Salami (Sandwich sandwich) {
      super(sandwich);
   }//constructor
   public String getName() {
      return super.getName() + " " + Salami";
   }//getNam
   private double chargeSalami() {
      return 8.5;
   }
   public double getPrice() {
      return decoratedSandwich.getPrice()
             + chargeSalami();
   }//getPrice
}//Salami
```

```
public class Tomato extends SandwichDecorator {
   public Tomato (Sandwich sandwich) {
      super(sandwich);
   }//constructor
   public String getName() {
     return super.getName() + " " + Tomato";
   }//getName
   private double chargeTomato() {
      return 2.5;
   }
   public double getPrice() {
      return decoratedSandwich.getPrice()
             + chargeTomato();
   }//getPrice
}//Tomato
```

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.

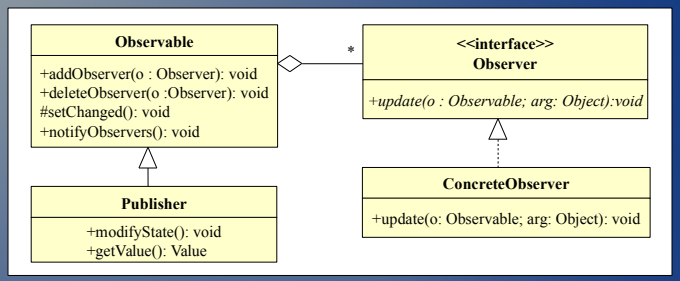
Designmönstret Observer

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

Java stöder designmönstret Observer genom:

- klassen Observable och interfacet Observer
- klassen PropertyChangeSupport och interfacet PropertyChangeListener

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*.
Ovanstående klassdigarm överensstämmer med interfacet Observer och klassen Observable i Java.

Designmönstret Observer

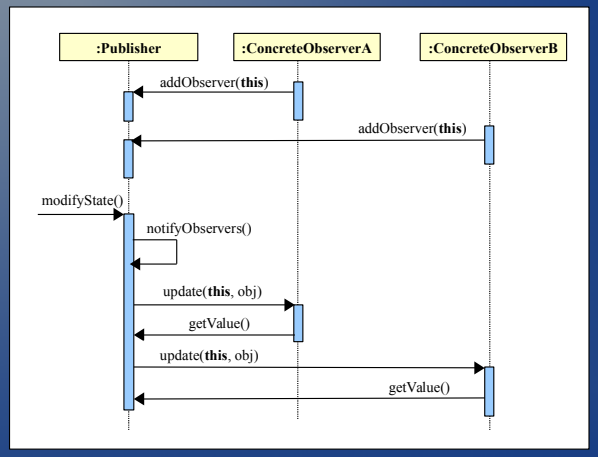
Klassen Observable i Java har bl.a följande metoder:

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. Måste anropas för att notifyObservers() verkligen kommer att meddela observatörerna.

Interfacet Observer i Java 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
}//IntrusionDetector
```

```
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
```

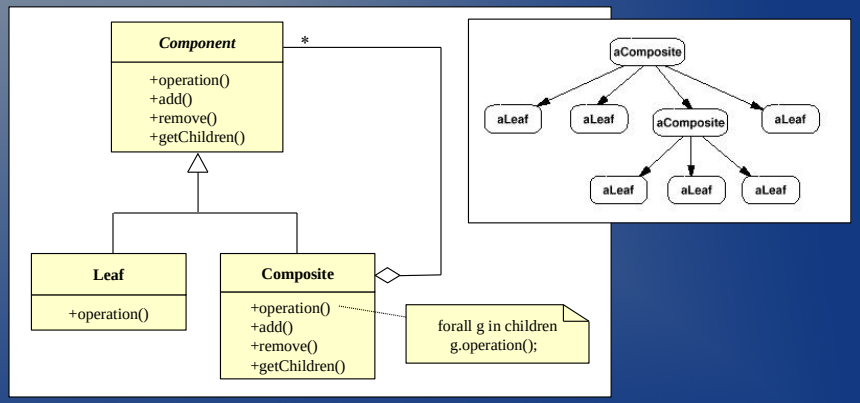

Exempel – Datorintrång

```
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 Composite

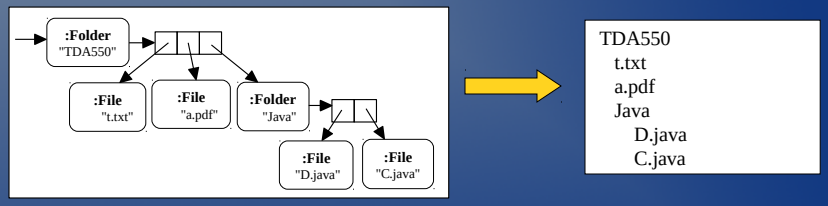
Designmönstret Composite används för att sätter samman objekt till trädstrukturer för att representera ”part-whole” hierarkier. Composite låter klienter hantera individuella objekt och grupper av kopplade objekt på ett likformigt sätt.



Designmönstret Composite

Exempel:

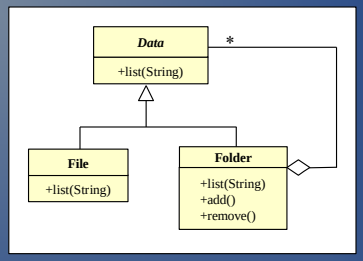
En folder kan innehålla filer och andra foldrar. Antag att vi i en applikation behöver en metod som när den anropas på en fil skriver filens namn och när den anropar på en folder skriver ut namnet på foldern samt innehållet i foldern och innehållet i alla underfoldlar:




```
TDA550
  t.txt
  a.pdf
  Java
    D.java
    C.java
```


Designmönstret Composite

Exempel:



Designmönstret Composite

```
interface Data {
   public void list(String str);
}
```

```
public class File implements Data {
   private String name;
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   @Override
   public void list(String str) {
      System.out.println(str + this.getName());
   }
}//File
```

```
import java.util.*;
public class Folder implements Data {
   private String name;
   private List<Data> folder = new ArrayList<Data>();
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   @Override
   public void list(String str) {
      System.out.println(str + this.getName());
      for(Data data : folder) {
         data.list("   " + str);
      }
   }
   public void add(Data data) {
      folder.add(data);
   }
   public void remove(Data data) {
      folder.remove(data);
   }
}//Folder
```


Designmönstret Composite

```
//Client Program
public class CompositePattern {
   public static void main(String[] args) {
      Folder f1 = new Folder();    f1.setName("Folder 1");
      Folder f2 = new Folder();    f2.setName("Folder 2");
      Folder f3 = new Folder();    f3.setName("Folder 3");
      File file1 = new File();     file1.setName("File 1");
      File file2 = new File();     file2.setName("File 2");
      File file3 = new File();     file3.setName("File 3");
      File file4 = new File();     file4.setName("File 4");
      f1.add(file1);
      f1.add(f3);
      f2.add(file2);
      f3.add(f2);
      f3.add(file3);
      f3.add(file4);
      f1.list();
   }
}//CompositePattern
```

```
Folder 1
  File 1
  Folder 3
    Folder 2
      File 2
    File 3
    File 4
```

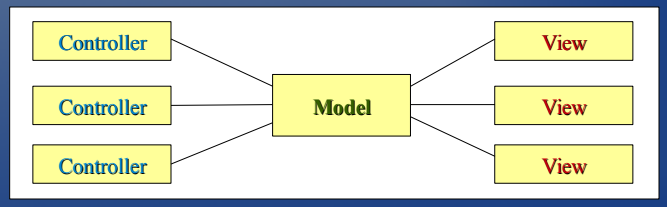
Designmönstret Model – View - Control

Designmönstret Model-View-Control (MVC) frikopplar (det grafiska) 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 klasser som ansvarar för tillståndet i applikationen.

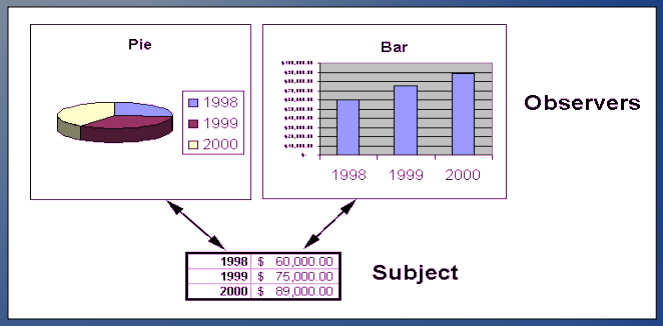
View: De klasser som ansvarar för att presentera modellen (som grafiska användargränssnitt).

Control: De klasser som handhar de händelsers som påverkar modellens tillstånd.



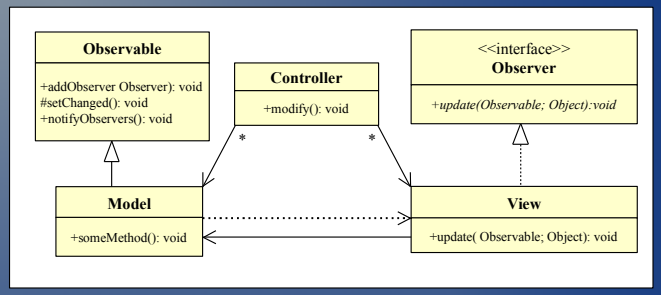
Model-View-Control

Genom att separera modellen och vyn kan förändringar göras i vyn och flera vyer skapas utan att behöva ändra koden i den underliggande modellen.



När tillståndet i modellen förändras skall vyerna uppdateras, vi har alltså designmönstret Observer, mellan modellen och vyn.

Model-View-Control



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

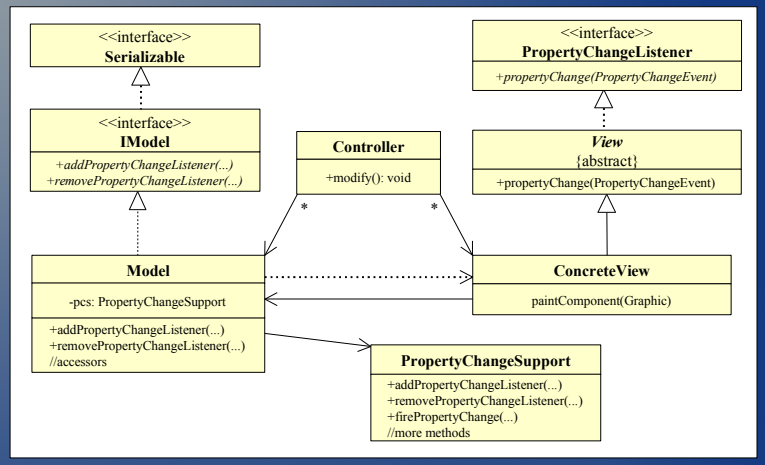
MVC och Java

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 design ovan 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

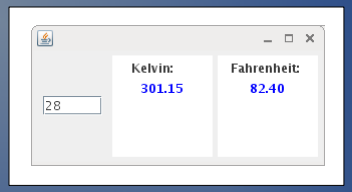
void firePropertyChange(String propertyName, Object oldValue, Object newValue)
meddelar alla lyssnare att något har hänt genom att generera en händelse av typen PropertyChangeEvent. oldValue och newValue måste vara olika för att verkligen skall genereras.

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);
}//IModel
```


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
```

Avfyras endast om oldValue != newValue

View:

```
import java.beans.*;
import javax.swing.*;
public abstract class View extends JPanel implements PropertyChangeListener {
   protected Model model;
   public View(Model m) {
      model = m;
      model.addPropertyChangeListener(this);
   }//setModel
   public abstract void propertyChange(PropertyChangeEvent e);
}//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) {
      super(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) {
      double toK  = model.getValue()+273.15;
      k.setText(String.format("%.2f", 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) {
      super(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) {
      double toF  =  model.getValue()*9.0/5.0+32;
      f.setText(String.format("%.2f", toF));
   }//propertyChange
}//FahrenheitView
```

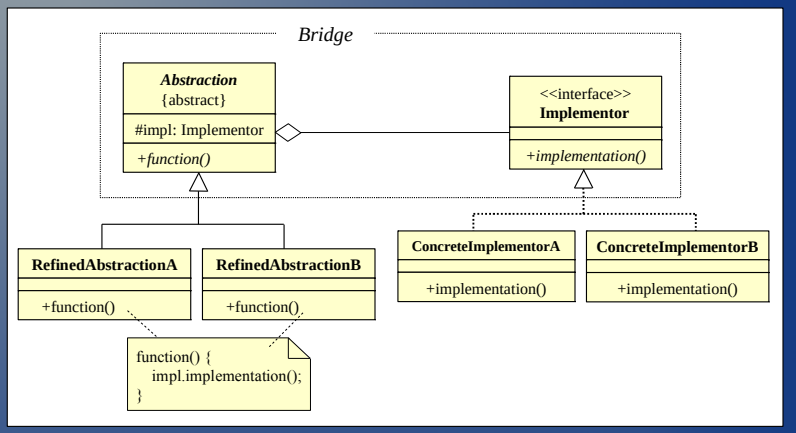

Controller:

```
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) {
      model.setValue(Integer.parseInt(f.getText()));
   }//actionPerformed
}//Controller
```

Hopkoppling av komponenterna:

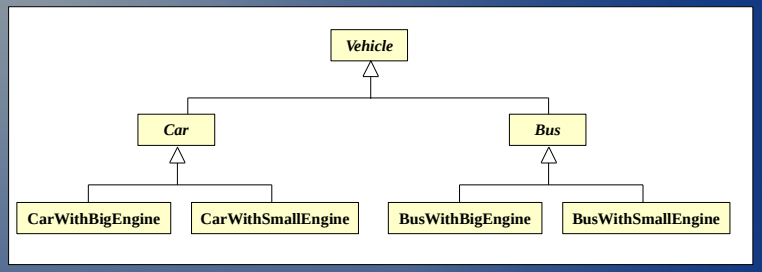
```
import java.awt.*;
import javax.swing.*;
public class MVCDemo extends JFrame {
   public MVCDemo() {
      Model m = new Model();
      View kv = new KelvinView(m);
      View fv = new FahrenheitView(m);
      Controller c = new Controller(m);
      setLayout(new FlowLayout());
      add(c);
      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 Bridge

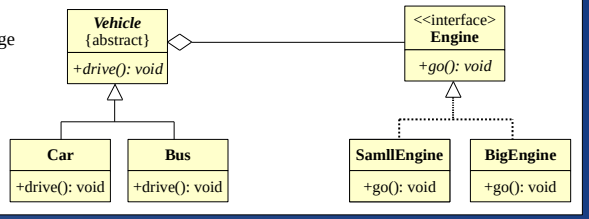


Att frigöra en abstraktion från sin implementering så att båda kan variera oberoende.

Designmönstret Bridge



Med användning av designmönstret Bridge



Designmönstret Bridge

```
public abstract class Vehicle {
        protected Engine engine;
        private int weightInKilos;
        public Vehicle(int weightInKilos, Engine engine) {
            this.weightInKilos = weightInKilos;
            this.engine = engine;
        }
        public abstract void drive();
        public void setEngine(Engine engine) {
           this.engine = engine;
        }
        public void reportOnSpeed(int horsepower) {
           int ratio = weightInKilos / horsepower;
           if (ratio < 3)
                System.out.println("The vehicle is going at a fast speed.");
           else if ((ratio >= 3) && (ratio <= 8))
                System.out.println("The vehicle is going an average speed.");
           else
                System.out.println("The vehicle is going at a slow speed.");
        }
}//Vehicle
```

Designmönstret Bridge

```
public class Car extends Vehicle {
   public Car(Engine engine) {
      super(600, engine);
   }
   @Override
   public void drive() {
      System.out.println("\nThe car is driving");
      int horsepower = engine.go();
      reportOnSpeed(horsepower);
   }
}//Car
```

```
public class Bus extends Vehicle {
   public Bus(Engine engine) {
       super(3000, engine);
   }
   @Override
   public void drive() {
      System.out.println("\nThe bus is driving");
      int horsepower = engine.go();
      reportOnSpeed(horsepower);
   }
}//Bus
```


Designmönstret Bridge

```
public interface Engine {
   public int go();
}//Engine
```

```
public class SmallEngine implements Engine {
   private int horsepower;
   public SmallEngine() {
      horsepower = 100;
   }
   @Override
   public int go() {
      System.out.println("The small engine is running");
      return horsepower;
   }
}//SmallEngine
```

```
public class BigEngine implements Engine {
   private int horsepower;
   public BigEngine() {
      horsepower = 350;
   }
   @Override
   public int go() {
      System.out.println("The big engine is running");
      return horsepower;
   }
}//BigEngine
```

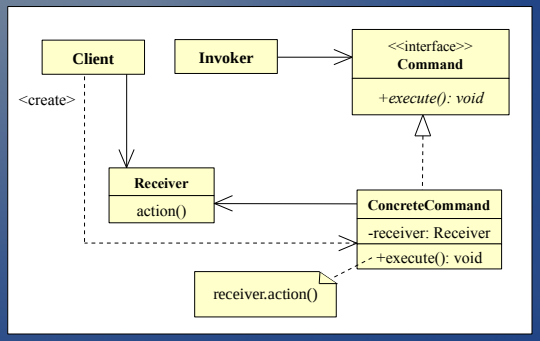

Designmönstret Bridge

```
public class BridgeDemo {
   public static void main(String[] args) {
      Vehicle vehicle = new Bus(new SmallEngine());
      vehicle.drive();
      vehicle.setEngine(new BigEngine());
      vehicle.drive();

      vehicle = new Car(new SmallEngine());
      vehicle.drive();
      vehicle.setEngine(new BigEngine());
      vehicle.drive();
   }
}//BridgeDemo
```

```
The bus is driving
The small engine is running
The vehicle is going at a slow speed.

The bus is driving
The big engine is running
The vehicle is going at a average speed.

The car is driving
The small engine is running
The vehicle is going an average speed.

The car is driving
The big engine is running
The vehicle is going at a fast speed.
```


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
```

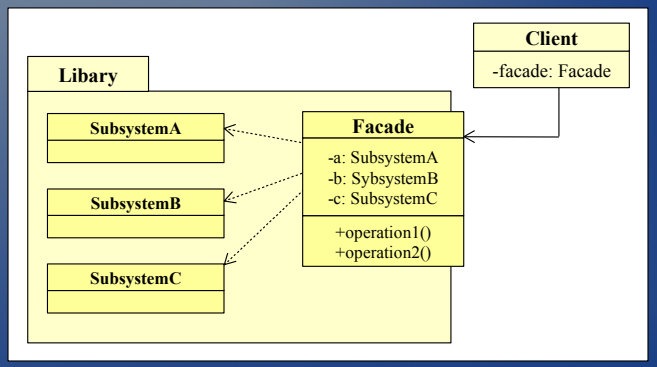
```
//ConcreteCommand class.
public class SellStockOrder implements Order {
   private StockTrade stock;
   public SellStockOrder(StockTrade st) {
      stock = st;
   }//constructor
   public void execute( ) {
      stock.sell( );
   }//execute
}//SellStockOrder
```

```
// Client
public class Client {
   public static void main(String[] args) {
      StockTrade stock = new StockTrade();
      BuyStockOrder bsc = new BuyStockOrder (stock);
      SellStockOrder ssc = new SellStockOrder (stock);
      Agent agent = new Agent();
      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();
   }
}
```