

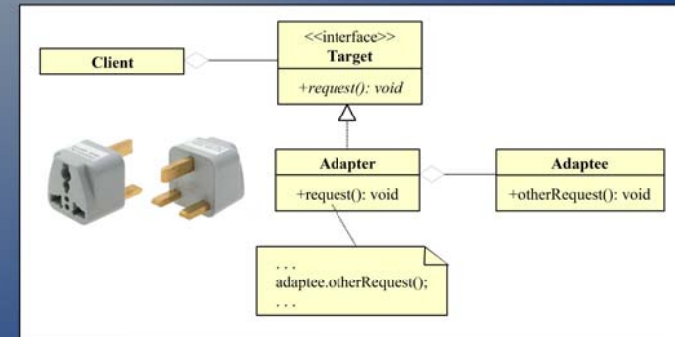
Föreläsning 10

Fler designmönster

Adapter, Façade, Factory Method, Decorator, Composite, Observer, Model-View-Controller

Designmönstret Adapter

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



Target beskriver gränssnittet som Client vill ha det. I klassen Adapter sker konverteringen av anropen, till anropen som klassen Adaptee tillhandahåller.

2

Designmönstret Adapter

```

public interface SquarePeg {
    insertSquarePeg();
}
    
```

```

public class OrdinarySquarePeg implements SquarePeg {
    public void insertSquarePeg() {
        System.out.println("SquarePeg insert");
    }
}
    
```

```

public class RoundPeg {
    public void insertRoundPeg() {
        System.out.println("RoundPeg insert");
    }
}
    
```

3

Designmönstret Adapter

```

public class PegAdapter implements SquarePeg {
    private RoundPeg roundPeg;
    public PegAdapter(RoundPeg peg) {
        this.roundPeg = peg;
    }
    public void insertSquarePeg() {
        roundPeg.insertRoundPeg(); // translate
    }
}
    
```

```

// Client
public class TestPegs {
    public static void main(String[] args) {
        SquarePeg squarePeg = new OrdinarySquarePeg();
        squarePeg.insertSquarePeg();
        // To use a RoundPeg we need an adapter!
        RoundPeg roundPeg = new RoundPeg();
        SquarePeg adapter = new PegAdapter(roundPeg);
        adapter.insertSquarePeg();
    }
}
    
```

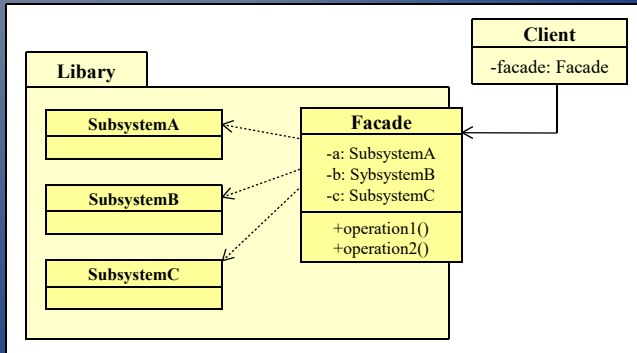
Utskrift:
SquarePeg insert
RoundPeg insert

4

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



5

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());
    }
}

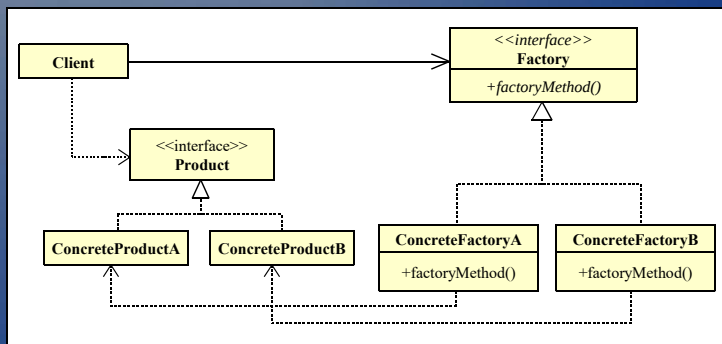
public class Client {
    public static void main (String[] arg) {
        Facade facade = new Facade();
        facade.operation1();
        facade.operation2();
    }
}
    
```

6

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.



7

Designmönstret Factory Method

```

public interface Supplier {
    String shipFrom();
}

public class SupplierAfrica implements Supplier {
    public String shipFrom() {
        return " from South Africa";
    }
}

public class SupplierSpain implements Supplier {
    public String shipFrom() {
        return " from Spain";
    }
}

public class SupplierMexico implements Supplier {
    public String shipFrom() {
        return " from Mexico";
    }
}

public class DefaultSupplier implements Supplier {
    public String shipFrom() {
        return " not available";
    }
}
    
```

Leverantörer (Produkter)

8

Designmönstret Factory Method

Fabriker

```
public interface SupplierFactory {
    Supplier getSupplier(int month);
}
```

```
public class SupplierFactoryA implements SupplierFactory {
    public Supplier getSupplier(int month) {
        if (month >= 4 && month <= 11)
            return new SupplierAfrica();
        else if (month == 1 || month == 2 || month == 12)
            return new SupplierSpain();
        else
            return new DefaultSupplier();
    }
}
```

```
public class SupplierFactoryB implements SupplierFactory {
    public Supplier getSupplier(int month) {
        return new SupplierMexico();
    }
}
```

9

Designmönstret Factory Method

Client

```
public class Main {
    public static void main(String[] args) {
        System.out.println("From SupplierFactoryA:");
        printYearlyDeliver(new SupplierFactoryA());
        System.out.println("From SupplierFactoryB:");
        printYearlyDeliver(new SupplierFactoryB());
    }

    private static void printYearlyDeliver(SupplierFactory f) {
        for (int i = 1; i <= 12; i++) {
            Supplier supplier = f.getSupplier(i);
            System.out.println("Avocados " + supplier.shipFrom());
        }
    }
}
```

Körningsexempel:

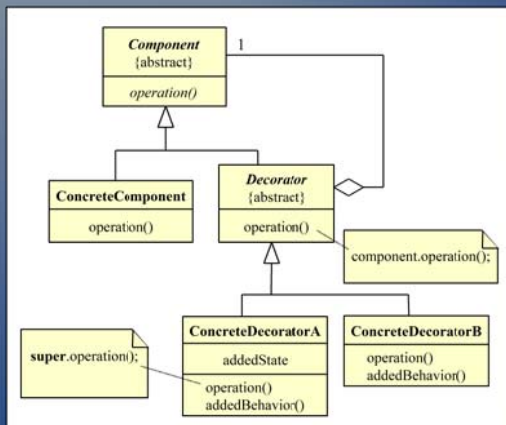
From SupplierFactoryA:
 Avocados from Spain
 Avocados from Spain
 Avocados not available
 Avocados from South Africa
 Avocados from South Africa
 Avocados from South Africa
 Avocados from South Africa
 Avocados from South Africa
 Avocados from South Africa
 Avocados from South Africa
 Avocados from Spain

From SupplierFactoryB:
 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

10

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.

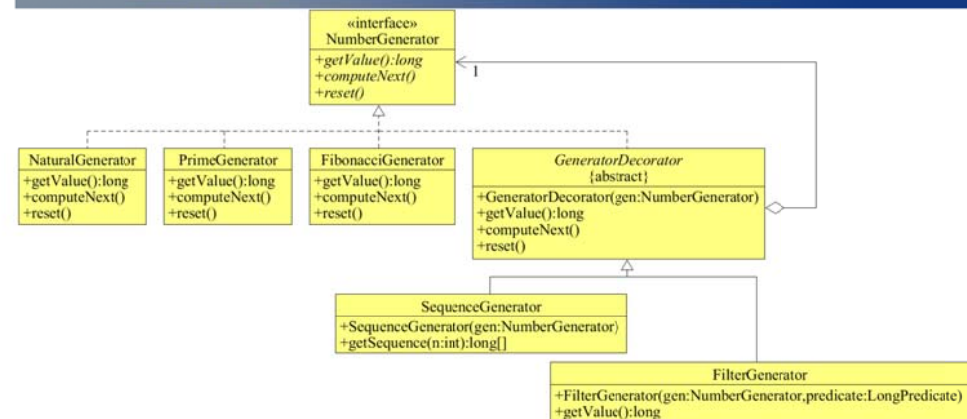
Decorator: Har en referens till ett Component-objekt och definierar ett gränssnitt som överensstämmer med gränssnittet för Component.

ConcreteDecorator: Läger till funktionalitet på objektet.

11

Exempel på användning av Decorator

Gränssnittet NumberGenerator beskriver klasser som kan skapa olika talserier, t.ex. naturliga tal: 0, 1, 2,..., primtal: 2, 3, 5, 7,..., Fibonaccital: 1, 1, 2, 3, 5, 8,..., etc.



12

Exempel på användning av Decorator

Gränssnittet NumberGenerator:

```
public interface NumberGenerator {  
    long getValue();  
    void computeNext();  
    void reset();  
}
```

13

Exempel på användning av Decorator

En generator för de naturliga talen 0, 1, 2, ... :

```
public class NaturalGenerator implements NumberGenerator {  
    private long n = 0;  
    @Override  
    public long getValue() { return n; }  
    @Override  
    public void computeNext() { n++; }  
    @Override  
    public void reset() { n = 0; }  
}
```

14

Exempel på användning av Decorator

En generator för primtalen 2, 3, 5, 7, ... :

```
public class PrimeGenerator implements NumberGenerator {  
    private long currentPrime;  
    public PrimeGenerator() { reset(); }  
    private boolean isPrime(long x) {  
        if ( x == 2 ) return true;  
        for (long i = 2; i <= (long)Math.ceil(Math.sqrt(x)); i++)  
            if (x % i == 0) return false;  
        return true;  
    }  
    public long getValue() { return currentPrime; }  
    public void computeNext() {  
        do { currentPrime++; } while (! isPrime(currentPrime));  
    }  
    public void reset() { currentPrime = 2; }  
}
```

15

Exempel på användning av Decorator

En generator för Fibonaccitalen 1, 1, 2, 3, 5, 8, 13, ... :

```
public class FibonacciGenerator implements NumberGenerator {  
    private long currentFib, nextFib;  
    public FibonacciGenerator() { reset(); }  
    public long getValue() { return currentFib; }  
    public void computeNext() {  
        nextFib += currentFib;  
        currentFib = nextFib - currentFib;  
    }  
    public void reset() {  
        currentFib = 1;  
        nextFib = 1;  
    }  
}
```

16

Exempel på användning av Decorator

Nu inför vi vår Decorator-klass, GeneratorDecorator, vilken är en abstrakt klass som implementerar NumberGenerator och som har en instansvariabel av typen NumberGenerator:

```
public abstract class GeneratorDecorator implements NumberGenerator {
    private final NumberGenerator generator;
    public GeneratorDecorator(NumberGenerator generator) {
        this.generator = generator;
    }
    public long getValue() { return generator.getValue(); }
    public void computeNext() { generator.computeNext(); }
    public void reset() { generator.reset(); }
}
```

En GeneratorDecorator *delegerar* de tre metodanropen till sitt dekorerade objekt. Subklasser kan addera ytterligare metoder.

17

Exempel på användning av Decorator

Nu kan vi skapa dekorerande subklasser, t.ex. för att samla talen från en generator i ett fält:

```
public final class SequenceGenerator extends GeneratorDecorator {
    public SequenceGenerator(NumberGenerator generator) {
        super(generator);
    }
    /**
     * Compute an array of numbers.
     * @param n the number of numbers to compute
     * @return an array containing the next n numbers in the sequence
     */
    public long[] getSequence(int n) {
        long[] numArray = new long[n];
        for (int i = 0; i < n; i++) {
            numArray[i] = getValue();
            computeNext();
        }
        return numArray;
    }
}
```

18

Exempel på användning av Decorator

Denna generatorklass filtrerar talen från en annan generator m.h.a. ett *predikat*:

```
import java.util.function.LongPredicate;
public class FilterGenerator extends GeneratorDecorator {
    private LongPredicate predicate;

    public FilterGenerator(NumberGenerator gen, LongPredicate predicate) {
        super(gen);
        this.predicate = predicate;
    }

    public long getValue() {
        while ( ! predicate.test(super.getValue()) )
            computeNext();
        return super.getValue();
    }
}

public interface LongPredicate {
    boolean test(long value);
}
```

19

Exempel på användning av Decorator

```
public class TestGenerators {
    private static void printSomeValues(NumberGenerator g, int n) {
        for (int i = 0; i < n; i++) {
            System.out.print(g.getValue() + " ");
            g.computeNext();
        }
        System.out.println();
    }

    private static void printArray(long[] a) {
        System.out.println(java.util.Arrays.toString(a));
    }

    public static void main(String[] args) {
        printSomeValues(new NaturalGenerator(), 10);
        printSomeValues(new PrimeGenerator(), 10);
        printSomeValues(new FibonacciGenerator(), 10);
        ...
    }
}
```

När programmet körs erhålls utskriften:

```
0 1 2 3 4 5 6 7 8 9
2 3 5 7 11 13 17 19 23 29
1 1 2 3 5 8 13 21 34 55
```

20

Exempel på användning av Decorator

Ex. Låt en SequenceGenerator *addera* metoden getSequence till ett PrimeGenerator-objekt:

```
// main forts.
...
SequenceGenerator seqGen = new SequenceGenerator(new PrimeGenerator());
seqGen.computeNext();
seqGen.computeNext();
printArray(seqGen.getSequence(5));
...
}
```

När koden ovan körs erhålls utskriften:
[5, 7, 11, 13, 17]

21

Exempel på användning av Decorator

```
import java.util.function.LongPredicate;
public class PrimePredicate implements LongPredicate {
    public boolean test(long x) {
        if ( x < 2 ) return false;
        if ( x == 2 ) return true;
        for (long i = 2; i <= (long)Math.ceil(Math.sqrt(x)); i++)
            if (x % i == 0)
                return false;
        return true;
    }
}
```

```
// main forts.
seqGen =
    new SequenceGenerator(
        new FilterGenerator(
            new NaturalGenerator(),
            new PrimePredicate()
        )
    );
printArray(seqGen.getSequence(10));
...
}
```

När koden till höger körs erhålls utskriften:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

22

Exempel på användning av Decorator

Ex. Beräkna de tio minsta Fibonaccitalen som också är primtal:

```
// main forts.
seqGen =
    new SequenceGenerator(
        new FilterGenerator(
            new FibonacciGenerator(),
            new PrimePredicate()
        )
    );
printArray(seqGen.getSequence(10));
...
}
```

När koden ovan körs erhålls utskriften:
[2, 3, 5, 13, 89, 233, 1597, 28657, 514229, 433494437]

23

Exempel på användning av Decorator (lunchövning)

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

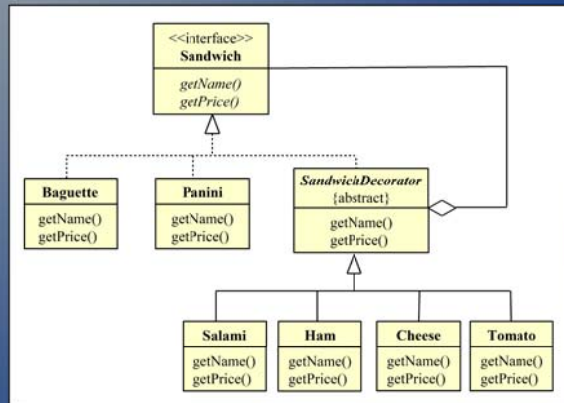


Kunden skall kunna kombinera sin smörgås efter eget önskemål.



24

Exempel på användning av Decorator



25

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 {
    String getName();
    double getPrice();
}
```

26

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";
    }
    public double getPrice() {
        return 10.5;
    }
}
```

```
public class Baguette implements Sandwich {
    public String getName() {
        return "Baguette";
    }
    public double getPrice() {
        return 8.5;
    }
}
```

27

Exempel på användning av Decorator

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

```
abstract public class SandwichDecorator implements Sandwich {
    private Sandwich decoratedSandwich;
    public SandwichDecorator(Sandwich sandwich) {
        decoratedSandwich = sandwich;
    }
    public String getName() {
        return decoratedSandwich.getName();
    }
    public double getPrice() {
        return decoratedSandwich.getPrice();
    }
}
```

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

28

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);
    }
    public String getName() {
        return super.getName() + " + Cheese";
    }
    private double chargeCheese() {
        return 5.5;
    }
    public double getPrice() {
        return super.getPrice() + chargeCheese();
    }
}

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

29

Exempel på användning av Decorator

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

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

30

Exempel på användning av Decorator

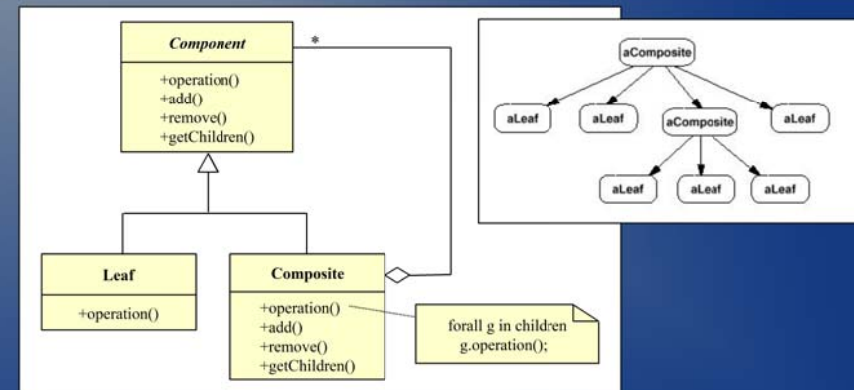
```
public class TestSandwich {
    public static void main (String[] args){
        //lets create a baguette with cheese, ham and tomato
        Sandwich ourBaguette= new Baguette();
        ourBaguette = new Cheese(ourBaguette);
        ourBaguette = new Ham(ourBaguette);
        ourBaguette = new 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");
    }
}
```

När programmet körs erhålls utskriften:
 Baguette + Cheese + Ham + Tomato = 24.0 kronor
 Panini + Salami + Tomato = 21.5 kronor

31

Designmönstret Composite

Designmönstret Composite används för att sätta 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.

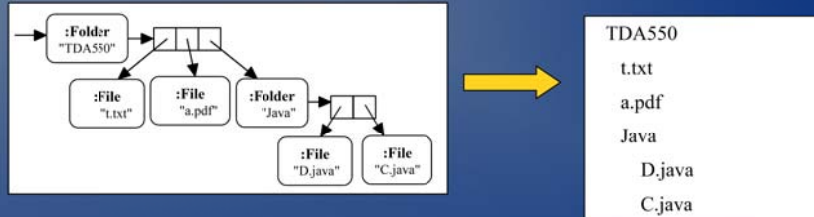


32

Designmönstret Composite

Exempel:

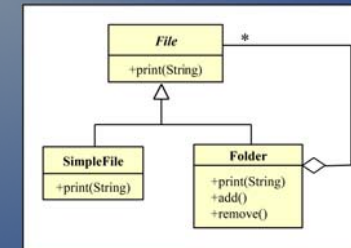
En folder kan innehålla filer och andra foldrar. Antag att vi i en applikation behöver en metod som när den anropas på en fil skriver filens namn och när den anropas på en folder skriver ut namnet på foldern samt innehållet i foldern och innehållet i alla underfoldrar:



33

Designmönstret Composite

Exempel:



34

Designmönstret Composite

```

public abstract class File {
    private String name;
    public File(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    abstract void print(String space);
}

```

```

public class SimpleFile extends File {
    public SimpleFile(String name) {
        super(name);
    }
    @Override
    public void print(String space) {
        System.out.println(space + getName());
    }
}

```

```

import java.util.*;
public class Folder extends File {
    private List<File> folder = new ArrayList<File>();
    public Folder(String name) {
        super(name);
    }
    @Override
    public void print(String space) {
        System.out.println(space + "*" + getName());
        for(File f : folder) {
            f.print(" " + space);
        }
    }
    public void add(File f) {
        folder.add(f);
    }
    public void remove(File f) {
        folder.remove(f);
    }
}

```

35

Designmönstret Composite

```

//Client Program
public class CompositePattern {
    public static void main(String[] args) {
        Folder folder1 = new Folder("Folder 1");
        Folder folder2 = new Folder("Folder 2");
        Folder folder3 = new Folder("Folder 3");
        SimpleFile file1 = new SimpleFile("file 1");
        SimpleFile file2 = new SimpleFile("file 2");
        SimpleFile file3 = new SimpleFile("file 3");
        SimpleFile file4 = new SimpleFile("file 4");
        folder1.add(file1);
        folder1.add(folder2);
        folder3.add(file2);
        folder2.add(folder3);
        folder2.add(file3);
        folder2.add(file4);
        folder1.print();
    }
}

```

Utskrift:
*Folder 1
file 1
*Folder 2
*Folder 3
file 2
file 3
file 4

36

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

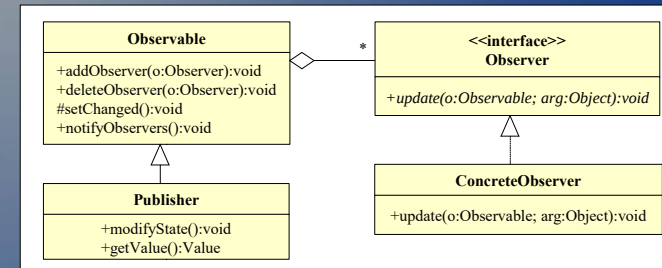
- definierar *en-till-många relation*
- minskar kopplingen mellan objekt

Java stöder designmönstret Observer genom:

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

37

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 klassdiagram överensstämmer med interfacet `Observer` och klassen `Observable` i Java.

38

Designmönstret Observer

Klassen `Observable` i Java har bl.a följande metoder:

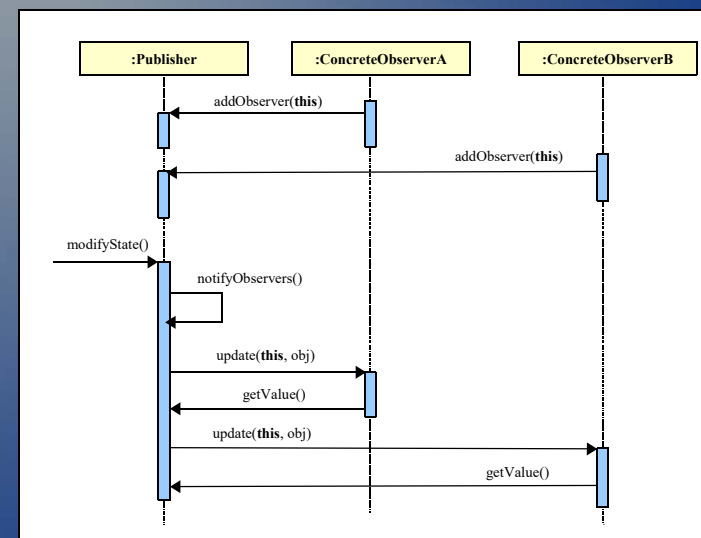
- void** `addObserver(Observer o)` lägger till en ny observatör `o`.
- void** `deleteObserver(Observer o)` tar bort observatören `o`.
- void** `notifyObservers()` meddelar alla observatörer att tillståndet har förändrats.
- protected void** `setChanged()` markerar att tillståndet har ändrats. Måste anropas för att `notifyObservers()` verkligen kommer att meddela observatörerna.

Interfacet `Observer` i Java har metoden:

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

39

Designmönstret Observer



40

Exempel – Datorintrång

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

```
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;
    }
    public void subscribe() {
        dectector.addObserver(this);
    }
    public void update(Observable server, Object err) {
        System.out.println( name + " got the message!" );
    }
}
```

41

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();
    }
}
```

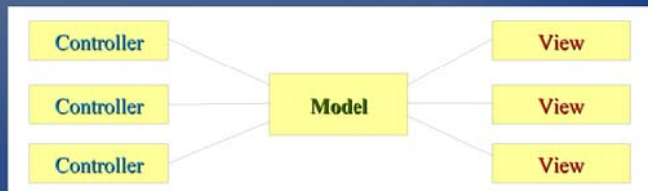
Rune got the message!
Kalle got the message!

42

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:

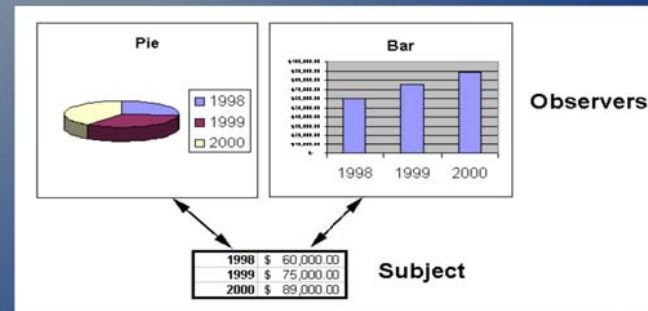
- Model:** Den datamodell som används – en abstrakt representation av den information som bearbetas i programmet. Utgörs av de klasser som ansvarar för tillståndet i applikationen.
- View:** De klasser som ansvarar för att presentera modellen (som grafiska användargränssnitt).
- Control:** De klasser som handhar de händelser som påverkar modellens tillstånd.



43

Model-View-Control

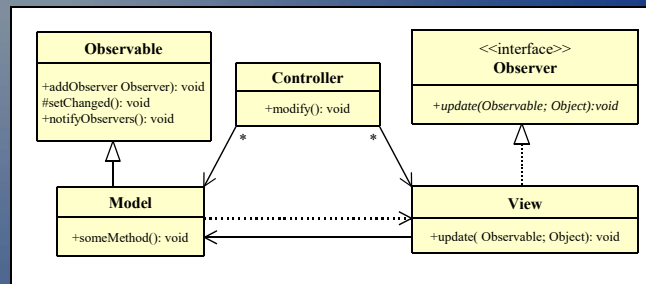
Genom att separera modellen och vyn kan förändringar göras i vyn och flera vyer skapas utan att behöva ändra koden i den underliggande modellen.



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

44

Model-View-Control



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

45

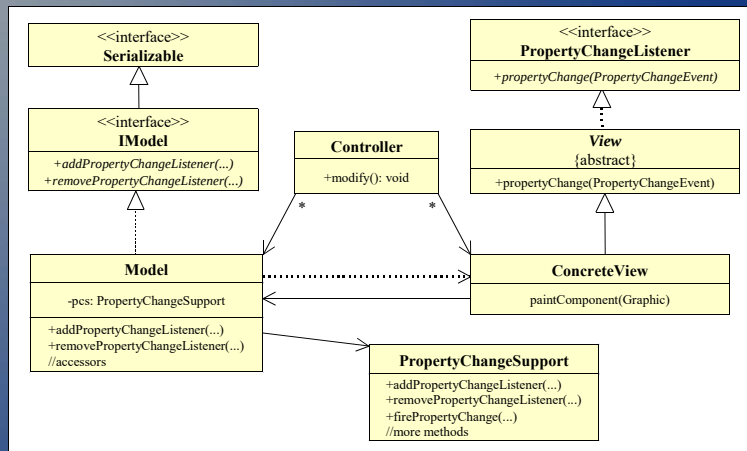
MVC och Java

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

- klassen `PropertyChangeSupport` och
- interfacet `PropertyChangeListener` istället för `Observable` och `Observer`.

46

MVC och Java



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

47

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 händelsen verkligen skall genereras.

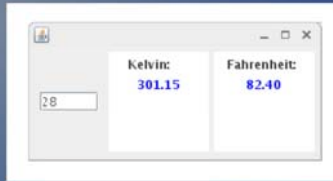
I interfacet `PropertyChangeListener` finns endast en metod specificerad:

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

48

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.



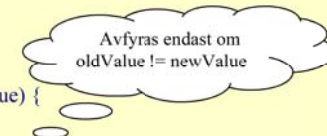
IModell:

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

49

Modell:

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



50

View:

```
import java.beans.*;
import javax.swing.*;
public abstract class View extends JPanel implements PropertyChangeListener {
    public View(Model m) {
        m.addPropertyChangeListener(this);
    }
    public abstract void propertyChange(PropertyChangeEvent e);
}
```

51

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);
    }
    public void propertyChange(PropertyChangeEvent e) {
        double toK = ((Double)e.getNewValue())+273.15;
        k.setText(String.format("%.2f", toK));
    }
}
```

52

Konkreta vyer – FahrenheitView:

```
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);
    }
    public void propertyChange(PropertyChangeEvent e) {
        double toF = ((Double)e.getNewValue())*9.0/5.0+32;
        f.setText(String.format("%.2f", toF));
    }
}
```

53

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);
    }
    public void actionPerformed(ActionEvent e) {
        model.setValue(Double.parseDouble(f.getText()));
    }
}
```

54

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);
    }
    public static void main(String[] arg){
        MVCDemo demo = new MVCDemo();
    }
}
```

55