Föreläsning 9

Designmönster

Template Method
Strategy
Gränssnittet Comparator
Singleton
State

Refaktorering

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

- eventuella fel behöver bara lokaliseras och åtgärdas på en plats (annars behövs upprepade ändringar)

Dubblerad kod i samma klass

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

Refactoring: Extrahera metod

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


Dubblerad kod i olika klasser

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

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

Refactoring: Implementationsarv

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

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

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

Dubblerad kod i olika klasser

Refactoring: Delegering till objekt

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

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

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

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

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


Dubblerad kod i olika klasser

Refactoring: Delegering till klassmetod

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

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

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

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

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


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 SinePlotter extends JPanel{
  private int xorigin;
  private int yorigin;
  private int xratio = 50;
  private int yratio = 50;
  public SinePlotter() {
    setBackground(Color.WHITE);
  }
```

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

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

```
  private 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);
    }
  }
}
```


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:

```
private 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);
  }
}
```

Metoden plotFunction för cosinus-kurva:

```
private 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);
  }
}
```

Dubblerad kod mixad med omgivningsberoende kod

Varför skall vi inte skapa en plotter som som ritar ut en cosinus-kurva genom *klipp-och-klistra*?

```
import  java.awt.*;
import  javax.swing.*;
public class CosinePlotter extends JPanel {
   private int xorigin;
   private int yorigin;
   private int xratio = 50;
   private int yratio = 50;
   public CosinePlotter() {
     setBackground(Color.WHITE);
   }
```


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

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

```
   private 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);
      }
   }
}
```


Vad är ett designmönster?

Ett designmönster beskriver en *beprövad lösning* på ett generellt återkommande problem

- förmedlar kunskap mellan experter och noviser (lära sej av bra design – inte av sina misstag)
- ger en vokabulär för att kommunicera (möjliggör att diskutera lösningar på en abstraktare nivå).

Designmönster finns inom många ämnesdiscipliner. Introducerades först inom arkitektur av Christopher Alexander.

Inom objektorienterad programutveckling lanserades design-mönster av i mitten av 1990-talet av Gamma, Helm, Johnson och Vlissides, vilka kallas för ”Gang of Four” (GoF).

Vad är ett designmönster?

Inom objektorienterad programutveckling är designmönster en *teknik* för att åstadkomma kod som är mer flexibel.

Utgår från designprinciperna:

- *Depend on abstractions, not on concrete implementations (DIP)*
- *Favor composition over inheritance*
- *Encapsulate what varies*.

Designmönster syftar till att bringa ordning och överblickbarhet

- reducerar antalet beroenden mellan komponenterna i systemet.

Beskrivning av designmönster

- Mönstrets namn
- Problemet
  - beskrivning av problemet och dess kontext.
- Lösningen
  - hur mönstret löser problemet i kontexten.
- Syfte
  - meningen med mönstret.
- Deltagare
  - de olika elementen som behövs.
- Konsekvenser
  - fördelar och nackdelar med att tillämpa mönstret.

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 skjuta upp vissa steg till subklasserna. Låt subklasserna definiera dessa steg.

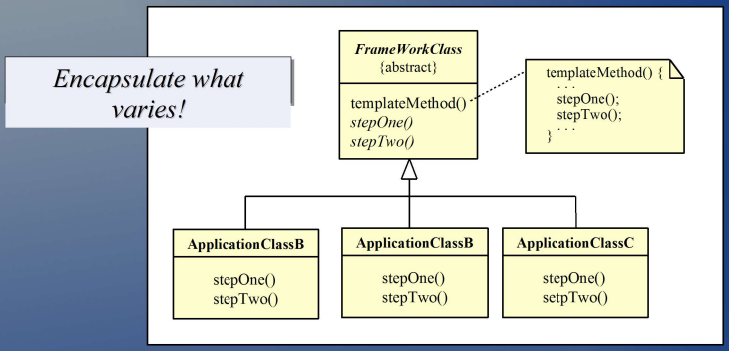
Enheter: En superklass som implementerar skal (*template*) för en eller flera metoder, vilka nyttjar abstrakta metoder, s.k. *hook*-metoder.

Konkreta klasser implementerar *hook*-metoderna.

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

Designmönstret Template Method

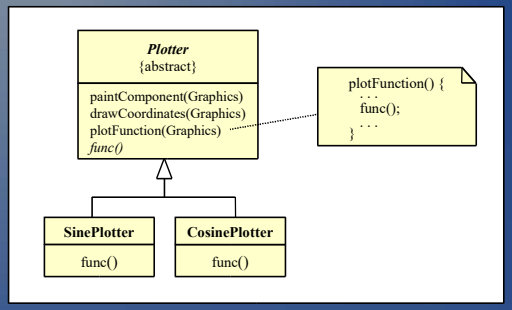
Encapsulate what varies!



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

Refaktorering med Template Method

Design av SinePlotter och CosinePlotter med användning av designmönstret Template Method:



Enkelt att skapa kurvritare för andra funktioner.
Stöder *The Open-Closed Principle*.

Klassen Plotter

```
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);
   }
   public void paintComponent(Graphics g) {
     //som tidigare
   }
   private void drawCoordinateAxes(Graphics g) {
      //som tidigare
   }
```

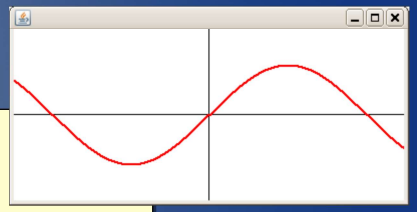
```
//template method
   private void plotFunction(Graphics g) {
      g.setColor(Color.RED);
      xorigin = getWidth()/2;
      yorigin = getHeight()/2;
      for (int px = 0; px < getWidth(); px++)  {
         double x = (px - xorigin) / (double) xratio;
         double y = func(x);
         int py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

Plotter för sinus-funktionen

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

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

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



b) samt en main-metod :

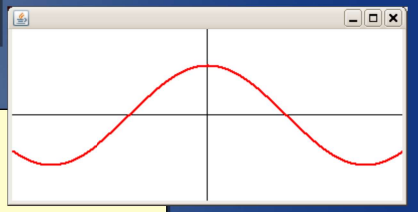
```
import javax.swing.*;
public class Main {
   public static void main(String[] args) {
      JFrame w = new JFrame();
      Plotter ps = new SinePlotter();
      w.add(ps);
      w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      w.setSize(400,200);
      w.setVisible(true);
   }
}
```


Plotter för cosinus-funktionen

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

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

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

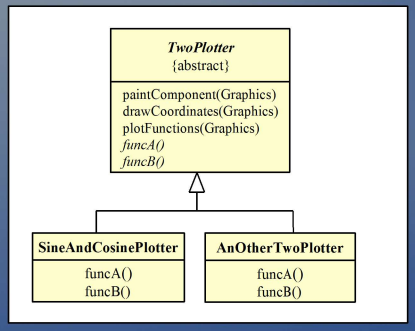


b) samt en main-metod :

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


Hur rita flera funktioner med samma plotter?

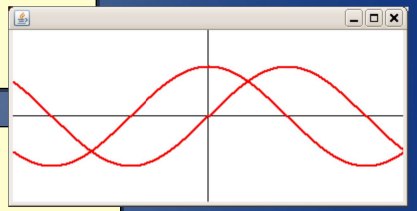
Om vi vill rita ut två funktioner samtidig med samma plotter måste vi skapa en plotter med två hook-metoder:




```
public abstract class TwoPlotter extends JPanel{
   . . .
   // template-method
   private void plotFunctions(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);
      }
   }
   // hook-methods
   protected abstract double funcA(double x);
   protected abstract double funcB(double x);
}
```


Hur rita flera funktioner med samma plotter?

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



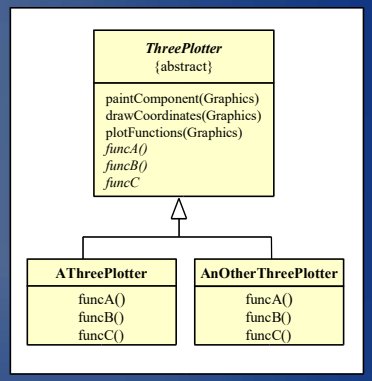
```
import javax.swing.*;
public class Main {
   public static void main(String[] args) {
      JFrame w = new JFrame();
      TwoPlotter ps = new SineCosinePlotter();
      w.add(ps);
      w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      w.setSize(400,200);
      w.setVisible(true);
   }
}
```

Hur rita flera funktioner med samma plotter?

Om vi vill rita tre funktioner samtidigt måste de konkreta klasserna implementera tre hook-metoder.

Strider mot *Open-Closed Principle*.

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.

Användbarhet:

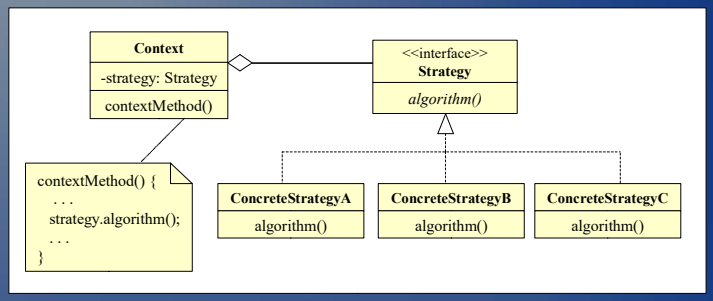
- när flera relaterade klasser skiljer sig endast i beteende, inte i hur de används
- när algoritmerna som styr dessa beteenden använder information som klienten inte behöver känna till
- när olika varianter av en algoritm behövs.

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

Konkreta klasser som implementerar olika strategier.

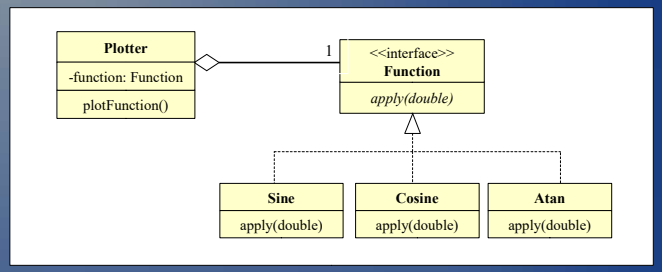
Klassen Context som handhar referenser till strategiobjekten.

Designmönstret Strategy



Plotter implementerad med designmönstret Strategy

Design av Plotter med användning av designmönstret Strategy:



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, vilket tillhandahåller funktionen.

Plotter implementerad med designmönstret Strategy

Context: Plotter

```
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;
   private Function function;
   public Plotter(Function function) {
      this.function = function;
      setBackground(Color.WHITE);
   }
```

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

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

```
   private 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 = function.apply(x);
         int py = yorigin - (int) (y * yratio);
         g.fillOval(px - 1, py - 1, 3, 3);
      }
   }
}
```


Plotter implementerad med designmönstret Strategy

Stategy-interface:

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

Konkreta strategier:

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

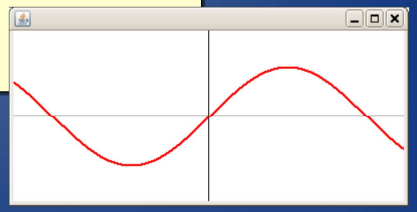
```
public class Cosine implements Function {
   public double apply(double x) {
      return Math.cos(x);
   }
}
```

```
public class Atan implements Function {
   public double apply(double x) {
      return Math.atan(x);
   }
}
```


Plotter implementerad med designmönstret Strategy

Plotter som ritar ut en sinus-kurva:

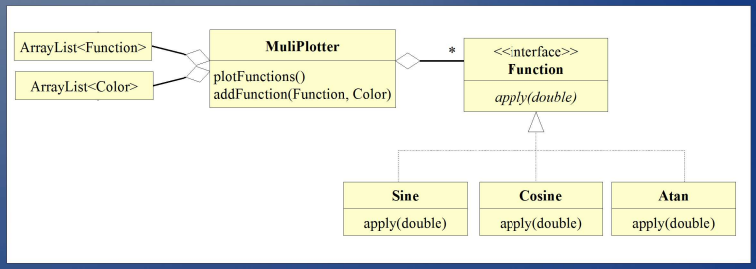
```
import javax.swing.*;
public class MainPlotSine {
   public static void main(String[] args) {
      JFrame w = new JFrame();
      Plotter ps = new Plotter(new Sine());
      w.add(ps);
      w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      w.setSize(400,200);
      w.setVisible(true);
  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



MultiPlotter med användning av Strategy-mönstret

En MultiPlotter, för att rita ut ett godtyckligt antal funktioner, kan implementeras med en dynamisk lista av objekt av typen Function. I metoden plotFunctions genomlöps listan och genom att nyttja dynamisk bindning anropas apply-metoden på respektive objekt (t.ex. Sine Cosine eller Atan).

För att göra plottningen tydligare associeras varje funktion med en färg (ett objekt av typen Color).



Klassen MultiPlotter

```
import java.awt.*;
import javax.swing.*;
import java.util.*;
public class MultiPlotter extends JPanel{
   private int xorigin;
   private int yorigin;
   private int xratio = 50;
   private int yratio = 50;
   private ArrayList<Function> functions = new ArrayList<Function>();
   private ArrayList<Color> colors = new ArrayList<Color>();
   public MultiPlotter() {
      setBackground(Color.WHITE);
   }
```

```
   protected void addFunction(Function f, Color c) {
      functions.add(f);
      colors.add(c);
   }
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      drawCoordinates(g);
      plotFunctions(g);
   }
```


Klassen MultiPlotter

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


MultiPlotter som ritar tre funktioner

```
import javax.swing.*;
import java.awt.*;
public class MainMultiPlotter {
   public static void main(String[] args) {
      MultiPlotter mp = new MultiPlotter();
      mp.addFunction(new Sine(), Color.GREEN);
      mp.addFunction(new Cosine(), Color.BLUE);
      mp.addFunction(new Atan(), Color.RED);
      JFrame w = new JFrame();
      w.add(mp);
      w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      w.setSize(400,200);
      w.setVisible(true);
   }
}
```

Gränssnitten Comparable<T>

För att lokalisera ett specifikt objekt i en samling, eller för att sortera en samling av objekt, är det nödvändigt att kunna jämföra två objekt på storlek.

Java tillhandahåller interfacet Comparable<T> för att definiera en ordning på två objekt av typen T:

```
public interface Comparable <T> {
   public int compareTo(T o);
}
```

Ett anrop

```
a.compareTo(b)
```

skall returnera värdet 0 om a har samma storlek som b, returnera ett negativt tal om a är mindre än b och returnera ett positivt tal om a är större än b.

Gränssnitten Comparable<T>

Om en klass implementerar interfacet Comparable<T> blir objekten i klassen jämförbara på storlek.

```
public class Person implements Comparable <Person>{
   private String name;
   private int id;
   public Person(String name, int id) {
      this.name = name;
      this.id = id;
   }
   public int getId() {
      return id;
   }
   @Override
   public String toString() {
      return ("Namn: " + name + "   IDnr: " + id);
   }
   @Override
   public int compareTo(Person otherPerson) {
      return name.compareTo(otherPerson.name);
   }
}
```

Ordningen som definieras av metoden compareTo kallas för den *naturliga ordningen*.

Gränssnitten Comparable<T>

För att sortera fält respektive listor tillhandahåller Java klassmetoderna Arrays.sort(T[]) och Collections.sort(List<T>).

Nedan ges ett program som lagrar ett antal objekt av klassen Person i ett fält. Fältet sorteras och skrivs i den *naturliga ordningen* som definieras av metoden compareTo, d.v.s. i bokstavsordning med avseende på namnen.

```
import java.util.*;
public class PersonTest1 {
   public static void main(String[] arg) {
      Person[] persons = {new Person("Kalle", 22), new Person("Lisa", 62),
                          new Person("Emilia", 42), new Person("Rune", 12)};
      Arrays.sort(persons);
      System.out.println("I bokstavordning:");
      for (int i = 0; i < persons.length; i++ )
         System.out.println(persons[i]);
   }
}
```

Utskriften av programmet blir:

```
I bokstavordning:
Namn: Emilia   IDnr: 42
Namn: Kalle   IDnr: 22
Namn: Lisa   IDnr: 62
Namn: Rune   IDnr: 12
```


Gränssnitten Comparator<T>

Fråga: Men hur gör vi om vi vill sortera på något annat sätt än i bokstavsordning med avseende på namnen, t.ex. i växande ordning på id?

Svar: Vi använder designmönstret Strategy!

Java tillhandahåller (Stratey-interfacet) Comparator<T>:

```
public interface Comparator<T> {
   public int compare(T o1, T o2);
   public boolean equals(Object obj); *)
}
```

samt klassmetoderna Arrays.sort(T[], Comparator<T>) och Collections.sort(List<T>,Comparator<T>).

Genom att implementera interfacet Comparator<T> kan vi definera den sorteringsstrategi vi önskar.

*) Metoden equals används för att undersöka om två objekt av Comparator är lika och behöver normalt inte överskuggas.

Exempel: Comparator<T>

Antag att vi vill jämföra objekt av klassen Person med avseende på växande ordning på komponenten id.

Vi måste då införa en konkret strategy-klass, vilken vi kallar IDComparator, som implementerar gränssnittet Comparator:

```
import java.util.*;
public class IDComparator implements Comparator <Person> {
   @Override
   public int compare(Person a, Person b) {
      if (a.getId() < b.getId())
         return -1;
      if (a.getId() == b.getId())
         return 0;
      return 1;
   }
}
```

Fortsättning på exempel:

Nedan ges ett program som lagrar ett antal objekt av klassen Person i ett fält och objekten skrivs ut i växande ordning med avseende på id-numret.

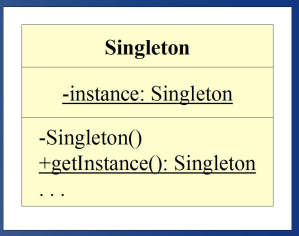
```
import java.util.*;
public class PersonTest2 {
   public static void main(String[] arg) {
      List<Person> list = new ArrayList<Person>();
      list.add(new Person("Stina ", 76));
      list.add(new Person("Agneta", 32));
      list.add(new Person("Karin ", 15));
      list.add(new Person("Lisa  ", 98));
      list.add(new Person("Berit ", 52));
      Collections.sort(list, new IDComparator());
      System.out.println("I id-ordning:");
      for (Person p : list)
         System.out.println(p);
   }
}
```

Utskriften av programmet blir:

```
I id-ordning:
Namn: Karin   IDnr: 15
Namn: Agneta IDnr: 32
Namn: Berit    IDnr: 52
Namn: Stina    IDnr: 76
Namn: Lisa     IDnr: 98
```


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 klassmetoden 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;
   }
   // Other useful methods here
}
```

Statiska attribut i en klass initieras när Javas Viritual Machine laddar in klassen. Eftersom 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 kan skjutas upp tills metoden getInstance anropas för första gången. Detta innebär att denna metod måste göras trådsäker (konstruktionen **synchronized**).

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

Designmönstret Singleton

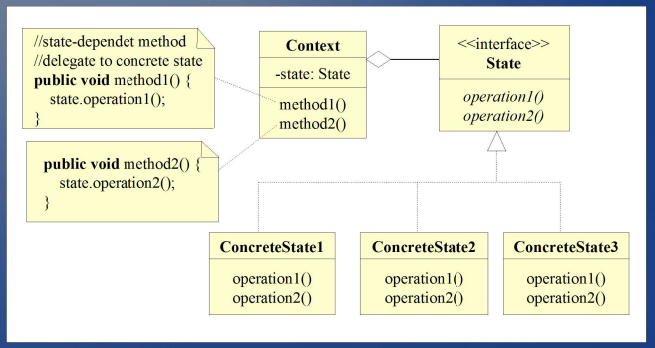
Exempel: En singeltonklass NumberGenerator för att generera löpnummer.

```
public class NumberGenerator {
  private static NumberGenerator instance;
  private int theNumber;
  private NumberGenerator() {
    theNumber = 0;
  }
  public static synchronized NumberGenerator getInstance() {
    if (instance == null) {
      instance = new NumberGenerator();
    }
    return instance;
  }
  public synchronized int nextNumber() {
    theNumber++;
    return theNumber;
  }
}
```


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.

Designmönstret State har samma strukturell uppbyggnad som designmönstret Strategy, men har ett annat syfte.

Exempel på State-mönstret

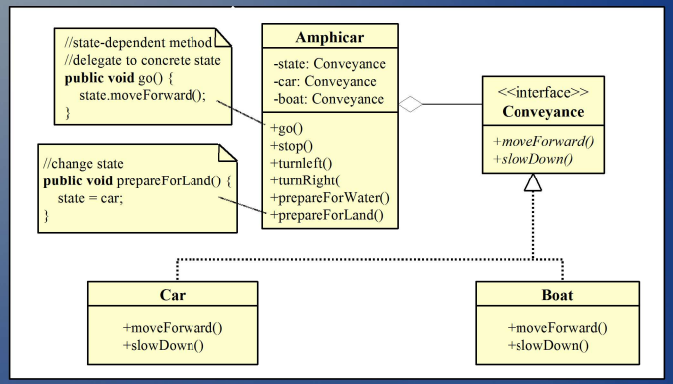
Antag att vi har en amfibiebil.

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



Interfacet Conveyance och klasserna Car och Boat

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

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

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


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


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();
   }
}
```

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

Fler designmönster i nästa föreläsning!