Informatik IIb

Objektorientierte Programmierung mit Java

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Wintersemester 2010/11

Vorlesung 5. Interfaces

Lernziele dieser Vorlesung

Interfaces allgemein

- Delegation

Interfaces speziell

- Comparable
- Mehrfachimplementierung
- Vererbung von Interfaces
- Ableiten von Interfaces

Gegenüberstellung von Vererbung und Delegation

Zusammenfassung

Lernziele

- Kenntnis des Vererbungsprinzips
- Kenntnis des Delegationsprinzips
- Fähigkeit der Bewertung von Software-Entwurf bzgl. Vererbung und Delegation

Das Problem

- „Guter“ Software-Entwurf.
- Hier: Realisierung des Dependency Inversion Principle:
  - „High level modules should not depend upon low level modules. Both should depend upon abstractions.“
  - „Abstractions should not depend upon details. Details should depend upon abstractions.“
- Erforderlich ist die Trennung von generischem Algorithmus und konkretem Kontext.
- Ziel ist die Wiederverwendung von Modulen hoher Ebene.

Entwurfsmuster

- Klassenvererbung
  - Der abstrakte Algorithmus wird in einer Basisklasse implementiert.
  - Der konkrete Kontext wird in einer abgeleiteten Klasse implementiert.
- Schnittstellenvererbung: Delegation
  - Der abstrakte Algorithmus wird in einer konkreten Klasse implementiert
  - Die konkrete Klasse enthält abstrakte Methoden, die der abstrakte Algorithmus über eine Schnittstelle aufruft.
  - Der Konkrete Kontext wird in einer abgeleiteten Schnittstelle implementiert.

Delegation

- In Java gibt es keine Mehrfachvererbung.
- Klassen können aber mehrere Schnittstellendefinitionen erben.
- Das Konstrukt für Schnittstellendefinitionen in Java heißt interface.

Definition eines Interfaces

- Ein Interface enthält ausschließlich abstrakte Methoden und Konstanten.
- Anstelle des Schlüsselwortes class wird ein Interface mit dem Bezeichner interface deklariert.
- Alle Methoden eines Interfaces sind implizit abstrakt und öffentlich.
- Neben Methoden kann ein Interface auch Konstanten enthalten, die Definition von Konstruktoren ist allerdings nicht erlaubt.

Ein einfaches Interface

Interface Groesse, das die drei Methoden laenge, hoehe und breite enthält

```
public interface Groesse
{
  public int laenge();
  public int hoehe();
  public int breite();
}
```

- Schnittstelle für den Zugriff auf die räumliche Ausdehnung eines Objekts
- Keine Details, etwa Maßeinheiten oder Objekte mit mehr oder weniger als drei Dimensionen

Implementierung eines Interfaces

- Definition eines Interfaces stellt keine Funktionalität zur Verfügung.
- Realisierung der Funktionalität erfordert die Implementierung des Interfaces in einer Klasse.
- Dazu erweitert die Klasse die class-Anweisung um eine implements-Klausel, hinter der der Name des zu implementierenden Interfaces angegeben wird. Der Compiler sorgt dafür, dass alle im Interface geforderten Methoden definitionsgemäß implementiert werden.
- Der Compiler verleiht der Klasse einen neuen Datentyp, der ähnliche Eigenschaften wie eine echte Klasse hat.

Implementierung 1 des Interfaces

```
public class Auto
implements Groesse
{
  public String name;
  public int    erstzulassung, leistung;
  public int    laenge, hoehe, breite;
  public int laenge()
  {
    return this.laenge;
  }
  public int hoehe()
  {
    return this.hoehe;
  }
  public int breite()
  {
    return this.breite;
  }
}
```

Implementierung 2 des Interfaces

```
public class FussballPlatz
implements Groesse
{
  public int laenge()
  {
    return 105000;
  }
  public int hoehe()
  {
    return 0;
  }
  public int breite()
  {
    return 70000;
  }
}
```

Implementierung 3 des Interfaces

```
public class PapierBlatt
implements Groesse
{
  public int format; //0=DIN A0, 1=DIN A1 usw.
  public int laenge()
  {
    int ret = 0;
    if (format == 0) {
      ret = 1189;
    } else if (format == 1) {
      ret = 841;
    } else if (format == 2) {
      ret = 594;
    } else if (format == 3) {
      ret = 420;
    } //usw...
    return ret;
  }
```

Implementierung 3 des Interfaces (Forts.)

```
  public int hoehe()
  {
    return 0;
  }
  public int breite()
  {
    int ret = 0;
    if (format == 0) {
      ret = 841;
    } else if (format == 1) {
      ret = 594;
    } else if (format == 2) {
      ret = 420;
    } else if (format == 3) {
      ret = 297;
    } //usw...
    return ret;
  }
}
```

Implementierung von Interfaces

- Die Art der Realisierung der vereinbarten Methoden spielt für das Implementieren eines Interfaces keine Rolle.
- Standardfall ist, dass Interfaces von sehr unterschiedlichen Klassen implementiert und die erforderlichen Methoden auf sehr unterschiedliche Weise realisiert werden.
- Eine Klasse kann ein Interface auch teilweise implementieren, d. h. nicht alle seine Methoden implementieren. Dann muss die Klasse allerdings als abstract deklariert sein.

Verwenden eines Interfaces

- Nützlich ist ein Interface dann, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in seiner normalen Vererbungshierarchie abgebildet werden können.
- Beispiel Groesse-Interface: als abstrakte Klasse hätten dann Autos, Fußballplätze und Papierblätter daraus abgeleitet werden müssen.
- Interface vermeidet unnatürliche Verebungshierarchie.
- Durch Implementieren des Groesse-Interfaces können sie die Verfügbarkeit der drei Methoden laenge, hoehe und breite dagegen unabhängig von ihrer eigenen Vererbungslinie garantieren.

Beispiel Berechnung der Grundfläche

```
public class Flaeche
{
  public static long grundflaeche(Groesse g)
  {
    return (long)g.laenge() * g.breite();
  }
```

Beispiel Berechnung der Grundfläche (Forts.)

```
  public static void main(String[] args)
  {
    Auto auto = new Auto();
    auto.laenge = 4235;
    auto.hoehe = 1650;
    auto.breite = 1820;

    PapierBlatt blatt = new PapierBlatt();
    blatt.format = 4;

    FussballPlatz platz = new FussballPlatz();

    System.out.println("Auto:  "+grundflaeche(auto));
    System.out.println("Blatt: "+grundflaeche(blatt));
    System.out.println("Platz: "+grundflaeche(platz));
  }
}
```

Interface Comparable

- Interfaces werden verwendet, um Eigenschaften auszudrücken, die auf Klassen aus unterschiedlichen Klassenhierarchien zutreffen können.
- Beispiel: Interface Comparable des Pakets java.lang:

```
public interface Comparable
{
  public int compareTo(Object o);
}
```

- Implementierung in Klassen, deren Objekte paarweise vergleichbar sind.
- Ergebnis von compareTo: kleiner 0, wenn „kleiner“, größer 0, wenn „größer“, 0, wenn „gleich“
- Z. B. für String und Character

Beispiel: Kleinstes Objekt finden, Objekte sortieren

```
public class CompTest
{
  public static Object getSmallest(Comparable[]
                                    objects)
  {
    Object smallest = objects[0];
    for (int i = 1; i < objects.length; ++i) {
      if (objects[i].compareTo(smallest) < 0) {
        smallest = objects[i];
      }
    }
    return smallest;
  }
```

Beispiel: Kleinstes Objekt finden, Objekte sortieren (Forts.)

```
public static void bubbleSort(Comparable[] objects)
{
  boolean sorted;
  do {
    sorted = true;
    for (int i = 0; i < objects.length - 1; ++i) {
      if (objects[i].compareTo(objects[i + 1])>0) {
        Comparable tmp = objects[i];
        objects[i] = objects[i + 1];
        objects[i + 1] = tmp;
        sorted = false;
      }
    }
  } while (!sorted);
}
```

Beispiel: Kleinstes Objekt finden, Objekte sortieren (Forts.)

```
  public static void main(String[] args)
  {
    //Erzeugen eines String-Arrays
    Comparable[] objects = new Comparable[4];
    objects[0] = "STRINGS";
    objects[1] = "SIND";
    objects[2] = "PAARWEISE";
    objects[3] = "VERGLEICHBAR";
    //Ausgeben des kleinsten Elements
    System.out.println((String)getSmallest(objects));
    System.out.println("--");
    //Sortieren und Ausgaben
    bubbleSort(objects);
    for (int i = 0; i < objects.length; ++i) {
      System.out.println((String)objects[i]);
    }
  }
}
```

Mehrfachimplementierung

- Es ist möglich (und gebräuchlich), dass eine Klasse mehrere Interfaces implementiert.
- Sie muss dann zu jedem Interface alle darin definierten Methoden implementieren.
- Mit jedem implementierten Interface wird sie zu dem dadurch definierten Datentyp kompatibel.
- Eine Klasse, die n Interfaces implementiert, ist demnach zu n + 1 Datentypen (plus ihren jeweiligen Oberklassen) kompatibel:
  - Der Vaterklasse, aus der sie abgeleitet wurde (bzw. der Klasse Object, falls keine extends-Klausel vorhanden war).
  - Den n Interfaces, die sie implementiert.

Beispiel: Auto

```
public class Auto2
implements Groesse, Comparable
{
  public String name;
  public int    erstzulassung;
  public int    leistung;
  public int    laenge;
  public int    hoehe;
  public int    breite;
```

Beispiel: Auto2 (Forts.)

```
public int laenge()
{
  return this.laenge;
}

public int hoehe()
{
  return this.hoehe;
}

public int breite()
{
  return this.breite;
}
```

Beispiel: Auto2 (Forts.)

```
  public int compareTo(Object o)
  {
    int ret = 0;
    if (leistung < ((Auto2)o).leistung) {
      ret = -1;
    } else if (leistung > ((Auto2)o).leistung) {
      ret = 1;
    }
    return ret;
  }
}
```

Vererbung von Interfaces

- Kombination von Vererbung und Implementierung von Interfaces.
- Beispiel: Auto2 kann zunächst von Klasse Auto erben und dann Comparable implementieren.

Beispel Auto2 mit Vererbung

```
public class Auto2
extends Auto
implements Comparable
{
  public int compareTo(Object o)
  {
    int ret = 0;
    if (leistung < ((Auto2)o).leistung) {
      ret = -1;
    } else if (leistung > ((Auto2)o).leistung) {
      ret = 1;
    }
    return ret;
  }
}
```

Ableiten von Interfaces

- Interfaces können abgeleitet werden.
- Ähnlich einer Klasse erbt das abgeleitete Interface alle Methodendefinitionen des Basis-Interfaces.
- Die implementierende Klasse muss alle Methoden von allen übergeordneten Interfaces implementieren.

Beispiel: Dimension-Interfaces abgeleitet

```
interface EinDimensional
{
  public int laenge();
}

interface ZweiDimensional
extends EinDimensional
{
  public int breite();
}

interface DreiDimensional
extends ZweiDimensional
{
  public int hoehe();
}
```

Beispiel: Dimension-Interfaces abgeleitet (Forts.)

```
interface VierDimensional
extends DreiDimensional
{
  public int lebensdauer();
}

public class TestDim
implements VierDimensional
{
  public int laenge() { return 0; }
  public int breite() { return 0; }
  public int hoehe() { return 0; }
  public int lebensdauer() { return 0; }
}
```

Beispiel Sortieren

```
// BubbleSorter.java
public class BubbleSorter {
  static int operations = 0;
  public static int sort(int [] array) { // Algorithmus
    operations = 0;
    if (array.length <= 1)
      return operations;
    for (int nextToLast = array.length-2;
         nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
        compareAndSwap(array, index);
    return operations;
  }
```

Beispiel Sortieren (Forts.)

```
  // Details
  private static void swap(int[] array,
                           int index) {
     int temp = array[index];
     array[index] = array[index+1];
     array[index+1] = temp;
   }

   // Details
   private static void compareAndSwap(int[] array,
                                      int index) {
     if (array[index] > array[index+1])
       swap(array, index);
     operations++;
   }
}
```

Beispiel Sortieren (Forts.)

- Die Klasse `BubbleSorter` implementiert die Sortierung eines Feldes von ganzen Zahlen mit dem Bubble-Sort-Algorithmus.
- Die Methode `sort` enthält den Algorithmus.
- Die Methoden `swap` und `compareAndSwap` enthalten die Details der ganzen Zahlen und Felder.
- `swap` und `compareAndSwap` implementieren die spezifischen Operationen für ganze Zahlen und Felder, die der `sort`-Algorithmus erfordert.

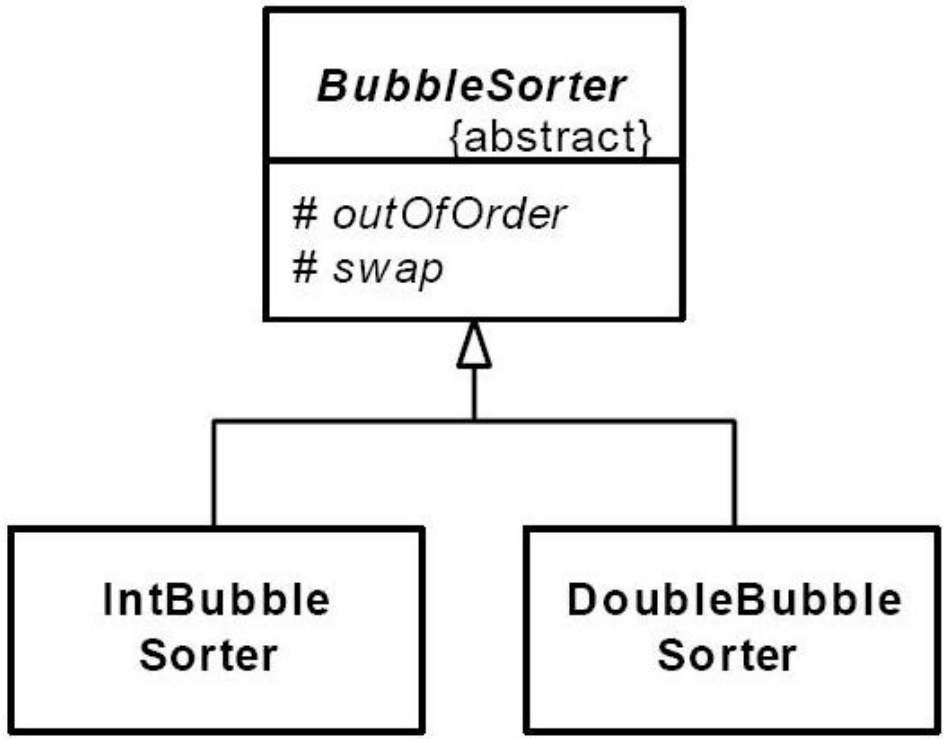
Trennung von Algorithmus und Details durch Klassenvererbung

- Abstrakte Basisklasse `BubbleSorter` enthält den Bubble-Sort-Algorithmus ohne Details über Datentypen.
- Die Funktion `doSort` verwendet die abstrakten Methoden
  - `outOfOrder`, die zwei benachbarte Elemente des Feldes bezüglich deren Ordnung vergleicht,
  - `swap`, die zwei benachbarte Elemente des Feldes vertauscht.
- `doSort` kennt weder ein Feld noch den Datentyp der Elemente.
- Implementierung von Datentyp-spezifischen abgeleiteten Klassen hängt von der Basisklasse ab.

Klassenvererbung: Basisklasse

```
// BubbleSorter.java
public abstract class BubbleSorter {
  private int operations = 0;
  protected int length = 0;
  protected int doSort() {
    operations = 0;
    if (length <= 1)
      return operations;
    for (int nextToLast = length-2;
      nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++) {
        if (outOfOrder(index))
          swap(index);
        operations++;
      }
    return operations;
  }
  protected abstract void swap(int index);
  protected abstract boolean outOfOrder(int index);
}
```

Vererbung: abgeleitete Klassen



- Abgeleitete Klassen für verschiedene Datentypen
- Implementierung von datentypen-spezifischen Details

Vererbung: abgeleitete Klasse, ganze Zahlen

```
// IntBubbleSorter.java
public class IntBubbleSorter extends BubbleSorter {
  private int[] array = null;
  public int sort(int [] theArray) {
    array = theArray;
    length = array.length;
    return doSort();
  }
  protected void swap(int index) {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }
  protected boolean outOfOrder(int index) {
    return (array[index] > array[index+1]);
  }
}
```

Trennung von Algorithmus und Details durch Delegation

- Konkrete Klasse enthält den Bubble-Sort-Algorithmus ohne Details über Datentypen.
- Die Funktion `sort` verwendet abstrakte Methoden, die in einer Schnittstelle definiert sind.
- `sort` kennt keine konkreten Datentypen
- Die Interface-Implementierung hängt nicht von der Bubble-Sort-Klasse ab.

Delegation: Klasse mit Interface

```
// BubbleSorter.java
public class BubbleSorter {
  private int operations = 0;
  private int length = 0;
  // Interface
  private SortHandle itsSortHandle = null;
  public BubbleSorter(SortHandle handle) {
    itsSortHandle = handle;
  }
```

Delegation: Klasse mit Interface

```
  public int sort(Object array) {
    itsSortHandle.setArray(array); // Interface
    length = itsSortHandle.length(); // Interface
    operations = 0;
    if (length <= 1)
      return operations;
    for (int nextToLast = length-2;
      nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
      {
      if (itsSortHandle.outOfOrder(index)) // Interface
        itsSortHandle.swap(index); // Interface
      operations++;
      }
    return operations;
  }
}
```

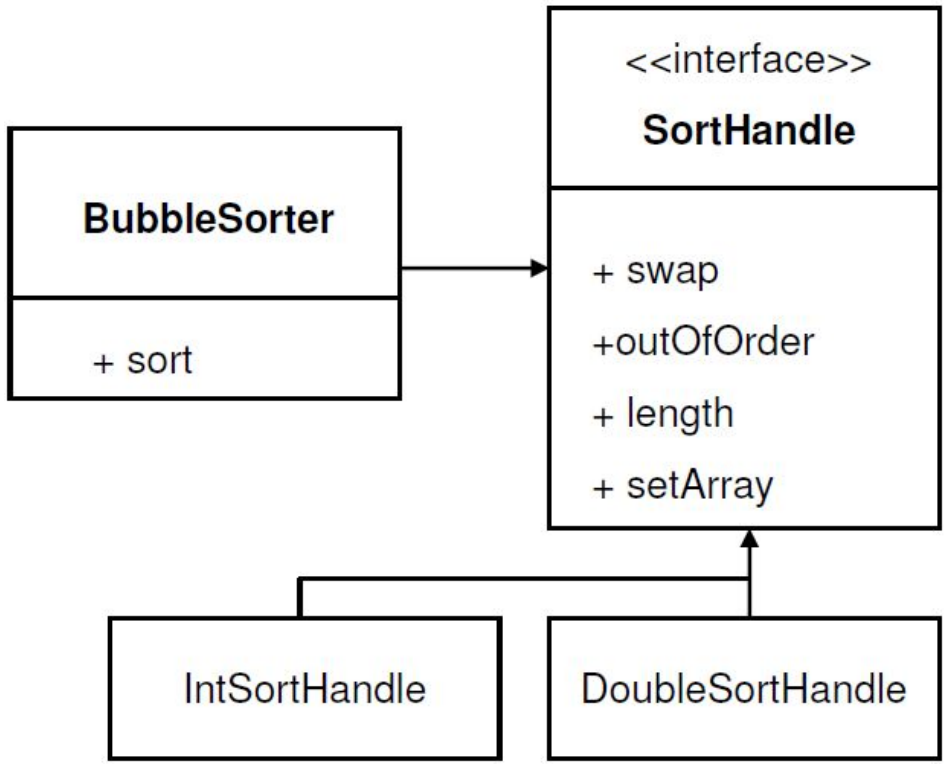
Delegation: Klasse mit Interface (Forts.)

```
// SortHandle.java
public interface SortHandle {
  public void swap(int index);
  public boolean outOfOrder(int index);
  public int length();
  public void setArray(Object array);
}
```

Abgeleitete Klasse, Implementierung Schnittstelle für ganze Zahlen

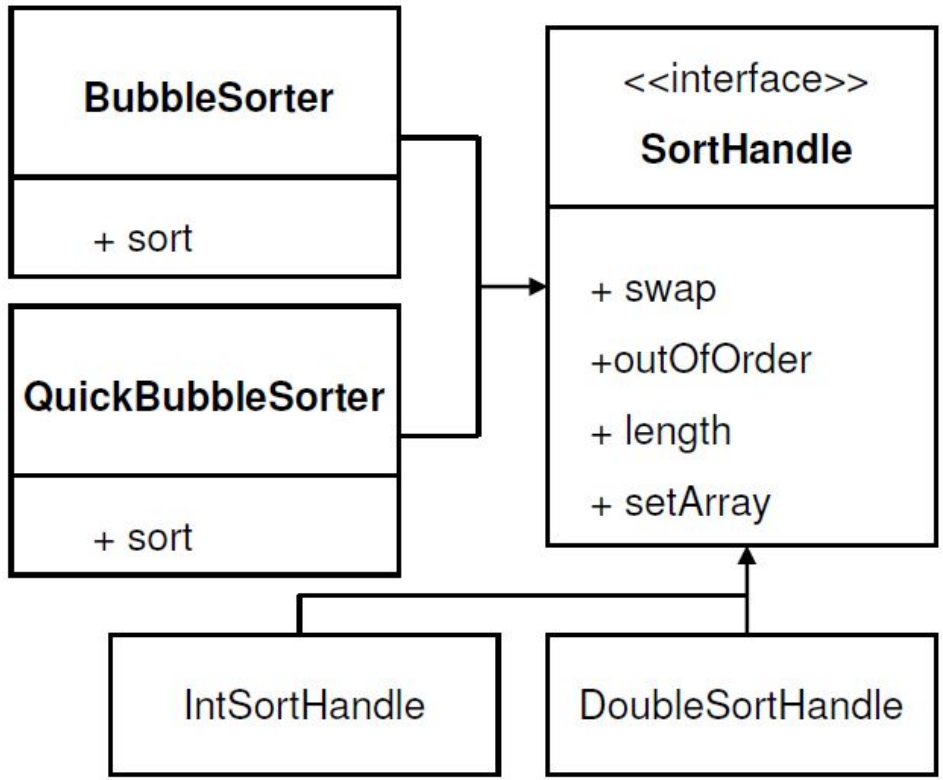
```
public class IntSortHandle implements SortHandle {
  private int[] array = null;
  public void swap(int index) {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }
  public void setArray(Object array) {
    this.array = (int[])array;
  }
  public int length() {
    return array.length;
  }
  public boolean outOfOrder(int index) {
    return (array[index] > array[index+1]);
  }
}
```

Delegation: Interface



- Algorithmus in konkreter Klasse
- Abstrakte Methoden in der Schnittstelle
- Schnittstellen-Implementierung für Datentypen-spezifische Details

Delegation: anderer Sortier-Algorithmus



- Detail-Implementierung der Schnittstelle hängt nicht von der Sortier-Klasse ab.
- Detail-Implementierung ist wiederverwendbar.

Delegation: Austausch des Algorithmus'

```
// QuickBubbleSorter.java
public class QuickBubbleSorter {
  private int operations = 0;
  private int length = 0;
  private SortHandle itsSortHandle = null;
  public QuickBubbleSorter(SortHandle handle) {
   itsSortHandle = handle;
  }
```

Delegation: Austausch des Algorithmus'

```
  public int sort(Object array) {
    itsSortHandle.setArray(array);
    length = itsSortHandle.length();
    operations = 0;
    if (length <= 1) return operations;
    boolean thisPassInOrder = false;
    for (int nextToLast = length-2;
          nextToLast >= 0 && thisPassInOrder; nextToLast--) {
      thisPassInOrder = true; //potenially.
      for (int index = 0; index <= nextToLast; index++)
      {
        if (itsSortHandle.outOfOrder(index)) {
          itsSortHandle.swap(index); thisPassInOrder = false;
        }
        operations++;
      }
    }
    return operations;
  }
}
```

Zusammenfassung: Vererbung oder Delegation?

- Vererbung und Delegation folgen dem Dependency Inversion Principle.
- Vererbung sollte zum Modellieren von Spezialisierungshierarchien verwendet werden.
- Viele Detail-Implementierungen können vom generischen Algorithmus bearbeitet werden.
- Delegation über Schnittstellen sollte für lose gekoppelte Softwaresysteme verwendet werden.
- Zusätzlich können viele Detail-Implementierungen von vielen generischen Algorithmen bearbeitet werden.