

## 2. Einsatz von Vererbung, Übersicht

Vererbung ist ein **Konzept und Mechanismus** in objektorientierten Programmiersprachen (Abschnitt 2.1)

Sie wird eingesetzt für unterschiedliche **Zwecke des konzeptionellen Entwurfs** und für unterschiedliche **Zwecke der Programmentwicklung**.

Welchem Zweck eine Vererbungsrelation dient, ist **schwierig am Programmtext zu erkennen**.

### Einsatzzweck Entwurfskonzepte:

- 2.2 Abstraktion
- 2.3 Spezialisierung
- 2.4 Rollen, Eigenschaften
- 2.5 Spezifikation

### Einsatzzweck Programmentwicklung:

- 2.6 Inkrementelle Weiterentwicklung
- 2.7 Komposition statt Vererbung
- 2.8 Weitere Techniken

siehe auch [Antero Taivalsaari: On the Notion of Inheritance, ACM Computing Surveys, Vol. 28, No. 3, September 1996]

## 2.1 Mechanismen und Konzepte Mechanismen der Sprache

- **Unterklasse S** zur Oberklasse A bilden: **subclassing**
- transitive Vererbungsrelation bildet **Klassenhierarchie**
- Unterklasse kann **Methoden und Datenattribute** ihrer Oberklassen **erben**
- Unterklasse kann **zusätzliche Methoden und Datenattribute definieren**
- Unterklasse kann **Methoden** ihrer Oberklassen **überschreiben**
  
- **Schnittstelle für Methoden spezifizieren**: abstrakte Methode
- **Methodenschnittstelle** durch konkrete Methode **implementieren**
  
- **Schnittstelle für Klassen spezifizieren**: Interface
- **Klasse implementiert** die für sie spezifizierten **Schnittstellen**
  
- **polymorphe Variable** haben statischen Typ T und können Objektreferenzen anderer Typen S aufnehmen.  
S ist Unterklasse von T oder S implementiert die Schnittstelle T

## Konzeptionelle Begriffe

**Vererbung** bewirkt gleichzeitig **Erweiterung** der Funktionalität und **Einschränkung** der Verwendbarkeit

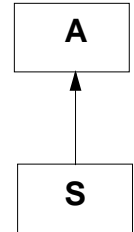
```
class PinBallGame extends Frame { ... }
```

**Überschreiben** einer Methode mit einer anderen kann die Funktionalität **verändern**.

Erschwert es, die konzeptionelle Bedeutung der Methode zu verstehen.  
Missbrauch ist möglich.

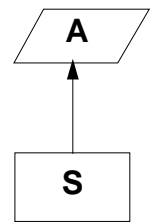
**Ersetzbarkeit (substitutability):**

Objekte einer Klasse **s** sind in Kontexten verwendbar, wo Objekte der Klasse **A** benötigt werden.  
Abweichungen vom Verhalten, das für **A** definiert ist, sind nicht beobachtbar.



**Untertypen (subtyping):**

wie Ersetzbarkeit, aber ausgedrückt durch Typen:  
Eine Variable vom Typ **A** kann **s**-Werte aufnehmen.  
Verhaltensabweichungen sind nicht beobachtbar.



Untertypeneigenschaft ist **technisch immer erfüllt** für  
Unterklasse **s** zur Oberklasse **A**, Klasse **s** zum Interface **A**

Untertypeneigenschaft ist **konzeptionell** in diesen Fällen **nicht immer erfüllt**,  
z. B. wenn die Definition von **s** eine Methode aus **A** durch Überschreiben „eliminiert“.

## Kriterien zur Anwendung von Vererbung

Nach [Coad, North, Mayfield], rigorose Kriterien **gegen freizügige Vererbung**:

1. **is-a Beziehung** muss gelten: Unterklasse **S** ist eine besondere Art der Oberklasse **A**  
führt zum **Paradigma Abstraktion** (siehe 2.2)  
Beispiel: Kaufvertrag, Mietvertrag, Darlehensvertrag sind **Arten von** Verträgen
2. Objekte **gehören der Unterklasse für immer an**, sie wechseln nicht in eine andere Klasse  
Gegenbeispiel: Kaulquappe wird erwachsener Frosch; führt zu Rollen (2.4) statt Unterklassen
3. **Unterklasse erweitert** die Fähigkeiten der Oberklasse, sie **entfernt keine** und **ändert keine**  
Gegenbeispiel: Stack durch Vektor implementiert; besser: Implementierung mit Komposition
4. Die Relation „A spielt eine **Rolle S**“ **nicht** durch Unterklasse **S** modellieren  
sondern durch Rollen (siehe 2.4)  
Beispiel: Person spielt Rolle Student, oder Rolle Assistent - oder beide!
5. **Nicht** Unterklasse **S** aus **A** bilden, nur weil die **Funktionalität von A gut nutzbar** ist  
Beispiel: Stack durch Vektor implementiert
6. **has-a Beziehung nicht** durch Unterklasse implementieren sondern durch **Komposition**  
Beispiel: Auto hat ein Antriebsaggregat
7. **Komposition** bewahrt die **Kapselung** der Klasse, **Vererbung öffnet** sie zur Unterklasse hin

## 2.2 Abstraktion

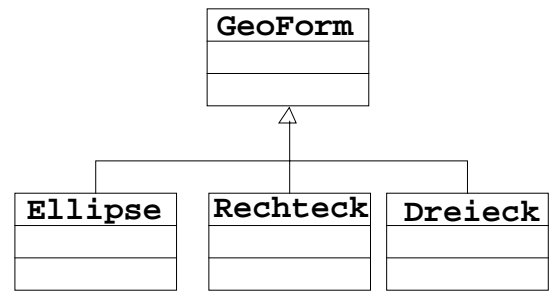
Unterklassen modellieren

**verschiedene Arten eines abstrakten Konzeptes**

(„ist spezielle Art von“, „is-a“)

**Beispiele:**

verschiedene Arten von geometrischen Formen,  
 verschiedene Arten von Dokumenten,  
 Strukturbäume in Übersetzern,  
 Design Pattern **Composite**



Einteilung in **disjunkte Objektmen**gen der Unterklassen,

kein Objekt ist zugleich in mehreren der Unterklassen

führt zu **Hierarchiebaum**.

Man muss jede Ebene vollständig kennen!

vergleiche: Klassifikation in der biologischen Systematik

**Gegenbeispiele:**

Fahrzeuge <- Landfahrzeuge, Wasserfahrzeuge <- Amphibienfahrzeuge

Schnabeltier: eierlegendes Säugetier

**signalisieren Entwurfsfehler:**

nicht durch Mehrfachvererbung lösen, sondern durch Rollen

alle **inneren Klassen der Hierarchie sind konzeptionell abstrakt**,

Objekte nur zu den Blattklassen

## Entwurf mit dem Paradigma Abstraktion

1. **Operationen, die allen Arten gemeinsam sind**,

als Methoden mit ihren Datenattributen in der Oberklasse zusammenfassen,

z. B. Operationen auf Koordinaten der Formen

2. **Operationen, die konzeptionell Gleiches leisten**,

aber unterschiedliche Ausprägungen haben,

als **Methoden-Schnittstellen** (abstrakte Methoden)

in die Oberklasse,

z. B. Flächenberechnung

3. Bei strenger Anwendung des Paradigmas **kommt Überschreiben nicht vor**,

wohl aber **Implementierung abstrakter Methoden**;

Ausnahmen z. B. Überschreiben zur Spezialisierung von „Service-Methoden“

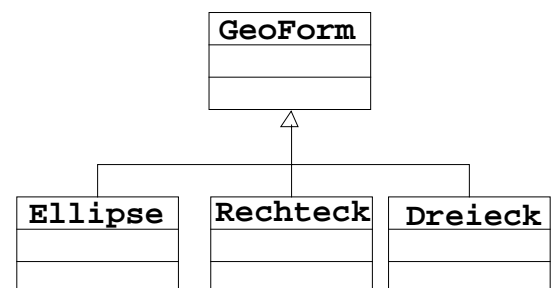
4. Beim Entwurf muss **eine Hierarchieebene hinreichend vollständig** bekannt sein;

sonst müssen die Abstraktionsentscheidungen beim Zufügen neuer Klassen revidiert werden.

Beispiel: wenn zu den Formen noch Linien hinzukommen, ist Flächenberechnung fragwürdig

5. Entwurf meist **bottom-up, um die Funktionalität zu explorieren**

**Achtung:** Spezialisierung in **eine** Unterklasse ist ein anderes Paradigma (siehe 2.4)



# Strukturbäume zu Grammatiken

Kontext-freie Grammatik definiert Baumstrukturen (abstrakte Syntax)

Beispiel:

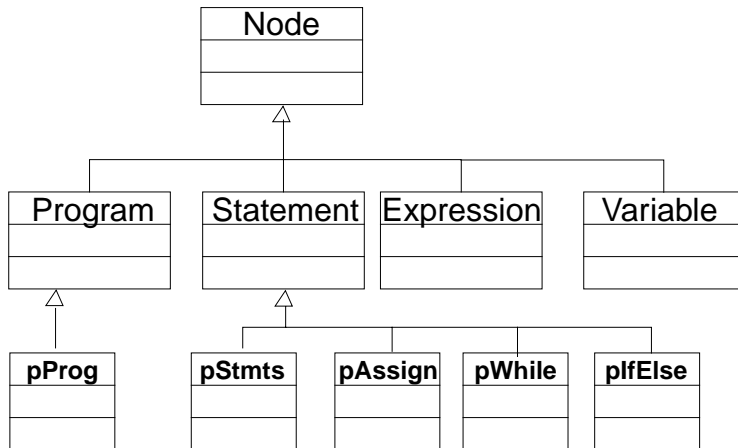
```

pProg:   Program   ::= Statement
pStmts:  Statement ::= Statement Statement
pAssign: Statement ::= Variable Expression
pWhile:  Statement ::= Expression Statement
plfElse: Statement ::= Expression Statement Statement
    
```

### 2-mal Paradigma Abstraktion:

1. alle Nichtterminalklassen abstrahiert zu Node
2. die alternativen Produktionen eines Nichtterminals N abstrahiert zu N

### 3-stufige Klassenhierarchie:



### allgemeine Knotenklasse

abstrakte Methode zum Baumdurchlauf

### Nichtterminalklassen

Methoden zum Setzen/Lesen spezieller Attribute, z. B. Typ von Expression

### Produktionenklassen

Unterbäume, Methoden zum Durchlauf, Attributberechnung

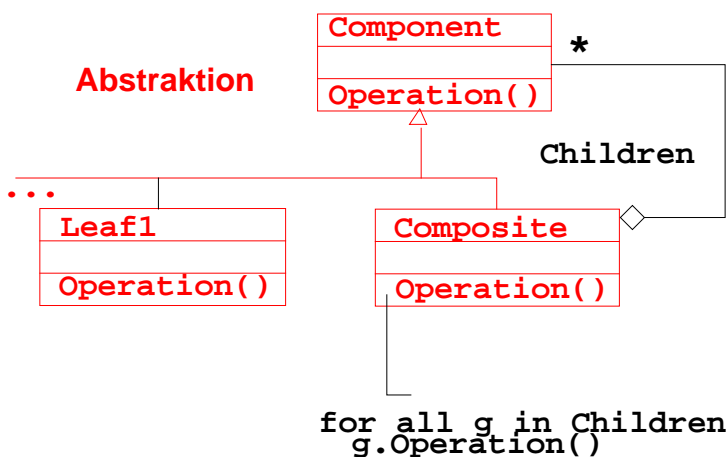
# Design Pattern Composite

## Komposition zusammengesetzter Strukturen

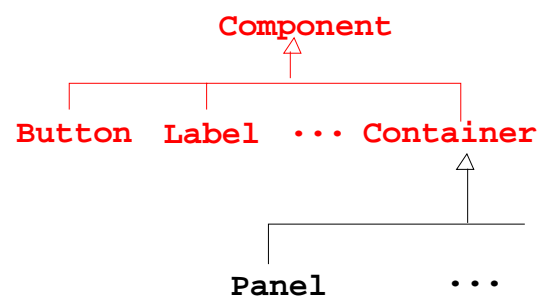
aus einfachen Komponenten unterschiedlicher Art, auch rekursiv.

Abstraktion der Blattklassen und Composite-Klasse zur Klasse Component

### Muster



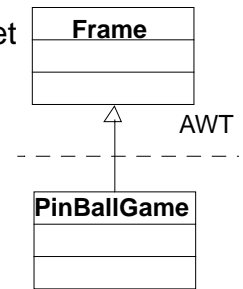
### z. B. Java AWT



## 2.3 Spezialisierung

Zu einer gegebenen Oberklasse wird **eine spezialisierte Unterklasse** gebildet

1. **Echte Erweiterung** der Oberklasse im Sinne der **Ersetzbarkeit**; die Unterklasse fügt spezifische Operationen hinzu. Die **is-a-Relation** gilt.
2. Die Oberklasse realisiert ein **komplexes Konzept**.
3. Es kommt in **anwendungsspezifischen Ausprägungen** vor.
4. Die Oberklasse liefert umfangreiche Funktionalität



**Wiederverwendbarkeit** ist sorgfältig **geplant**, sehr **aufwändig**, sehr **wirksam**

5. Balance zwischen **mächtiger Funktionalität** und **vielfältiger Verwendbarkeit**,
6. vorbereitete **Erweiterungsstellen**: Unterklasse impl. Methoden, Oberklasse ruft sie auf Techniken z.B. abstr. Methoden, Reaktion auf Ereignis, Verfahren an Schnittstelle einsetzen
7. Die Oberklasse ist häufig zusammen mit **verwandten Konzepten** Teil eines **objektorientierten Programmgerüsts (Framework)**, z. B. Java's AWT

**Abgrenzung:**

- zu *Abstraktion*: dort bottom-up, hier top-down; dort ganze Hierarchieebene, hier einzeln
- zu *inkrementeller Weiterentwicklung*: dort nicht Ersetzbarkeit, nicht is-a, Erweiterung ad hoc

## Frame Beispiel für Spezialisierung

Oberklasse **Frame** aus Java's AWT-Framework für Benutzungsoberflächen

**Konzept:**

1. Bildschirmfenster mit Bedienungsrahmen,
2. Fläche zum Aufnehmen und Anordnen von Bedienungselementen und
3. zum Zeichnen.

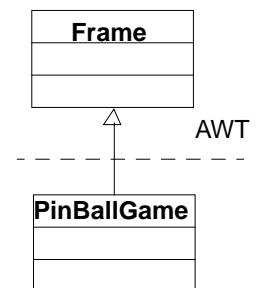
Umfangreiche **Funktionalität:**

4. Zeichnen von Fenster mit Inhalt,
5. Verändern des Fensters und des Rahmens,
6. Anordnen von Bedienungskomponenten

**Erweiterungsstellen:**

7. leere Methode `paint` zum Zeichnen, wird in Unterklasse überschrieben und aus dem Framework aufgerufen,
8. Einstellen der Strategie zur Anordnung der Bedienelemente (`LayoutManager`)
9. Ereignisbehandlung ist vorbereitet, ausfüllen mit `EventListener`

**Frame** in der AWT-Framework **zusammen mit anderen Container-Klassen** (`Panel`, `ScrollPane`, ...)



## Thread als Beispiel für Spezialisierung

Oberklasse `Thread` im Paket `java.lang`

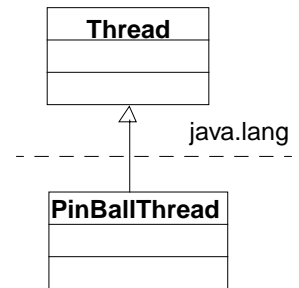
**Konzept:** parallel ausführbarer, leichtgewichtiger **Prozess**

umfangreiche **Funktionalität:**

- Prozesse starten, anhalten, synchronisieren,
- Prozessverwaltung mit Scheduler und Prozessumschaltung

**Erweiterungsstelle:**

leere Methode `run` für den Prozess-Code,  
wird von der Unterklasse überschrieben und vom System aufgerufen.



## 2.4 Rollen, Eigenschaften

### 2.4.1 Werkzeug-Material-Paradigma (statisch)

**Rolle:**

**Fähigkeit in bestimmter Weise zu (re)agieren.**

Verkäufer: anbieten, liefern, kassieren

**Eigenschaft:**

**Fähigkeit in bestimmter Weise zu (re)agieren.**

Steuerbar: starten, anhalten, wenden

Rollen und Eigenschaften als **Schnittstellen:**

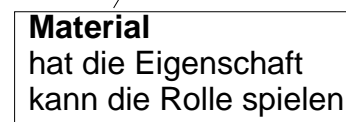
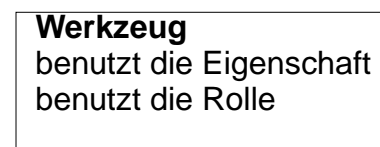
**Spezifikation** von Rollen oder Eigenschaften  
**entkoppelt die Entwicklung** von Werkzeugen  
und Material:

Zusammenfassen charakteristischer Fähigkeiten  
oder Operationen

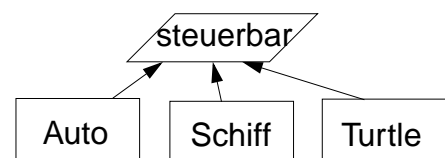
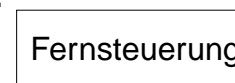
Werkzeug sieht Objekte nur in Ihrer Rolle

Rollen entkoppeln Software-Module:

Neue Werkzeuge, neues Material können  
bei der Entwicklung zugefügt werden



**Beispiel:**



## Rollen, Eigenschaften spezifiziert durch Interfaces

Die **Fähigkeiten** einer Rolle, einer Eigenschaft werden als **Methodenspezifikationen** in einem **Interface** zusammengefasst:

```
interface Movable
{ boolean start (); void stop (); boolean turn (int degrees); }
```

**Material-Klassen** haben Eigenschaften, können Rollen spielen; sie implementieren das Interface und ggf. weitere Interfaces:

```
class Turtle implements Movable
{ public boolean start () {...}
  // Implementierung von stop und turn sowie weiterer Methoden ...
}
class Car implements Movable, Motorized
{ // Implementierung der Methoden aus Movable und Motorized, sowie weiterer ...
}
```

In **Werkzeug-Klassen** werden Interfaces als **Typabstraktion** verwendet und werden die spezifizierten Fähigkeiten benutzt, z. B. innerhalb der Klasse zur Fernsteuerung:

```
Movable vehicle;... vehicle = new Car();... vehicle.turn(20);...
void drive (Movable vehicle, Coordinate from, Coordinate to)
{ vehicle.start (); ... vehicle.turn (45); ... vehicle.stop (); }
```

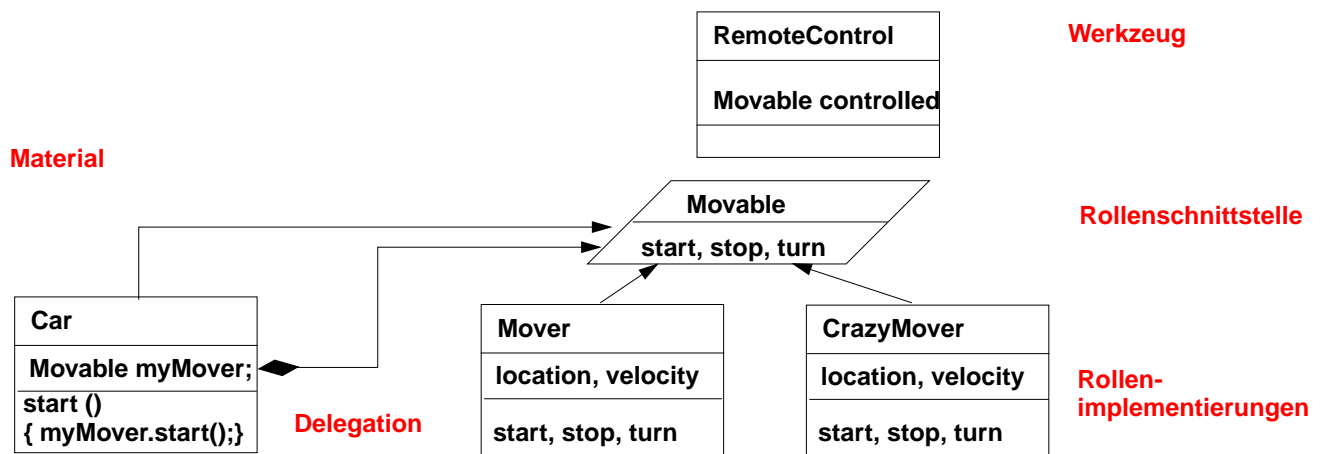
© 2005 bei Prof. Dr. Uwe Kastens

## 2.4.2 Delegation an implementierte Rolle (dynamisch)

Manche **Rolle** kann man **implementieren** - unabhängig von den Klassen, die die Rolle spielen.

**Delegation** vermeidet die wiederholte Implementierung der Rollenschnittstelle in jeder der *Material-Klassen*.

**Rollenschnittstelle** wird weiterhin für die Werkzeuge benötigt.



**Dynamisch wechselnde Rollen** sind so realisierbar, z. B.

Nach Zusammenstoß wird das Mover-Objekt durch ein CrazyMover-Objekt ersetzt.

In der Frog-Klasse wird das Kaulquappen-Verhalten durch ein Frosch-Verhalten ersetzt.

© 2013 bei Prof. Dr. Uwe Kastens

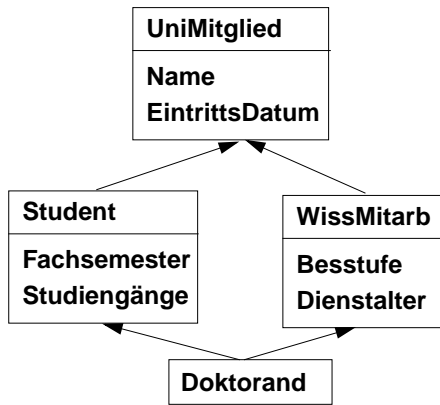
# Rollen statt Abstraktion

Häufiger Entwurfsfehler: „hat-Rollen“ als nicht-disjunkte Abstraktion modelliert

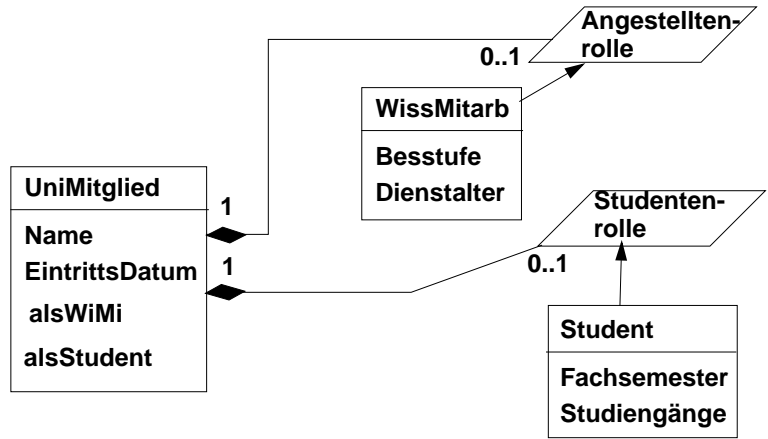
Symptome: „is-a“ gilt nicht; mehrere Rollen können zugleich oder nacheinander gespielt werden

Einsatz von Mehrfachvererbung verschlimmert das Problem

**schlecht:**  
nicht-disjunkte Abstraktion



**gut:**  
Komposition mit implementierten Rollen



**schlecht** „Abstraktion für hat-Rollen“:

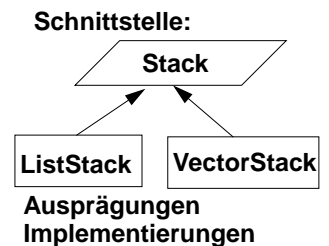
bei mehreren „Rollen-Klassen“ wird eine große Anzahl von „Kombinationsklassen“ benötigt, mehrere Rollen sind nicht zugleich möglich, dynamisch zugeordnete Rollen sind nicht möglich.

## 2.5 Spezifikation

Ein **abstraktes Konzept** wird durch eine **Schnittstelle** spezifiziert;

z. B. das Konzept Keller

```
public interface Stack
{ void push (Object x); void pop();
  Object top (); boolean isEmpty();
}
```



**Konkrete Ausprägungen** oder **Implementierungen** des Konzeptes durch Klassen, die die Schnittstelle implementieren.

Ziel: **Entkopplung** der Entwicklung von Anwendungen und Realisierungen

Gleiches Prinzip:

abstrakte Klasse statt Interface, mit Methodenschnittstellen und implementierten Methoden

Vergleich zu **Spezialisierung**:

Spezifikation betont die Schnittstelle (keine oder wenige implementierte Methoden), Spezialisierung betont das Wiederverwenden der Implementierung in der Oberklasse

Abgrenzung gegen **Abstraktion**:

Spezifikation: top-down, Ausprägungen werden **einzeln** entwickelt

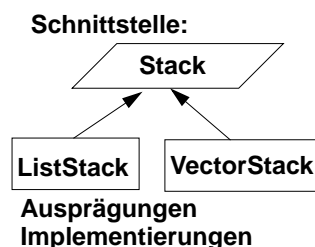
Abgrenzung gegen **Rollen**:

Spezifikation: eine konkrete Ausprägung gehört zu nur **einem** abstrakten Konzept  
Rollen: eine Materialklasse kann **mehrere** Rollen spielen



## Beispiele für Spezifikation

1. **Spezifikation einer Datenstruktur** (z. B. Stack), um ihre Implementierungen zu entkoppeln



2. **Ereignisbehandlung** in der AWT-Bibliothek für Benutzungsoberflächen:

```
interface ActionListener
{ void actionPerformed (ActionEvent e); }
```

Bei komplexeren Schnittstellen (z. B. `MouseListener`) auch **abstrakte Klasse** (`MouseAdapter`) mit Default-Implementierungen der Methoden.

3. **Abstraktes Konzept „arithmetische Typen“** mit arithmetischen Operationen +, -, \*, /, zero, one, ...:

Einige konkrete Ausprägungen:  
`Integer, Long, Float, Double, Polynom, Matrix, ...`

`Number` in der Bibliothek `java.lang` könnte so definiert sein.

**Grenzfall zum Paradigma „Abstraktion“**

## 2.6 Inkrementelle Weiterentwicklung

Die **Funktionalität** einer Oberklasse wird **genutzt**, um sie zu einer neuen Unterklasse **weiterzuentwickeln**.

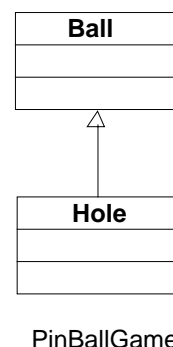
Mit **geringem Entwicklungsaufwand** wird etwas Neues geschaffen.

Die **Anforderungen der Spezialisierung** (2.3) brauchen **nicht alle erfüllt** zu werden:

1. Das **Konzept** der Oberklasse (sie ist meist konkret) braucht für die Unterklasse nicht mehr zu gelten.
2. Die **is-a-Relation** braucht nicht zu gelten.
3. Konzeptionell braucht „**subtyping**“ nicht erfüllt zu sein.
4. Die Unterklasse braucht sich **nicht auf Erweiterungen zu beschränken**; durch Überschreiben können **Methoden konzeptionell verändert oder gelöscht** werden.
5. Diese **Wiederverwendung** der Oberklasse ist **so nicht geplant** worden.

Meist zeigt die inkrementelle Weiterentwicklung **Probleme wegen unpassender** Konzepte, Methoden, Methodennamen, Methodensignaturen, Implementierungen, ...

Meist ist die bessere Lösung **Komposition statt Vererbung**:  
Ein Objekt der neuen Klasse nutzt ein Objekt der gegebenen Klasse zur Implementierung seiner Funktionalität.



## Beispiel für inkrementelle Weiterentwicklung

Die Klasse `Ball` in den Fallstudien des Buches von Budd realisiert das **Konzept von Bällen, die sich in der Ebene bewegen**. Sie hat Methoden, die Eigenschaften (Ort, Farbe, Geschwindigkeitsvektor) lesen und schreiben und den Ball zeichnen.

Im `PinBallGame` werden **Löcher als Hindernisse** benötigt. Man kann sie einfach aus der Klasse `Ball` durch **Vererbung entwickeln**:

```
class Hole extends Ball implements PinBallTarget
{
    public Hole (int x, int y)
    {
        super (x, y, 12); setColor (Color.Black);
    }

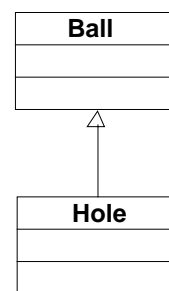
    public boolean intersects (Ball aBall)
    {
        return location.intersects (aBall.location);
    }

    public void hitBy (Ball aBall)
    {
        aBall.moveTo (0, PinBallGame.FrameHeight+30);
        aBall.setMotion (0, 0);
    }
}
```

Die Implementierung ist **beeindruckend kurz und einfach**. Sie erbt `move`, `paint`, Zugriffsmethoden, ...

Das **Konzept** „Loch“ unterscheidet sich vom Konzept „beweglicher Ball“; **is-a** gilt nicht, **subtyping** gilt nicht konzeptionell.

Es wird **nur erweitert**; die **unnötige „Beweglichkeit** von Löchern“ wird in Kauf genommen.



PinBallGame

© 2009 bei Prof. Dr. Uwe Kastens

## T. Budd: *Inheritance for Construction*

T. Budd: *Understanding Object-Oriented Programming with Java*; updated ed; Addison Wesley, 2000 pp. 129, 130, 164ff

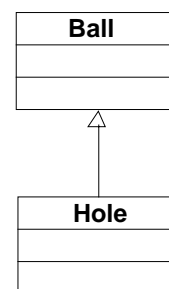
„A class can often **inherit almost all of its desired functionality** from a parent class, perhaps changing only the names of the methods used to interface to the class, or modifying the arguments. This may be true even if **the new class and the parent class fail to share any relationship as abstract concepts**.”

An example of subclassification for construction occurred in the pinball game application described in Chapter 7. In that program, the class `Hole` was declared as a subclass of `Ball`. There is **no logical relationship between the concepts of a `Ball` and a `Hole`**, but from a practical point of view much of the behavior needed for the `Hole` abstraction matches the behavior of the class `Ball`. Thus using inheritance in this situation **reduces the amount of work** necessary to develop the class `Hole`.”

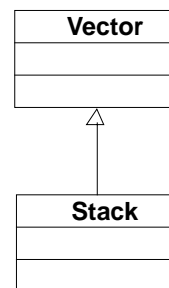
Similar explanations are given on p.130 for class `Stack` derived from `Vector`.

In Sect. 10.2, 10.3 (pp. 164) Budd discusses **composition** versus **inheritance**:

„But **composition** makes no explicit or implicit claims about substitutability. When formed in this fashion, the data types `Stack` and `Vector` are entirely distinct and neither can be substituted in situations where the other is required.“



PinBallGame



java.util

© 2013 bei Prof. Dr. Uwe Kastens

## Schlechte inkrementelle Weiterentwicklung

`java.util.Dictionary` spezifiziert das **abstrakte Konzept** einer als **Abbildung** organisierten **Menge von Paaren** (Key, Element).  
Key und Element sind **beliebige Objekte**. Signatur zweier Methoden:

```
Object put (Object key, Object elem);
```

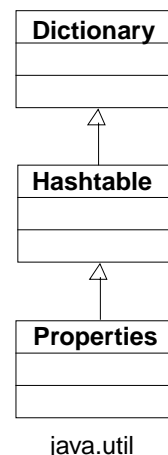
```
Object get (Object key);
```

`Hashtable` implementiert das abstrakte Konzept `Dictionary`.

`Properties` ist eine **inkrementelle Weiterentwicklung** aus `Hashtable`;  
realisiert dasselbe Konzept wie `Dictionary`,

**aber** - Key und Element sind beide **string-Objekte**,

- hat **zusätzliches Konzept**: vielstufige Suche in **Default-Properties**



### Probleme:

1. Mit `Hashtable-put` können auch nicht `string`-Paare eingefügt werden;  
in Java 1.2 wird eine `Properties`-Methode `setProperty` für `string`-Paare nachgeliefert.
2. Ergebnisse von `Hashtable-get` sind nicht notwendig Strings;  
deshalb gibt es eine `Properties`-Methode `getProperty` für `string`-Paare,  
seit Java 1.2 zeigt sie auch nicht-`string`-Paare nicht an.
3. `getProperty` sucht ggf. auch in der Default-Menge, `get` tut das nicht.
4. **Geerbte Methoden passen nicht. Chaos!!**  
Bei **Komposition statt Vererbung** könnte man sie anpassen!

## Eigene und geerbte Methoden von Properties

Method Summary	
String	<code>getProperty(String key)</code> Searches for the property with the specified key in this property list.
String	<code>getProperty(String key, String defaultValue)</code> Searches for the property with the specified key in this property list.
void	<code>list(PrintStream out)</code> Prints this property list out to the specified output stream.
void	<code>list(PrintWriter out)</code> Prints this property list out to the specified output stream.
void	<code>load(InputStream inStream)</code> Reads a property list (key and element pairs) from the input stream.
Enumeration	<code>propertyNames()</code> Returns an enumeration of all the keys in this property list, including the keys in the default property list.
void	<code>save(OutputStream out, String header)</code> <b>Deprecated.</b> This method does not throw an <code>IOException</code> if an I/O error occurs while saving the property list. As of JDK 1.2, the preferred way to save a properties list is via the <code>store(OutputStream out, String header)</code> method.
Object	<code>setProperty(String key, String value)</code> Calls the hashtable method <code>put</code> .
void	<code>store(OutputStream out, String header)</code> Writes this property list (key and element pairs) in this <code>Properties</code> table to the output stream in a format suitable for loading into a <code>Properties</code> table using the <code>load</code> method.

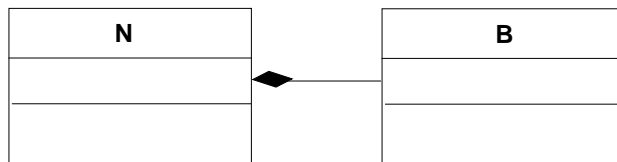
### Methods inherited from class `java.util.Hashtable`

`clear`, `clone`, `contains`, `containsKey`, `containsValue`, `elements`, `entrySet`, `equals`, `get`, `hashCode`, `isEmpty`, `keys`, `keySet`, `put`, `putAll`, `replaceAll`, `remove`, `size`, `toString`, `values`

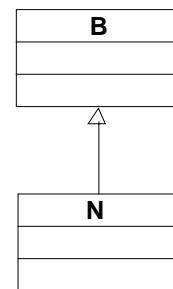
## 2.7 Komposition statt Vererbung

Gründe für die Bildung einer **neuen Klasse N** aus einer **bestehenden Klasse B** durch **Komposition** statt durch Vererbung:

1. Objekt der Klasse N **enthält** Objekt(e) der Klasse B;  
has-a-Relation statt is-a-Relation
2. N-Objekte **spielen B-Rollen**.
3. N realisiert ein **anderes Konzept** als B (kein subtyping).
4. Einige **Methoden aus B passen schlecht** für N („entfernen“, kein subtyping)
5. Die **Implementierung** von N durch B soll **nicht öffentlich** gemacht werden.
6. **Delegation** von Methoden der Klasse N an B-Objekte **entkoppelt** die Implementierungen.



Komposition



Vererbung

## Vergleich: Stack als Unterklasse von Vector

Beide Klassen sind im Package `java.util` deklariert:

```
class Vector
{ public boolean isEmpty () { ... }
  public int size () { ... }
  public void addElement (Object value) { ... }
  public Object lastElement () { ... }
  public Object removeElementAt (int index) { ... }
  ... viele weitere Methoden (-> OOP-2.23a)
}

class Stack extends Vector
{ public Object push (Object item)
  {addElement(item); return item;}

  public Object peek ()
  { return elementAt (size() - 1); }

  public Object pop ()
  { Object obj = peek ();
    removeElementAt (size () - 1);
    return obj;
  }
}
```

# Stack erbt viele Methoden von Vector

<b>Fields inherited from class java.util.Vector</b>
capacityIncrement, elementCount, elementData

<b>Fields inherited from class java.util.AbstractList</b>
modCount

protected

Method Summary	
boolean	<b>empty()</b> Tests if this stack is empty.
Object	<b>peek()</b> Looks at the object at the top of this stack without removing it from the stack.
Object	<b>pop()</b> Removes the object at the top of this stack and returns that object as the value of this function.
Object	<b>push(Object item)</b> Pushes an item onto the top of this stack.
int	<b>search(Object o)</b> Returns the 1-based position where an object is on this stack.

<b>Methods inherited from class java.util.Vector</b>
add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, lastIndexOf, remove, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toArray, toString, trimToSize

public!!

aus Version 1.2 of Java Platform API Specification, Copyright 1993-1998 Sun Microsystems

# Vergleich: Stack implementiert mit Vektor-Objekt

```

class Stack
{ private Vector theData;

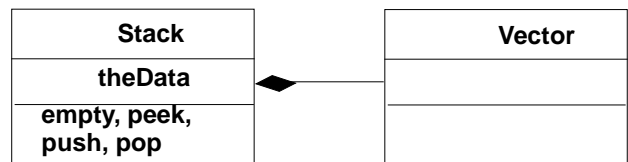
  public Stack ()
  { theData = new Vector (); }

  public boolean empty ()
  { theData.isEmpty (); }

  public Object push (Object item)
  {theData.addElement(item); return item;}

  public Object peek ()
  { return theData.lastElement (); }

  public Object pop ()
  { Object result = theData.lastElement ();
    theData.removeElementAt (theData.size () - 1);
    return result;
  }
}
    
```



Komposition

## Vergleich: Stack - Vector Vererbung - Komposition

- Stack und Vector sind **unterschiedliche Konzepte**; Vererbung verursacht subtyping-Problem
- Die **Kompositionslösung** ist **einfacher**.  
**Vollständige Definition**, knapp und **zusammenhängend** an einer Stelle.
- Die **Vererbungslösung** ist **schwerer zu durchschauen**; auf zwei Klassen verteilt; man muss hin- und herblättern.  
Viele **unnötige und unpassende Methoden** in der Klasse Vektor.
- Die **Vererbungslösung** ist **kürzer**.
- In der **Vererbungslösung** wird nicht verhindert, dass **ungeeignete Vector-Methoden** auf Stack-Objekte angewandt werden, z. B. `insertElementAt`.  
Die **Schnittstelle** der Stack-Klasse ist **viel zu groß**.
- Die **Kompositionslösung verbirgt die Implementierungstechnik**.  
Wir könnten sie ändern (z. B. lineare Liste statt Vector), ohne dass Anwendungen davon betroffen wären.
- Die Vererbungslösung hat **Zugriff auf protected Attribute** der Vector-Klasse (`elementData`, `elementCount`, `capacityIncrement`) und gibt den Zugriff an Unterklassen weiter. Komposition nutzt nur `public`-Zugriff.
- Die Vererbungsimplementierung ist etwas schneller.

© 2005 bei Prof. Dr. Uwe Kastens

## 2.8 Weitere Techniken, Varianten in mehreren Dimensionen

Beispiel `java.io.InputStream`:

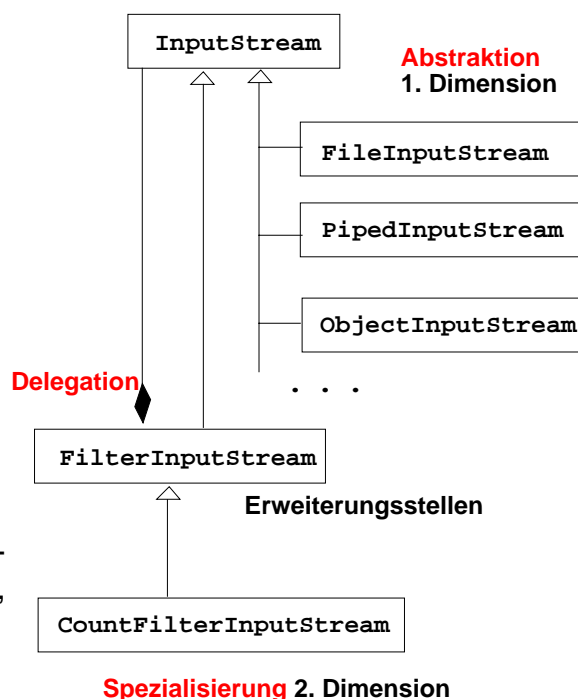
`InputStream` ist ein **abstraktes Konzept** zu verschiedenen Arten von Eingabequellen;  
**Abstraktion** zur 1. Varianten-Dimension

`FilterInputStream` bereitet **Erweiterungsstellen** für die 2. Dimension von Varianten vor:  
**Delegation** an ein `InputStream`-Objekt,  
Unterklasse von `InputStream`  
wegen **Spezialisierung**

**Spezialisierung** zu Varianten der 2. Dimension: Spezielle Filter;  
Wiederverwendung der delegierten Methoden

Bei Generierung eines `CountFilterInputStream`-Objektes wird ein **Delegationsobjekt übergeben**, das die Eingabequelle festlegt (1. Dimension):

```
new CountFilterInputStream
    (new FileInputStream ("myFile"))
```



© 2010 bei Prof. Dr. Uwe Kastens

# Eingebettete Objekte als Agenten

Schema:

Ein **Agent** handelt für ein Institut;  
 er ist untrennbar **an sein Institut gebunden**;  
 Kunden **sehen nur den Agenten** - nicht das Institut  
 Institut - Agent - Kunde

Beispiel: Versicherung - Vertreter - Kunde

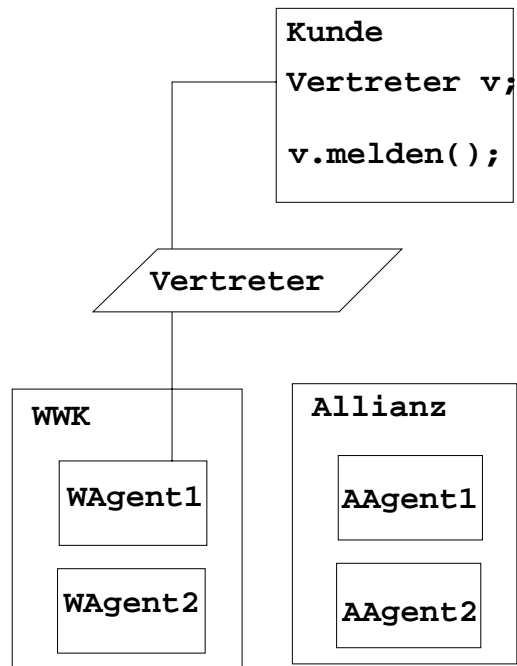
Technik: **eingebettete Klassen**

```
interface Vertreter
{ void abschließen(); void melden();}

class Versicherung
{ void regulieren() {...}

    class Agent implements Vertreter
    { void abschließen () { ... }
      void melden ()
        {... regulieren() ... }
    }
}
```

Objektbild:

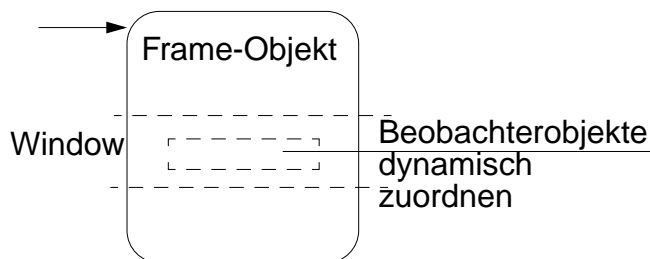
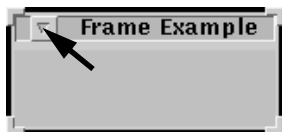


Schachtelung der Klassen: enge Einbettung der Objekte  
**Innere Objekte erfüllen global sichtbare Schnittstelle**,  
 sind also außerhalb ihrer Umgebung verwendbar.

© 2005 bei Prof. Dr. Uwe Kastens

# Beispiel: Listener der Ereignisbehandlung

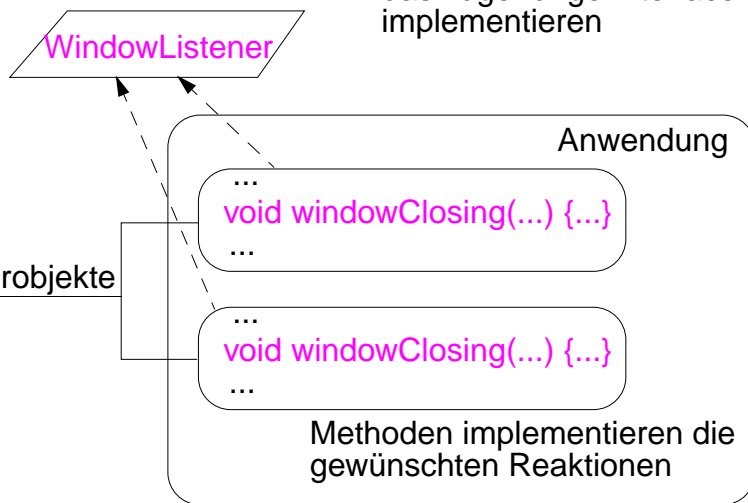
An AWT-Komponenten werden Ereignisse ausgelöst, z. B. ein WindowEvent an einem Frame-Objekt:



Ereignis auslösen:  
 zugehörige Methode in jedem Beobachter-Objekt aufrufen

„eingebettete Agenten“

**Beobachter** für bestimmte Ereignistypen:  
 Objekte von Klassen, die das zugehörige Interface implementieren



Entwurfsmuster „Observer“: Unabhängigkeit zwischen den Beobachtern und dem Gegenstand wegen Interface und dynamischem Zufügen von Beobachtern.