Assoziation und Aggregation

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

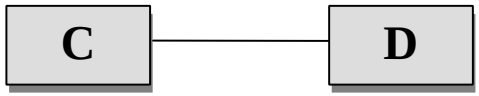
http://www.pst.ifi.lmu.de/lehre/SS06/infoII/

Ziele

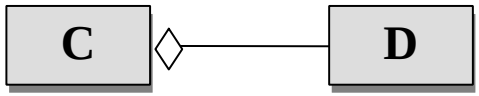
- Verstehen der Begriffe Assoziation und Aggregation
- Implementierung von Assoziationen in Java schreiben lernen
- Verstehen, wann Vererbung eingesetzt wird

Assoziation und Aggregation

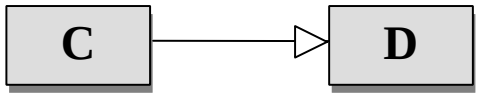
Assoziation ist eine Beziehung zwischen zwei oder mehr Klassen



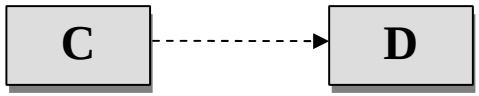
Assoziation: Die Klassen C und D stehen in Beziehung



Aggregation: Jedes Objekt von C enthält Objekte von D



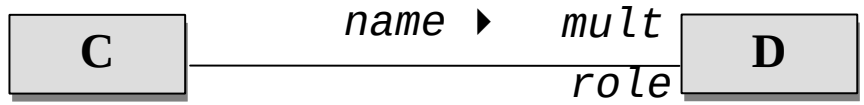
Vererbung: Die Klasse C ist Erbe der Klasse D



Abhängigkeit: Die Klasse C benützt Elemente der Klasse D (i.a. Methoden)

Assoziation

Die Assoziation ist die allgemeinste Art einer Beziehung



Das Diagramm drückt aus, daß jedes Objekt `o` von `C` mit *„mult“*-vielen Objekten von D im Beziehung steht, die die Rolle „role“ für `o` spielen.

Dabei ist *`role`* ein Name und
`mult` entweder eine natürliche Zahl, ein Stern „*“ für beliebig viele Objekte oder ein Intervall der Form `a..b` mit $a \in \mathbf{N}$, $b \in \mathbf{N} \cup \{*\}$

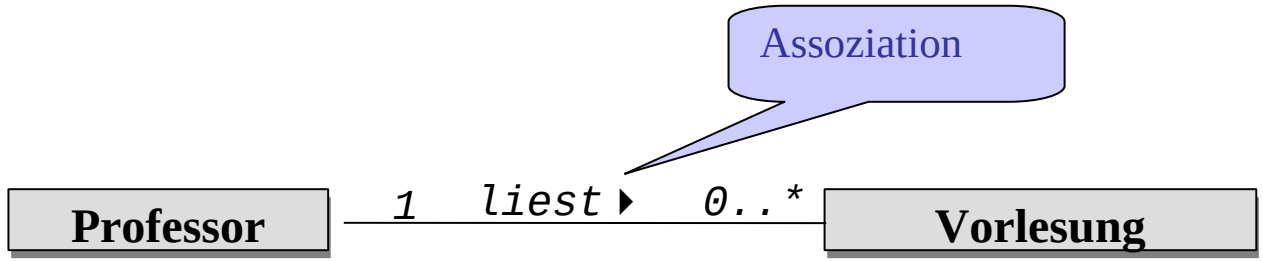
Der Name *`name`* gibt den Namen der Assoziation an.

Das ausgefüllte Dreieck ▸ bezeichnet die Leserichtung.

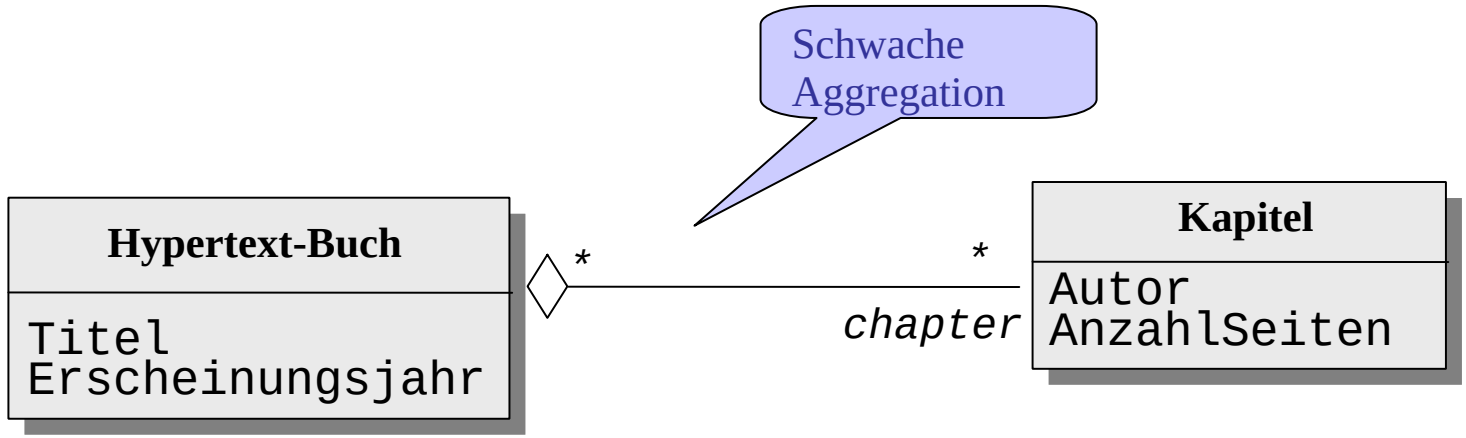
Assoziation und Aggregation

Beispiele:

1.



2.

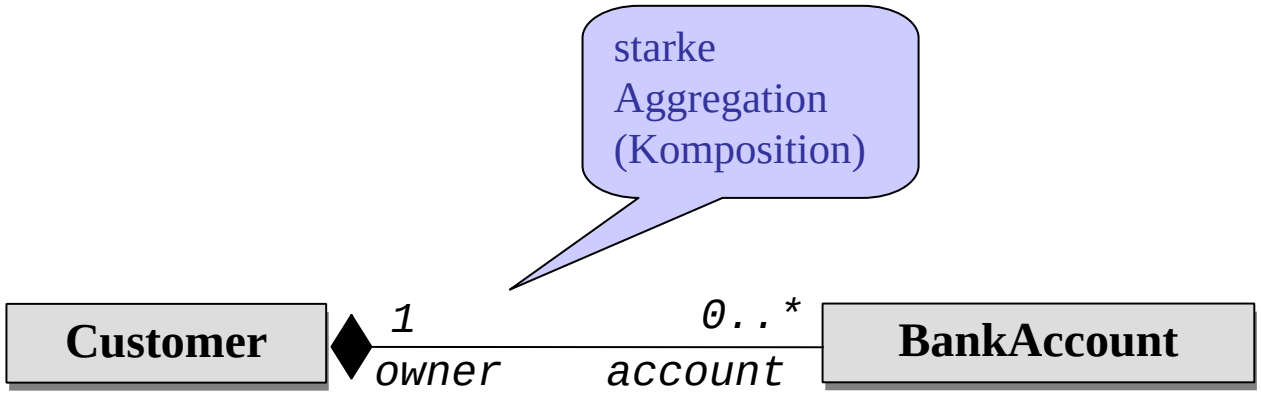


Jedes Hypertext-Buch ist gekennzeichnet durch seine Kapitel, d.h. Kapitel spielt die Rolle "*chapter*" für Hypertext-Buch.

Assoziation und Aggregation

Beispiele:

3.



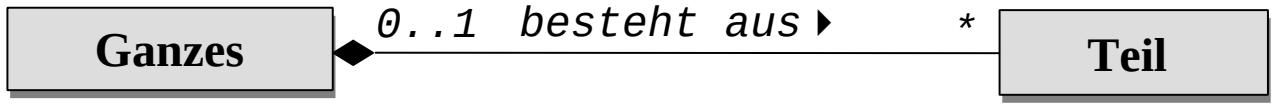
- Jeder Bankkunde besitzt 0 oder mehrere Konten, die die Rolle *"account"* spielen.
- Jedes Bankkonto hat genau einen Besitzer (*"owner"*).
- Die Lebensdauer eines Bankkontos ist durch die des Kunden beschränkt.

Schwache Aggregation

Beispiel:

- Schwache Aggregation ist eine „Teile-Ganzes“ Relation, die Lebensdauer der Teile hängt aber **nicht** von der Lebensdauer des Ganzen ab.
- Es gilt aber, dass die Objekte der schwachen Aggregation einen gerichteten azyklischen Graphen bilden. (Wenn B Teil von A, dann ist A nicht Teil von B).
- Ein Teil darf jedoch auch einem anderen Ganzen zugeordnet werden.

Starke Aggregation

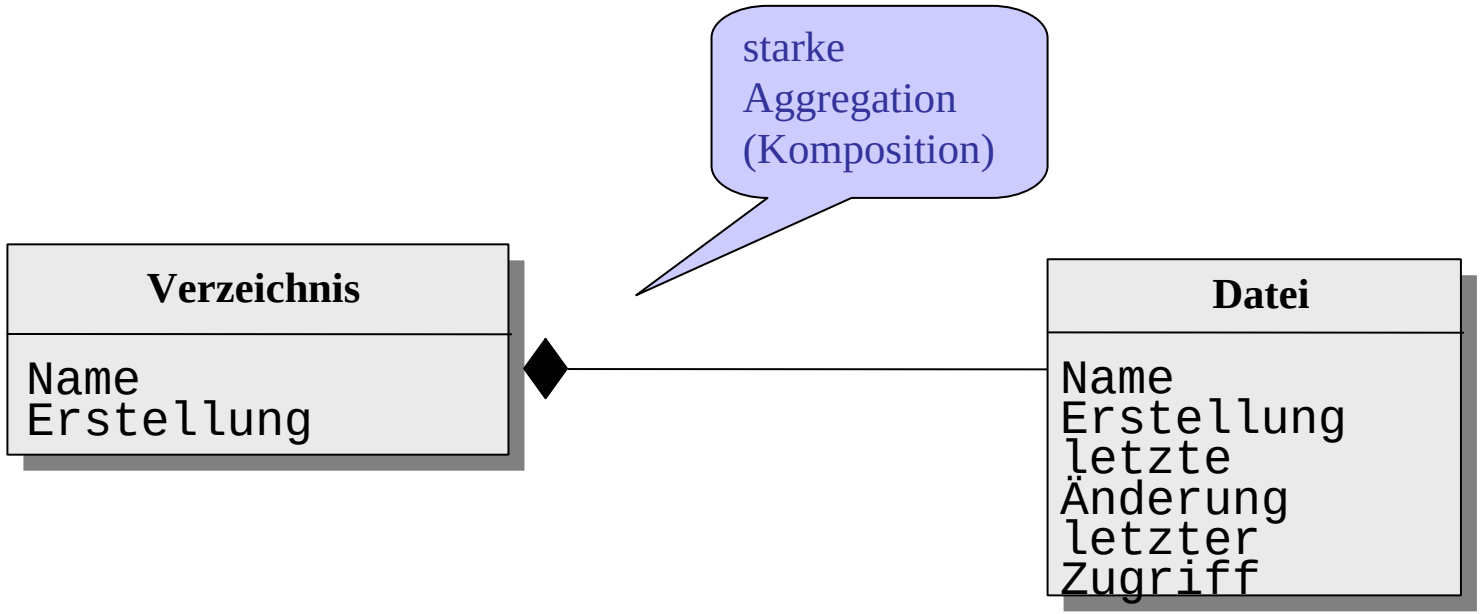


- Die Teile existieren nur innerhalb des Aggregats “Ganzes”; d.h. die Lebensdauer eines “Teil”-Objekts wird durch die Lebensdauer des zugehörigen Aggregats beschränkt.
- Objekte der Aggregation bilden einen Baum.

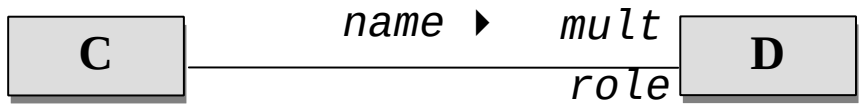
  (Wenn A ein Objekt B enthält, dann enthält B nicht A.)
- Jedes Objekt der Teilklasse kann - zu einem Zeitpunkt - nur Komponente eines einzigen Objekts der Aggregatklasse sein, d.h., die bei der Aggregatklasse angetragene Kardinalität darf nicht größer als eins sein *(unshared aggregation, strong ownership).*
- Die dynamische Semantik des Ganzen gilt auch für seine Teile *(propagation semantics).* Wird beispielweise das Ganze kopiert, so werden auch seine Teile kopiert.

Starke Aggregation

Beispiel:



Java Implementierung von Assoziationen



- In Java repräsentiert man Assoziationen durch Attribute.
- Jede Rolle `role` vom Typ `D` wird als Attribut von `C` implementiert.
- Ist die Multiplizität 0, 1 oder 0..1 erhält man ein Attribut `D role .`
- Ist die Multiplizität >1 oder * verwendet man eine Implementierung von Set<D> oder (falls eine geordnete Collection gefordert ist) eine Implementierung von List<D> .

Java Implementierung von Assoziationen

- **multD = 0..1** induziert das Schema:

```
class C {
   D myD;     …
}
```

- **multD = *** induziert das Schema:

bzw. wenn für `myD {ordered}` verlangt ist.

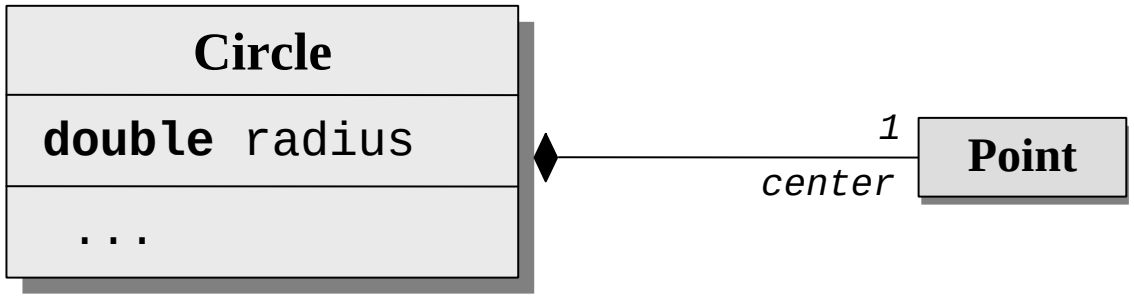
```
class C {
   Set<D> myD; …
}
```

```
class C {
   List<D> myD; …
}
```

Implementierung von starker Aggregation

- Erzeugung der Komponenten durch Konstruktor des Aggregats

Beispiel



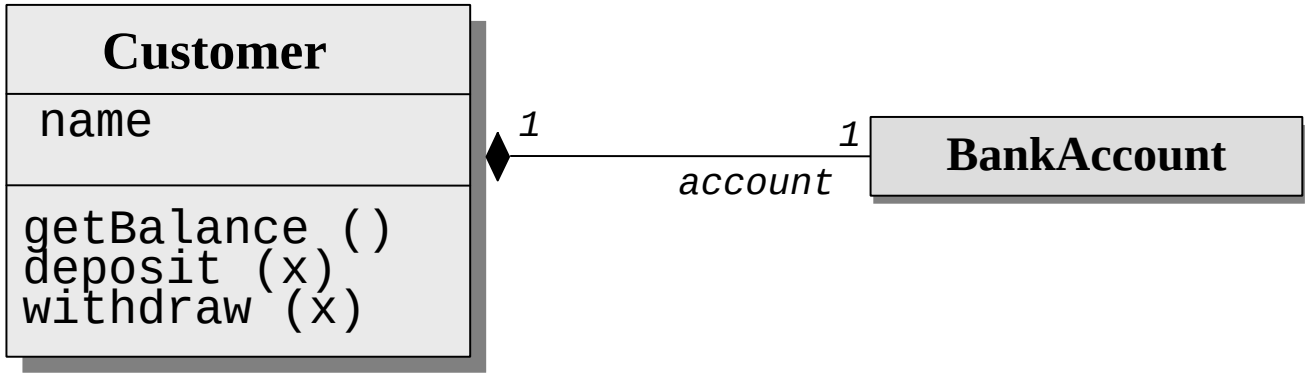

```
class Circle
    {
        private double radius;
        private Point center;

        public Circle(double rad, double x, double y)
        {
            radius = rad;
            center = new Point(x,y);
        }
        ...
    }
```

Das Point Objekt wird durch den Konstruktor von Circle erzeugt und **sollte keine Beziehungen zu "äußeren" Objekten besitzen**.

Implementierung von starker Aggregation

Beispiel Assoziation mit Multiplizität 1:

Kunde, der genau ein Bankkonto besitzen muss.



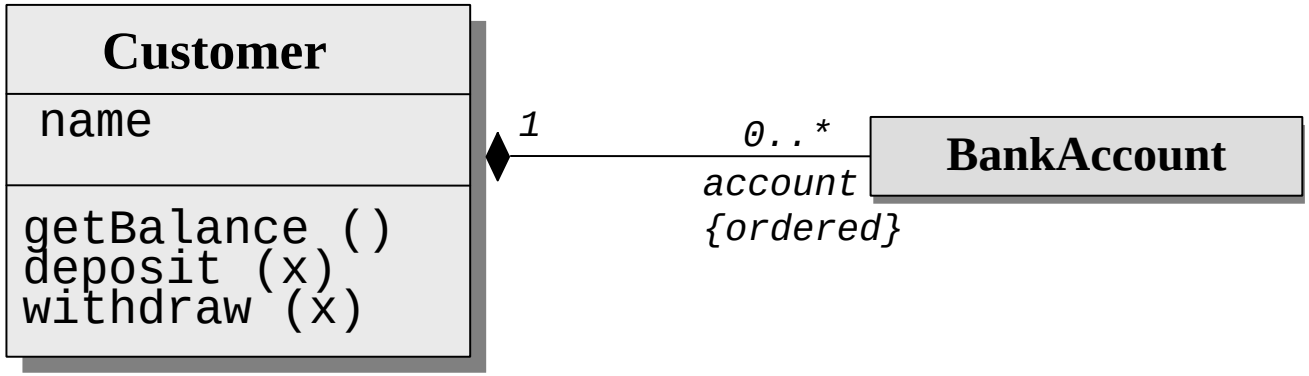
Implementierung von starker Aggregation

```
class Customer
    {
        private String name;
        private BankAccount account;

        public Customer(String cname, double initialBalance)
        {
            name = cname;
            account = new BankAccount(initialBalance);
        }
        public String getName()
        {
            return name;
        }
        public double getBalance()
        {
            return account.getBalance();
        }
        public void transferTo(BankAccount other, double amount)
        {
             account.transferTo(other, amount);
        }
        public void withdraw(double x)
        {
            account.withdraw(x);
        }
        public void deposit(double x)
        {
            account.deposit(x);
        }
    }
```

Implementierung von Assoziationen mit Multiplizität *

Beispiel:

Kunden mit 0 oder mehreren Bankkonten



Die Klasse ArrayList

```
class ArrayList<E>
{       public ArrayList()
               //Constructs an empty list with an initial capacity of 10.

        public ArrayList(int initialCapacity)
            //Constructs an empty list with the specified initial capacity.


        public E get(int index)
            //Returns the element at the specified position in this list.

        public E set(int index, E element)
               //Replaces the element at the specified position
               //in this list with the specified element.

        public boolean add(E element)
               //Appends the specified element to the end of this list.

        public int size()
               //Returns the number of elements in this list.
}
```

Implementierung von Assoziationen mit Multiplizität *

```
import java.util.*;

class Customer
    {   private String name;
        private List<BankAccount> accounts;

        public Customer(String cname)
        {   name = cname;
            accounts = new ArrayList();
        }

        public void createAccount(double initialBalance)
        {   BankAccount a = new BankAccount(initialBalance);
            accounts.add(a);
        }

        public String getName()
        {   return name;
        }
```

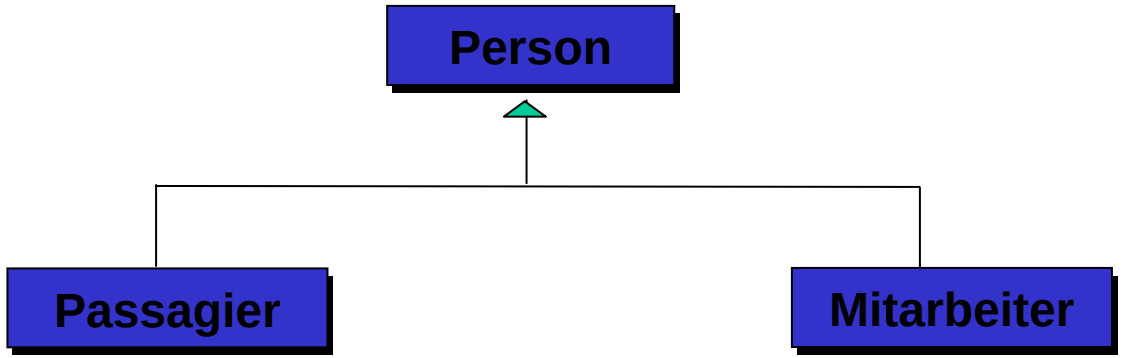
fügt neues Element an das Ende der ArrayList an.

Implementierung von starker Aggregation

```
    . . .

    public double getBalance(int i)
    {   BankAccount a = accounts.get(i);
        return a.getBalance();
    }

    public void transferTo(int i, BankAccount other, double amount)
    {   BankAccount a = accounts.get(i);
        a.transferTo(other, amount);
    }

    public void withdraw(int i, double x)
    {   BankAccount a = accounts.get(i);
        a.withdraw(x);
    }

    public void deposit(int i, double x)
    {   BankAccount a = accounts.get(i);
        a.deposit(x);
    }
}
```

Einsatz von Vererbung: Vererbung vs Rollen

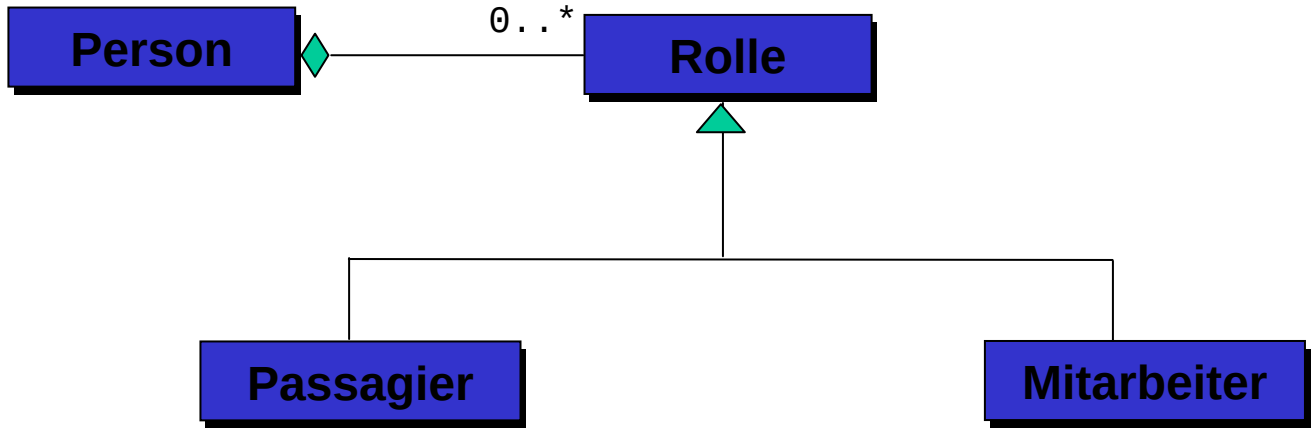
Die (einfache) Vererbungsbeziehung muss mit Vorsicht eingesetzt werden. Betrachten Sie folgendes Beispiel:



Was passiert, wenn ein Mitarbeiter als Passagier auftritt?

Das Rollenmuster

Das **Rollenmuster** erweitert die Verantwortlichkeit eines Objekts durch Delegation von Arbeit an ein anderes Objekt.



`Passagier` und `Mitarbeiter` sind die Arten von Rollen, die eine Person besitzt.

Einsatz von Vererbung: Vererbung vs Rollen

In Java kann ein Objekt nicht seine Klasse wechseln. Wird also das `Mitarbeiter`-Objekt gelöscht und ein neues `Passagier`-Objekt erzeugt?

Oder ist `Passagier-Mitarbeiter` ein Objekt einer gemeinsamen Subklasse von `Passagier` und `Mitarbeiter`?

Dies ist in Java **nicht** möglich, da es **keine mehrfache Vererbung** gibt.

Besser ist es, das Rollenmuster zu benutzen.

Ein Muster ist eine schematische Darstellung von Entwurfslösungen

Einsatz von Vererbung: Vererbung vs Rollenmuster

- Das **Rollenmuster** ist gut, wenn ein Objekt seine **Rolle wechseln** kann oder **mehrere Rollen** besitzt.

- **Vererbung** ist gut, wenn gilt:

  1. **„ist-Spezialisierung-von“,** nicht “Rolle-gespielt-bei“

     Beispiel: Passagier und Mitarbeiter sind spezielle Arten Personen**rollen**

  2. Das Erbenobjekt muss **niemals seine Klasse wechseln**.

     Beispiel: ein Passagierobjekt bleibt immer ein Passagierobjekt

  3. Die Oberklasse wird **erweitert** (und nicht redefiniert)

  4. Die Unterklasse bezeichnet **spezielle Arten** von Rollen, Transaktionen, Geräten

Zusammenfassung

- Eine Assoziation ist eine Relation (Beziehung) zwischen Klassen.
- Eine Aggregation ist eine Teile-Ganzes-Beziehung und damit eine spezielle Assoziation. Bei einer starken Aggregation sind die Teile existenzabhängig vom Ganzen. Insbesondere kann jedes Teil nur zu einem Aggregat gehören.
- Eine Rolle myB (vom Typ B) mit Multiplizität 1 oder 0..1 wird durch ein Attribut myB vom Typ B implementiert, bei Multiplizität * bekommt es den Typ Set<D> (oder List<D>, falls die Elemente von D geordnet sein sollen).
- Bei der starken Aggregation werden die abhängigen Objekte im Konstruktor des Aggregats erzeugt, um die Lebensdauerabhängigkeit zu garantieren.
- Die **Vererbungsbeziehung muss mit Vorsicht eingesetzt** werden. Die Vererbung ist angebracht bei einer Spezialisierung. Falls Objekte verschiedene Rollen spielen können, ist es besser, Komposition oder Assoziation zu verwenden