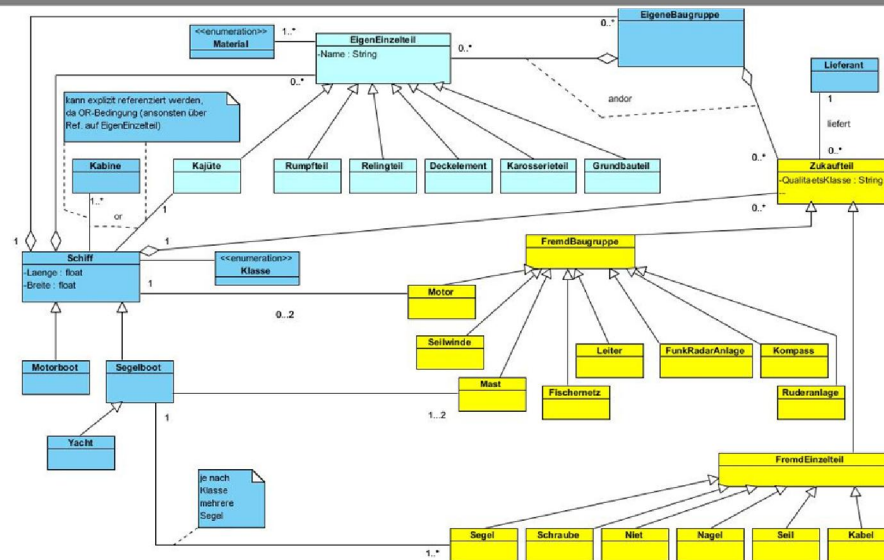


# Software Engineering I

## Grundlagen der objektorientierten Datenmodellierung mit UML Instanzen- und Klassendiagramme

Institut für Angewandte Informatik (IAI)



# Objekte, Klassen, Instanzen, Methoden, Operationen ???

## ■ Objekte:

- reale Elemente in Alltag, Problemumfeld und Programm

## ■ Instanzen:

- Alternative Bezeichnung für Objekt (Programmierung, Modellierung)
- Manchmal auch *Objektinstanzen* genannt

## ■ Klassen:

- Beschreibung (Definition, Schablone) eines Objekts.
- Eine Klasse besteht aus Attributen (Eigenschaften bzw. Daten) und Methoden (Verhalten)

## ■ Operation:

- Beschreibung des Verhaltens von Objekten einer Klasse

## ■ Methode:

- Realisierung (Quellcode) einer Operation
- Wird auch als alternativer Begriff für Operation verwendet

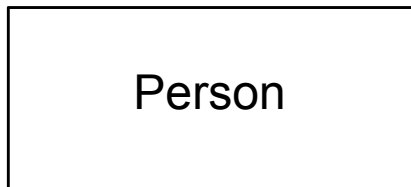
## ■ Objektdiagramm, Instanzendiagramm

- Formale grafische Notation, um **real vorhandene Objekte** und deren Relationen zu anderen Objekten darzustellen.
- Es können beliebig viele Objekte desselben Typs (Klasse) existieren.
- Besonders nützlich, um Testfälle (vor allem Szenarien) zu dokumentieren und Beispiele zu diskutieren.

## ■ Klassendiagramm

- Grafische Notation, um **Klassen** und deren Relationen zu anderen Klassen darzustellen.
- **Schema, Muster oder Template** (Schablone) zur Beschreibung vieler möglichen Objektinstanzen. Ein Klassendiagramm beschreibt Objektklassen.
- Jede Klasse ist nur einmalig vorhanden.
- Ein gegebenes **Klassendiagramm** entspricht einer unendlichen Menge von **Instanzendiagrammen**

# Darstellung von Klassen und Instanzen



Klasse



Instanz

UML-Notation für Instanznamen:

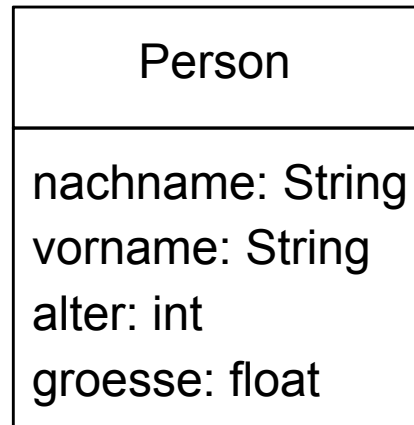
**Instanzname : Klasse** wenn die Instanz benannt werden soll. (Instanzname entspricht i. A. dem Referenznamen im Quellcode)

**: Klasse** bei einer anonymen Instanz nur Klassenname mit Doppelpunkt

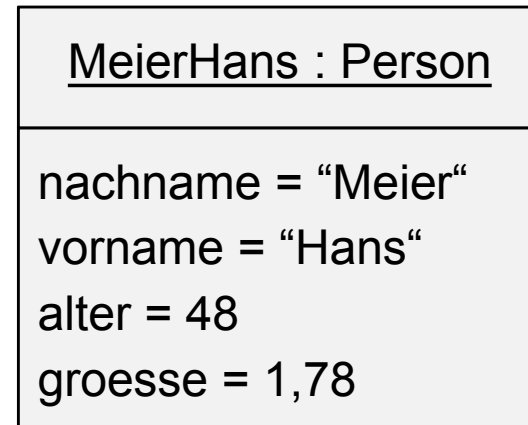
**Instanzname** wenn nur der Instanzname ausreicht ()

Der Instanzname wird unterstrichen

# Darstellung von Klassen- und Instanz-Attributen

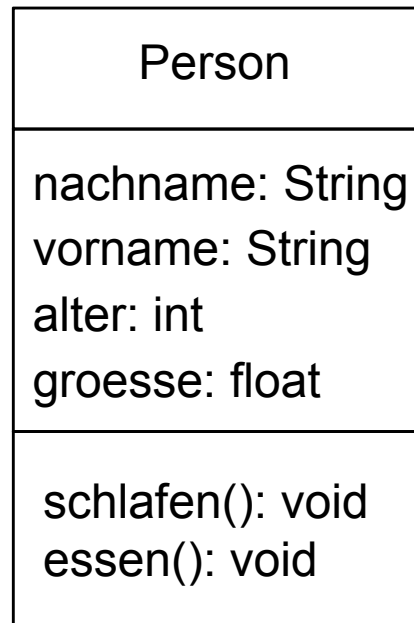


Klasse

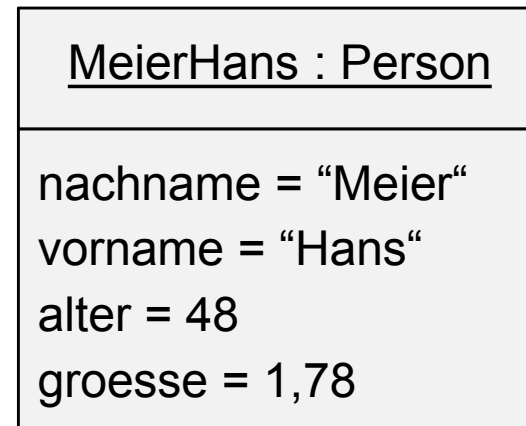


Instanz

# Darstellung von Klassen- und Instanz-Methoden



Klasse



---

  
(keine Methoden)

Instanz

# Klassen und Instanzen: allgemeine Darstellung

| Klassenname                                                        |
|--------------------------------------------------------------------|
| Attribut<br>Attribut: Datentyp<br>Attribut: Datentyp = Anfangswert |
| Operation<br>Operation()<br>Operation(Arg-Liste): Ergebnistyp      |

Klasse

| <u>Instanzname : Klassenname</u> |
|----------------------------------|
| Attribut : Datentyp = "Wert"     |

Instanz

# Klassen und Instanzen: Mapping zu Java-Code

| BeispielKlasse                                                       |
|----------------------------------------------------------------------|
| Attribut1<br>Attribut2: float<br>Attribut3: String = "Hi Welt"       |
| Operation1<br>Operation2()<br>Operation3(ff: float, ii: int): double |

Klasse

```
public class BeispielKlasse{  
  
    public int Attribut1;  
    public float Attribut2;  
    public String Attribut3="Hi Welt";  
  
    public void Operation1(){ ... };  
    public void Operation2(){ ... };  
    public double Operation3( float ff,  
                             int ii  ){  
  
        // some code lines ...  
    };  
}
```

Java-Code



## Beispiel zur Instanzen- und Klassendarstellung

- Gegeben: identifizierte Objekt und deren Attribute  
(S. Grundlagen der Objektorientierung)



### PKW:

Höhe, Breite, Länge, Leistung, Verbrauch, Gewicht,  
Farbe, an/aus, offen/zu, ...  
fahren, stehen, tanken, beschleunigen, ...  
➔ Motor, Räder, Sitze, Besitzer, Faltdach, ...



### LKW:

Höhe, Breite, Länge, Leistung, Verbrauch, Gewicht,  
Farbe, an/aus, offen/zu, ...  
fahren, stehen, tanken, beschleunigen, ...  
➔ Motor, Räder, Sitze, Besitzer, Kran, Seilwinde, ...



### Bus:

Höhe, Breite, Länge, Leistung, Verbrauch, Gewicht,  
Farbe, an/aus, offen/zu, Platzanzahl, ...  
fahren, stehen, tanken, beschleunigen, ...  
➔ Motor, Räder, Sitze, Besitzer, Mikrofon, ...

## Beispiel zur Klassen-Darstellung

| PKW                                                                 |
|---------------------------------------------------------------------|
| hoehe: int<br>breite: int<br>laenge: int<br>• • •<br>offen: boolean |
| fahren(): void<br>stehen(): void<br>• • •                           |



| LKW                                                                 |
|---------------------------------------------------------------------|
| hoehe: int<br>breite: int<br>laenge: int<br>• • •<br>offen: boolean |
| fahren(): void<br>stehen(): void<br>• • •                           |



| Bus                                                                 |
|---------------------------------------------------------------------|
| hoehe: int<br>breite: int<br>laenge: int<br>• • •<br>offen: boolean |
| fahren(): void<br>stehen(): void<br>• • •                           |



## Beispiel zur Instanzen-Darstellung

### Fiat500 : PKW

hoehe : int = 1440  
breite: int = 1350  
laenge: int = 3240  
• • •  
offen : boolean = true



### DB-ASW : LKW

hoehe : int = 2950  
breite : int = 2490  
laenge : int = 6858  
• • •  
offen: boolean = false



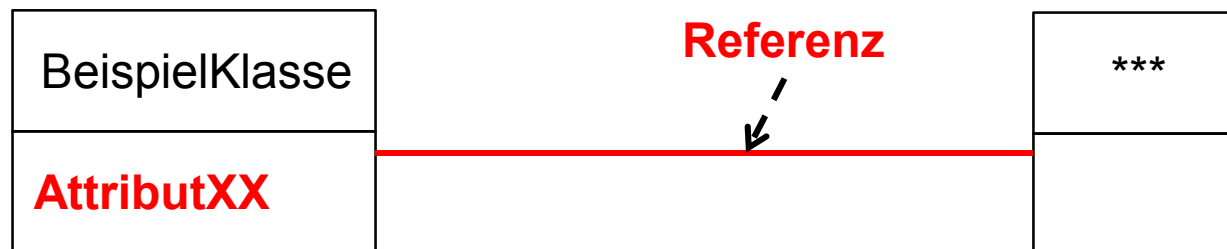
### DB-O321 : Bus

hoehe : int = 2800  
breite : int = 2450  
laenge : int = 11250  
• • •  
offen : boolean = false



## Begrifflichkeit: *Attribut* – *Referenz*

- „**Attribut**“:
  - primitive Typen (int, float, double, ...)
  - Arrays primitiver Typen
  - „primitive Klassen“ (Float, Double, Integer, String, ...)
- „**Referenz**“:
  - Höherwertige Elemente (Klassen und Interfaces)
- **Attribute** werden grafisch **innerhalb** des Klassen-Symbols dargestellt, **Referenzen** als Verbindungslinien **zwischen zwei Klassen** (Assoziation)



## Referenzen zwischen zwei Klassen bzw. Instanzen:

### Assoziationen



#### Label:

Beliebiger Bezeichner der Assoziation über/unter der Mitte der Linie. Beispiel: „*wird verwendet von*“

Label hat **keinen** Einfluss auf den späteren Quellcode!

#### Rolle:

**Genaue** Beschreibung der verknüpften Klasse aus der Sicht der jeweils anderen Klasse. **Steht am Ende der Linie.**

(„*Rolle\_2* ist die Rolle, die *Klasse\_2* aus Sicht von *Klasse\_1* spielt“)

**Multiplizität:** (auch „Kardinalität“) Anzahl der referenzierten Elemente

## Assoziation: Beispiel

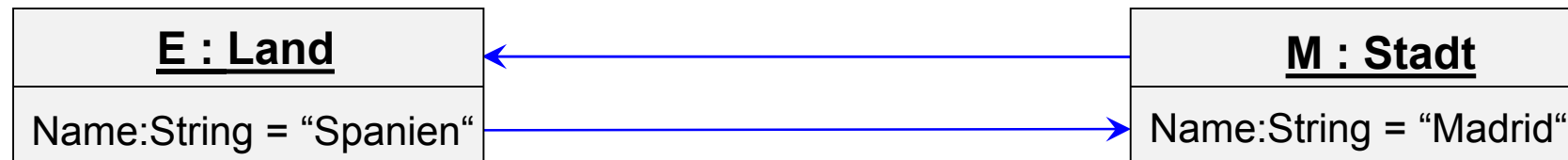


## ■ Bemerkungen zur Darstellung der Beziehung zwischen zwei **Instanzen**



Bidirektionale Assoziationen ohne Beschriftung sind bei **Instanzendiagrammen** nicht eindeutig, meist liegt tatsächlich eine Unidirektionale Beziehung vor! (Programmiersprachen kennen nur unidirektionale Assoziationen)

→ Eindeutiger sind dann gerichtete Assoziationen:



## Assoziationen: Mapping zu Java-Code



```
public class Land{

    private String Name;
    // hat Hauptstadt
    private Stadt stadt; //(hauptstadt)
    . . .
}
```

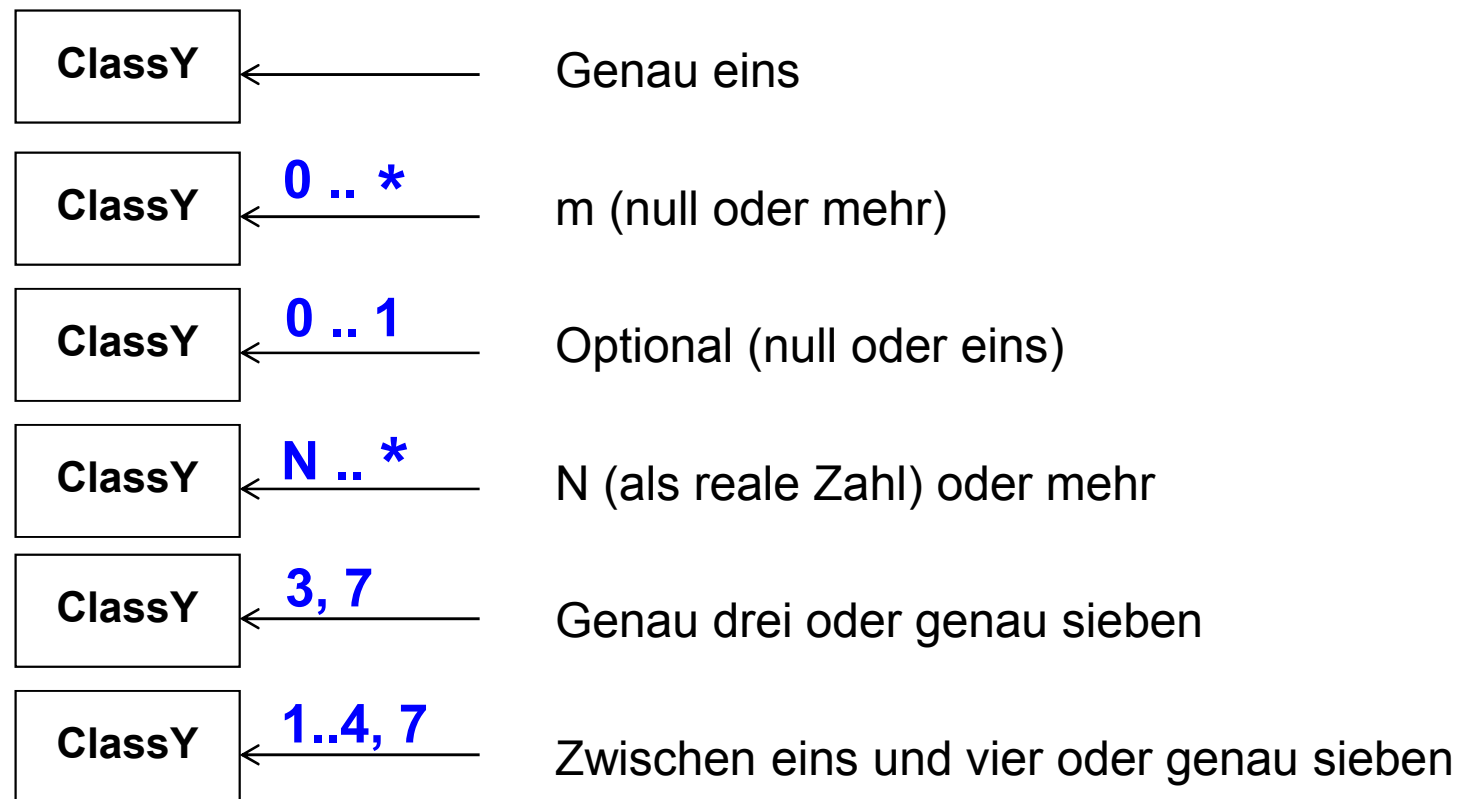
```
public class Stadt{

    private String Name;
    // ist Hauptstadt von
    private Land land;
    . . .
}
```

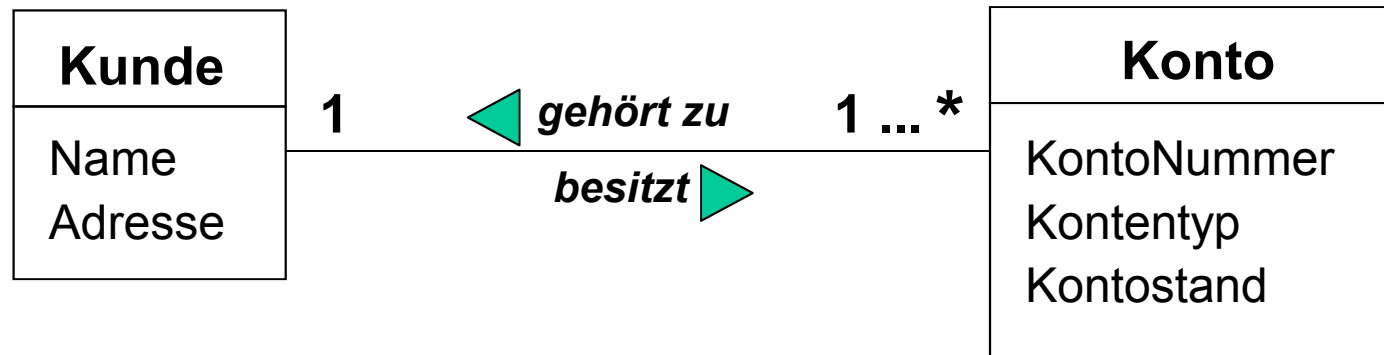


## Multiplizitäten

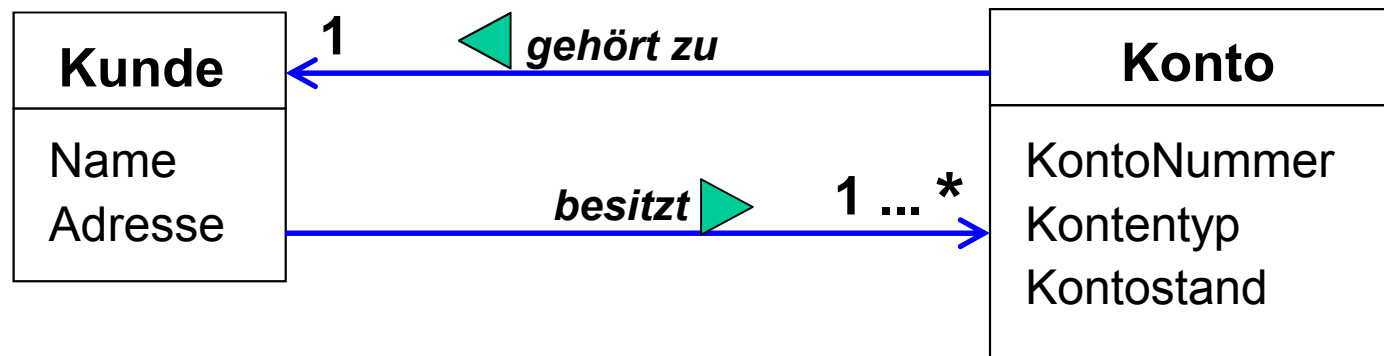
- Stellen die Anzahl der assoziierten Objekte einer Klassenbeziehung dar.
- Ein Objekt einer Klasse referenziert **X** Objekte der Klasse *ClassY*



## Multiplizitäten: Beispiel



## Alternativ (und besser lesbar): gerichtete Assoziationen

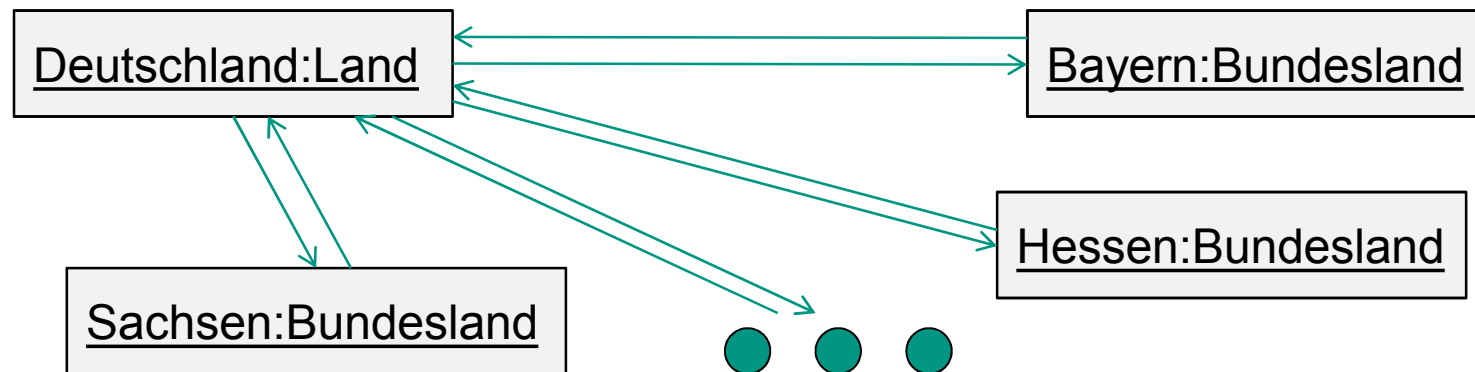


# Multiplizitäten: Beispiel Instanzen- u. Klassendiagramm

## ■ Klassendiagramm:



## ■ Instanzendiagramm:

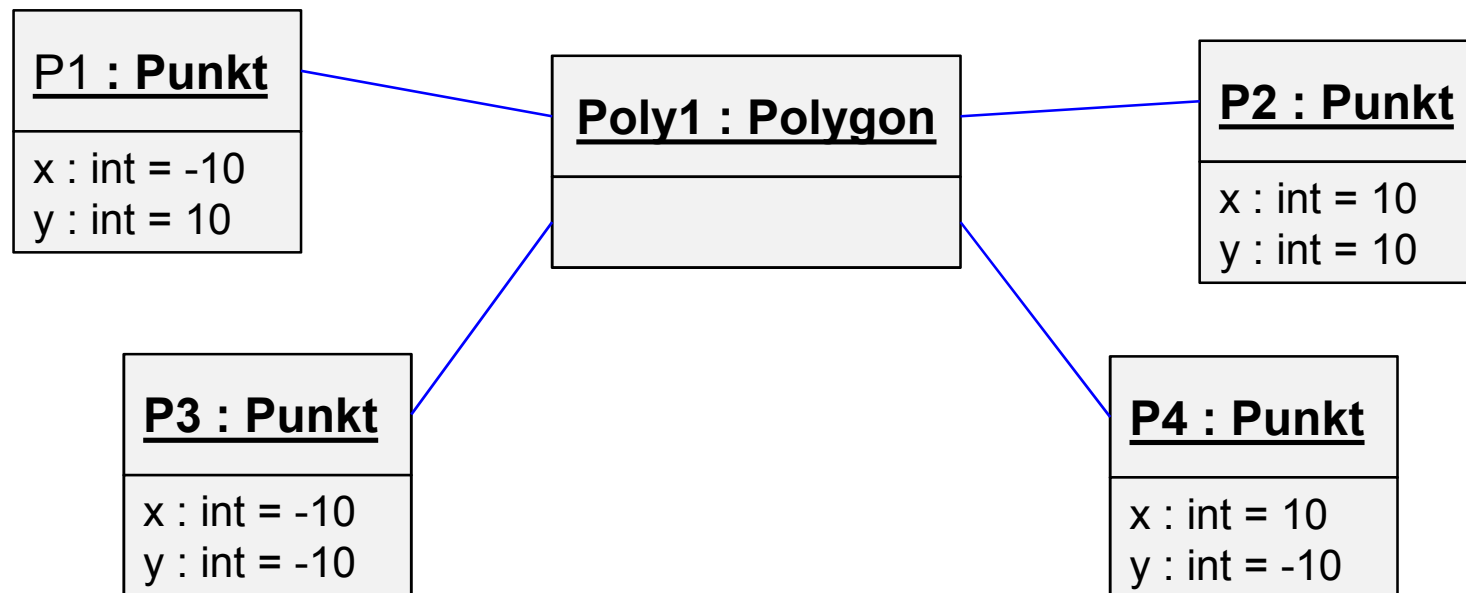


Es sollten keine Multiplizitäten bei (wichtigen) Instanzen verwendet werden!

# Übungsaufgaben

## Aufgabe 1: Instanzendiagramm → Klassendiagramm

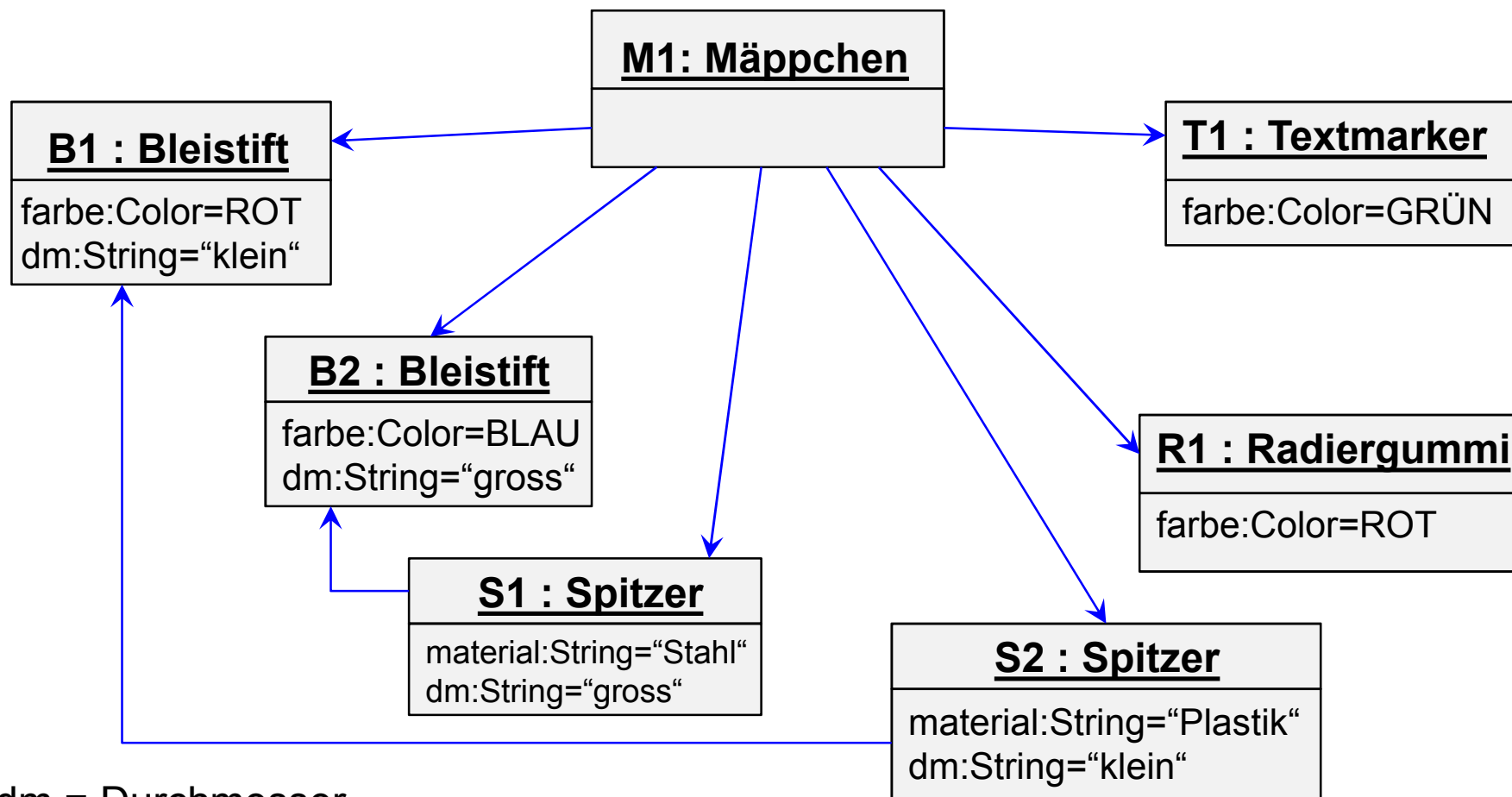
- Entwickeln Sie ein Klassendiagramm aus folgendem Instanzendiagramm



- Wie viele Punkte sind erforderlich, um ein Polygon zu konstruieren?
- Welche Konsequenzen ergeben sich aus bidirektionalen Verbindungen?

## Aufgabe 2: Instanzendiagramm → Klassendiagramm

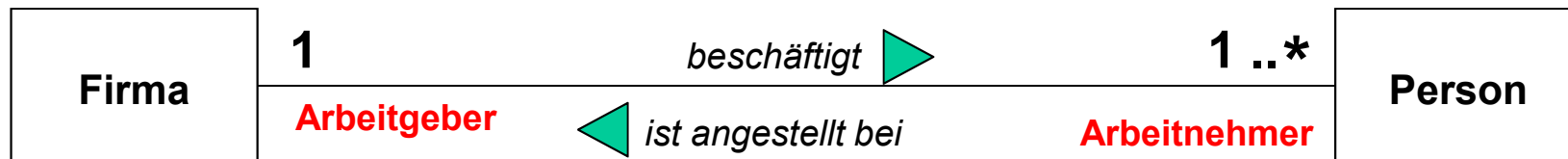
- Entwickeln Sie ein Klassendiagramm aus folgendem Instanzendiagramm



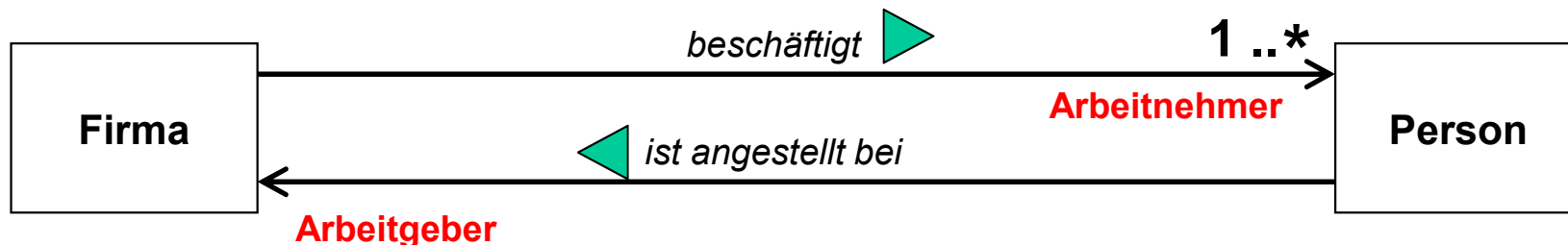
dm = Durchmesser

## ■ Ein Rollenname

- ist ein Name, der ein Ende einer Assoziation eindeutig identifiziert.
- beschreibt, welche Aufgabe ein Objekt in einer Assoziation wahrnimmt bzw. welche Rolle es aus der Sicht einer anderen Klasse spielt

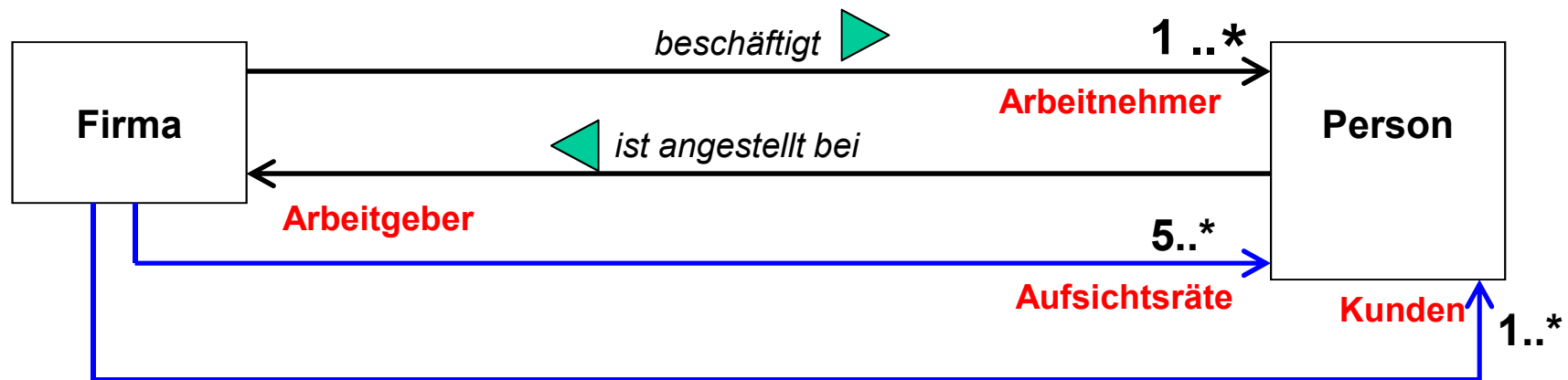


## Alternativ: gerichtete Assoziationen



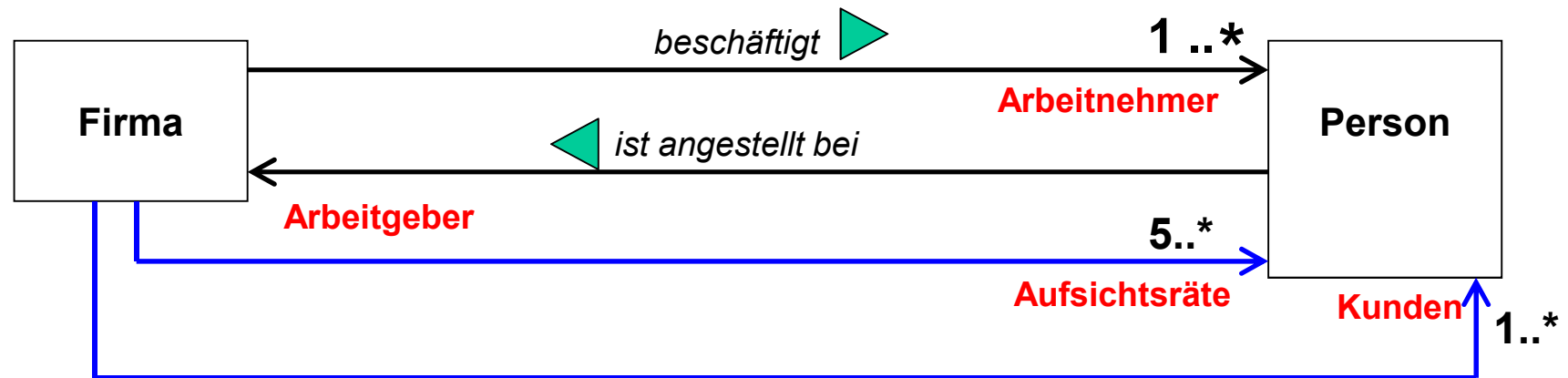
# Rollenamen

- Rollennamen werden **explizit verlangt**, wenn
  - mehrere Assoziationen von einer Klasse zu einer anderen existieren
  - mithilfe eines Modellierungswerkzeugs automatisch Quellcode erzeugt werden soll (Rollennamen werden dort oft als Referenznamen verwendet)





# Multiplizitäten + Rollen: Mapping zu Java-Code



```
public class Firma{

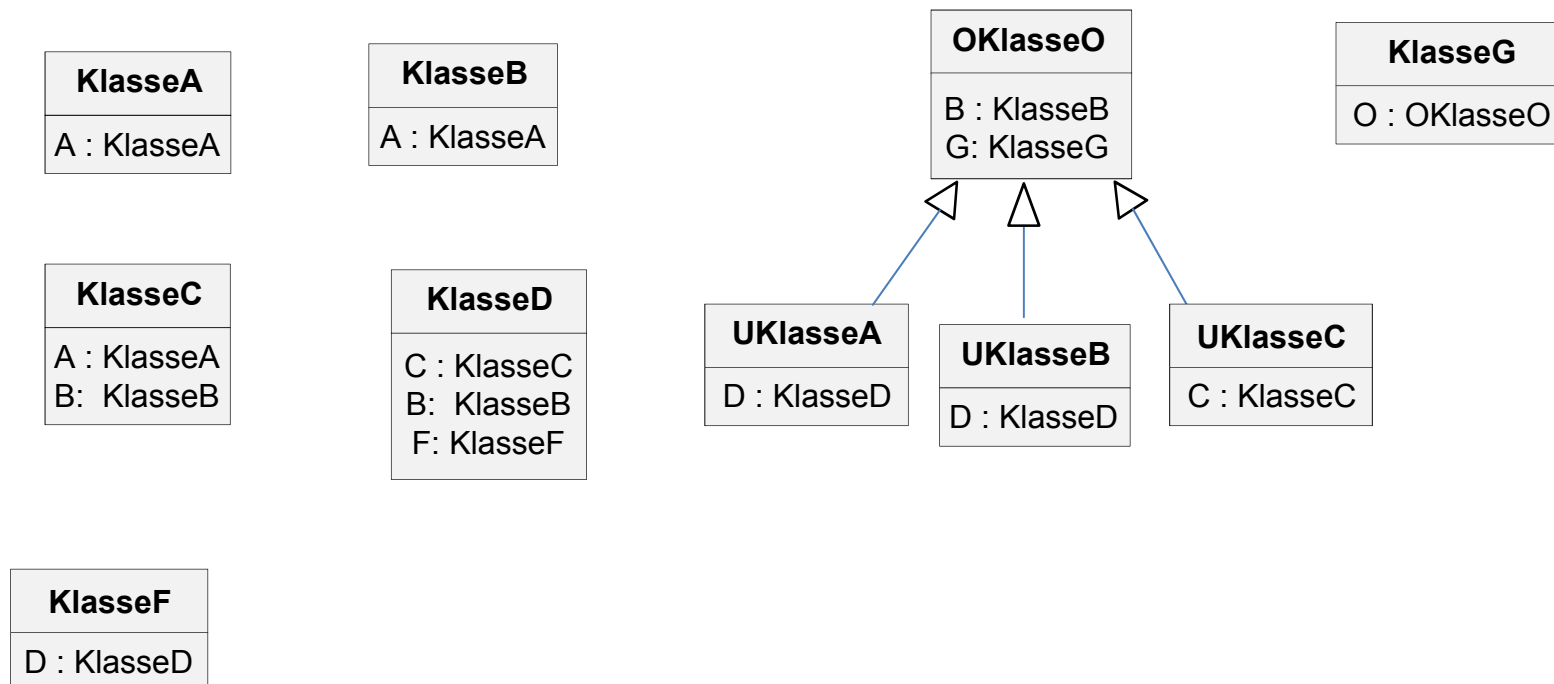
    // beschaeftigt
    private List<Person> Arbeitnehmer;
    private List<Person> Aufsichtsrat;
    private List<Person> Kunde;
    . . .
}
```

```
public class Person{

    // ist angestellt bei
    private Firma Arbeitgeber;
    . . .
}
```

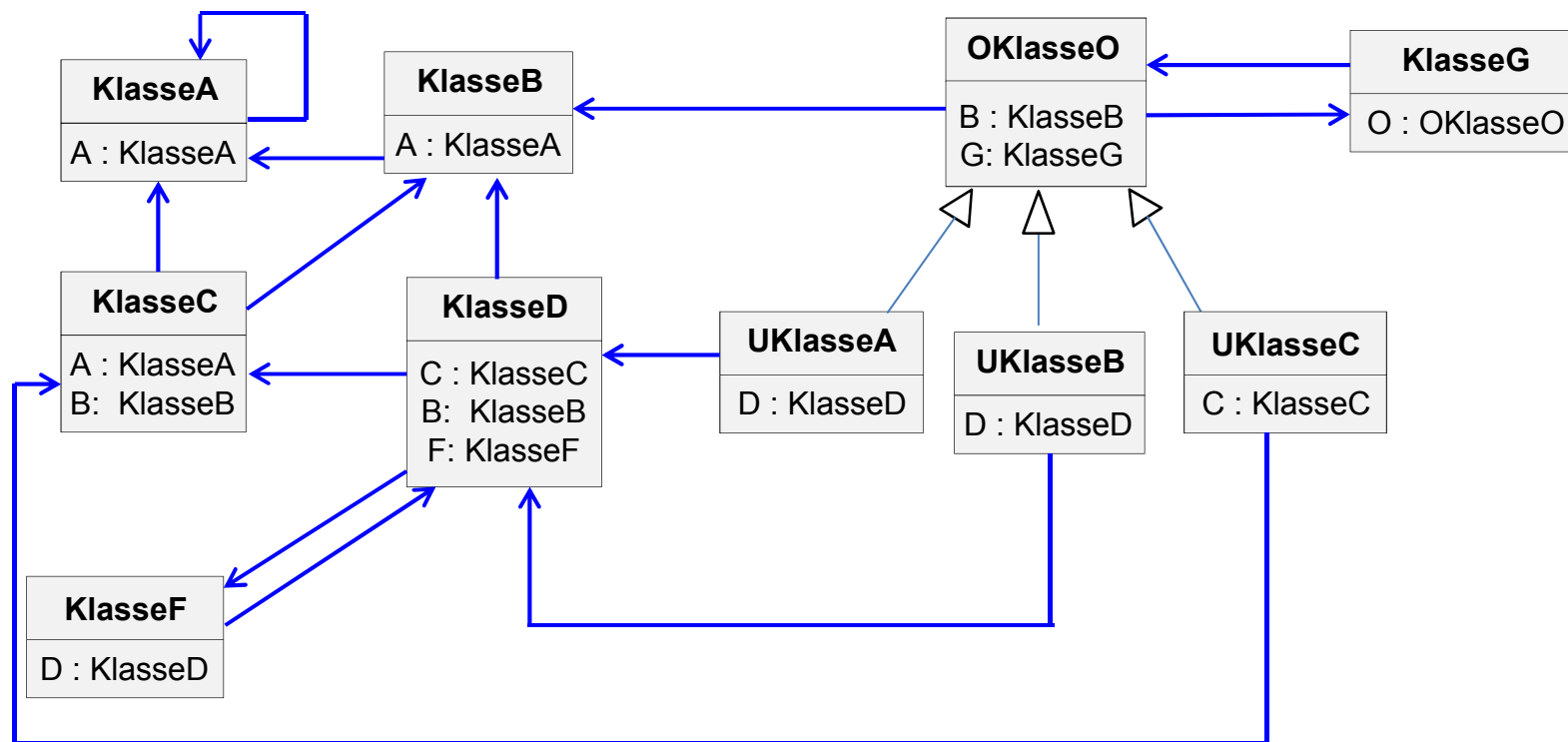
# Assoziationen und Rollen? Reichen nicht Attributnamen?

## Klassendiagramm nur mit Attributdarstellung



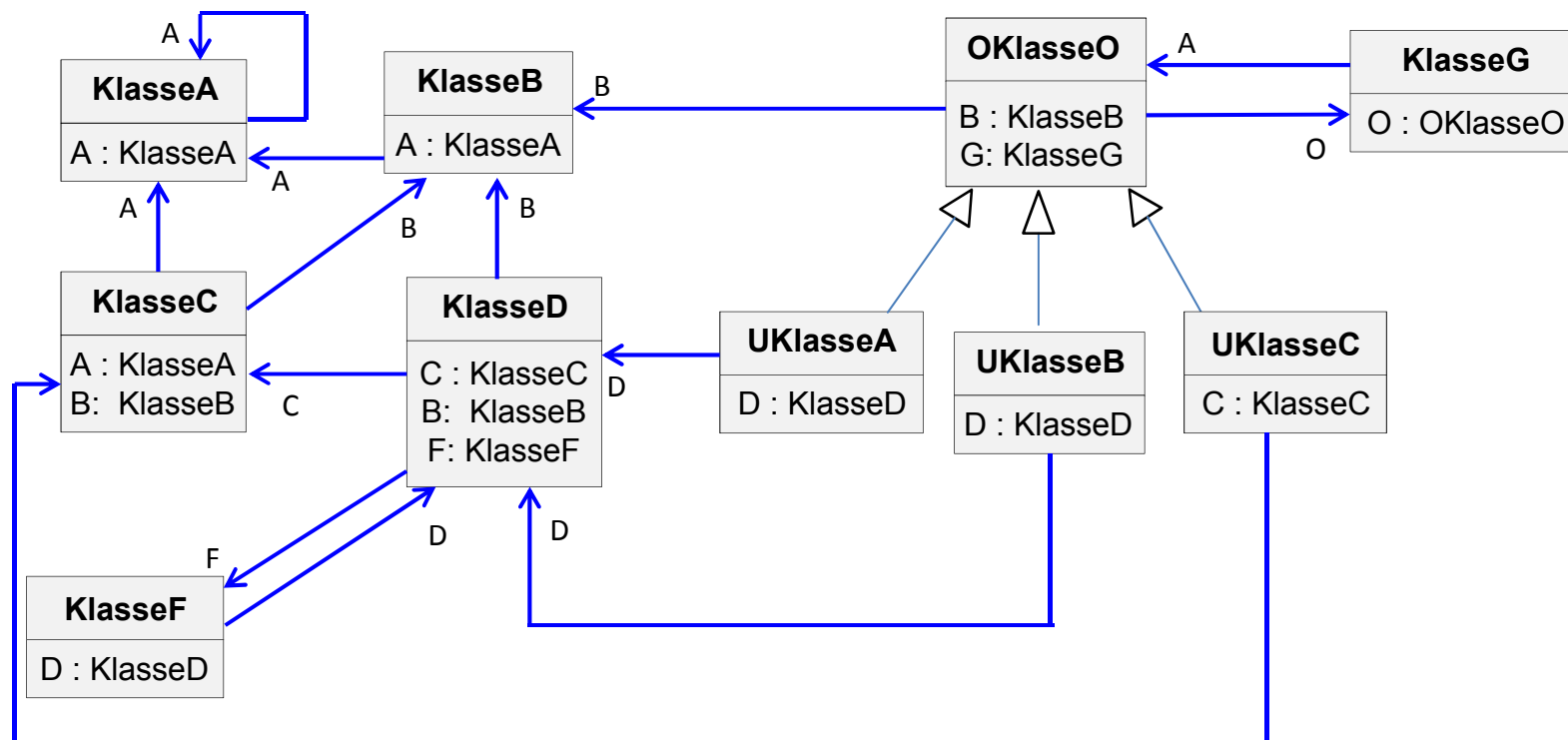
# Warum Assoziationen, reichen nicht Attributnamen?

## Klassendiagramm mit „zusätzlicher“ Referenzdarstellung



# Warum Assoziationen, reichen nicht Attributnamen?

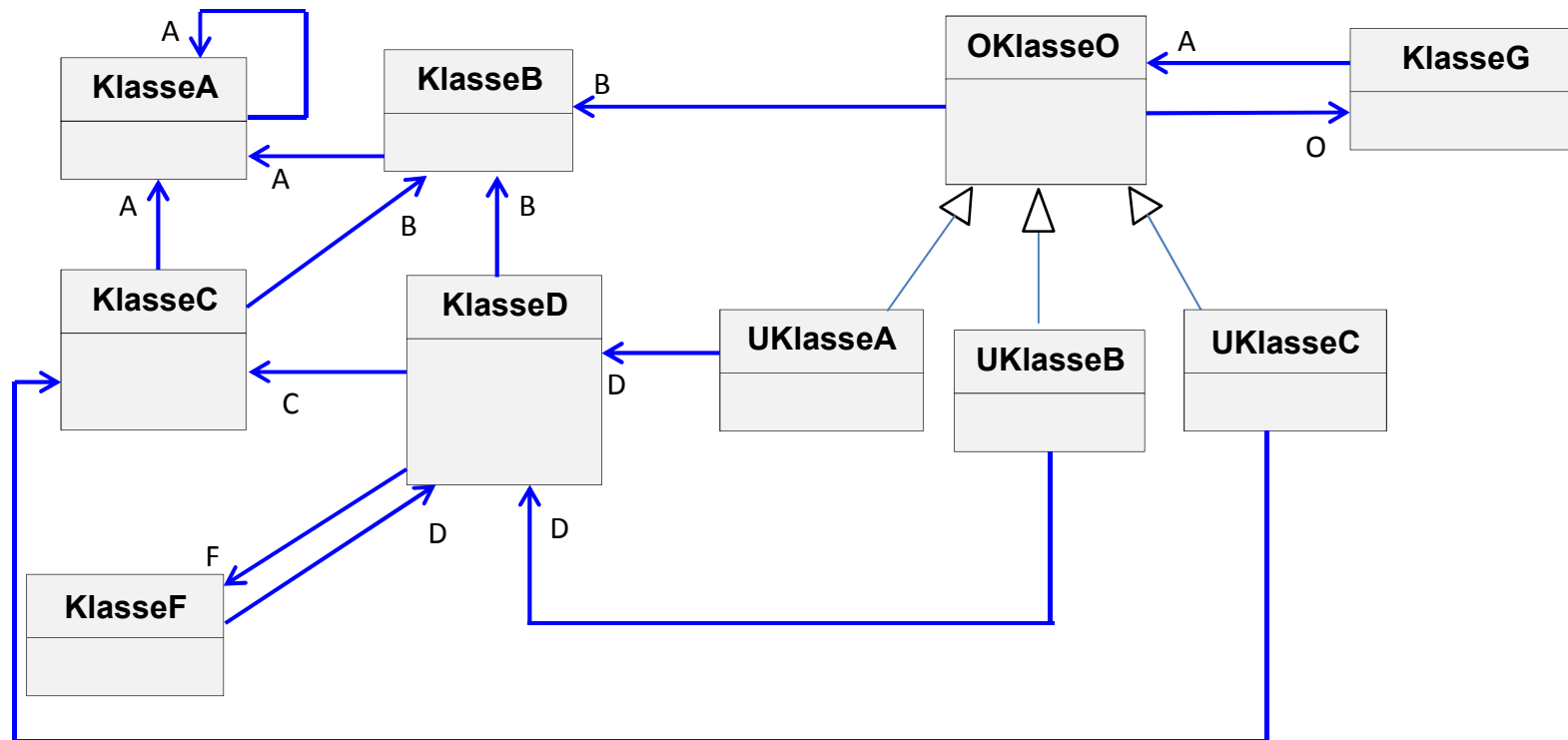
## Klassendiagramm mit Referenzdarstellung **plus** Rollennamen



# Warum Assoziationen, reichen nicht Attributnamen?

## Klassendiagramm mit Referenzdarstellung plus Rollennamen

→ ersetzt Attributdarstellung

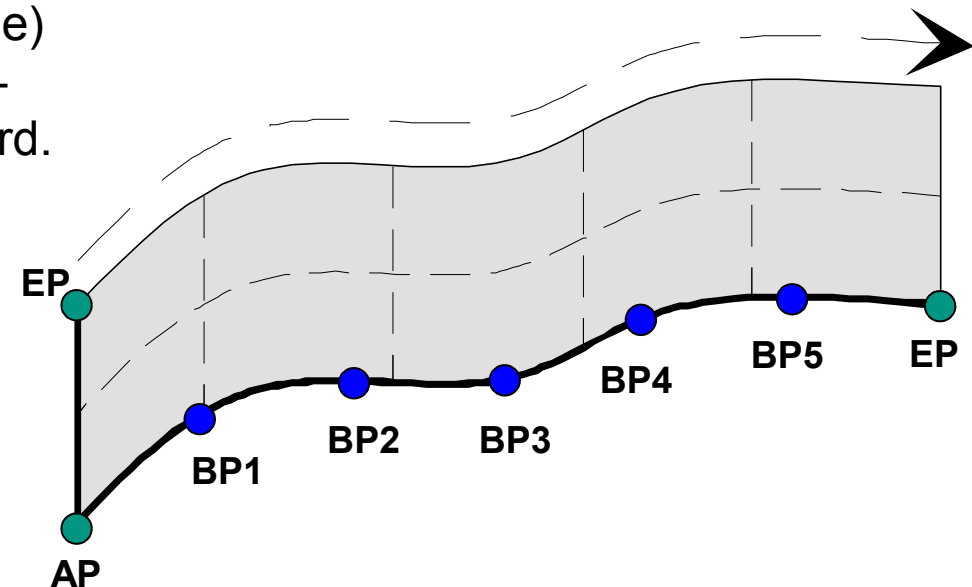


# Übungsaufgaben

## Aufgabe 3: Klassendiagramm für Translationsfläche

**Flächen** können auf unterschiedliche Art dargestellt werden.  
Erstellen Sie für jede der folgenden Darstellungen ein Klassen- **und** ein Instanzendiagramm:

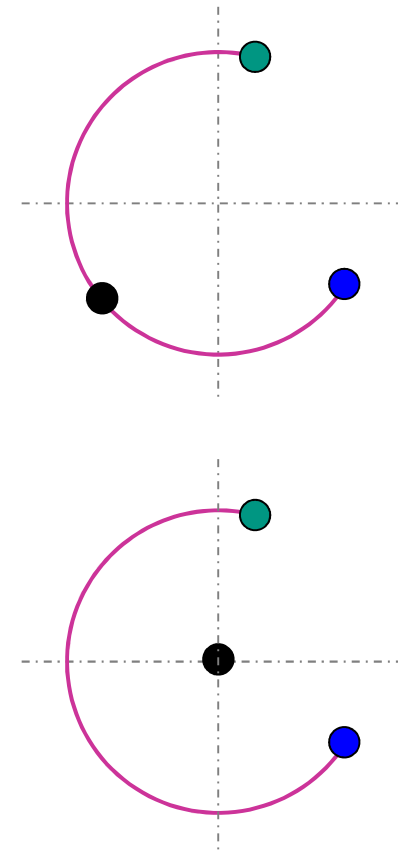
- Eine erste Fläche (Translationsfläche) ist definiert durch eine Linie, die entlang einer Raumkurve "gezogen" wird.
- Die Linie selbst verweist auf einen Anfangs- und einen Endpunkt.
- Die Raumkurve verweist ebenfalls auf einen Anfangs- und einen Endpunkt sowie auf eine Liste von (z.B. 5) Basispunkten.



## Aufgabe 4: Klassendiagramme für Kreisbögen

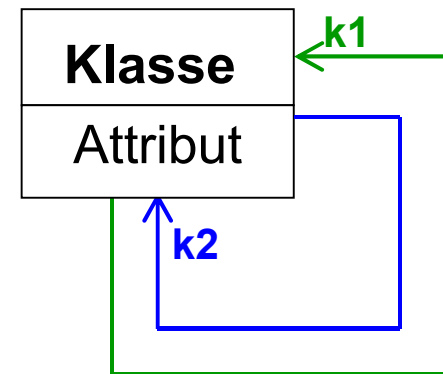
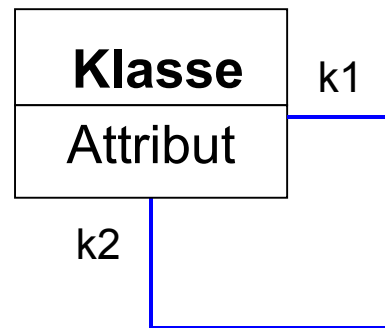
**Kreisbögen** können auf unterschiedliche Art dargestellt werden. Erstellen Sie für jede der folgenden Darstellungen ein Klassen- **und** ein Instanzendiagramm:

- Ein Kreisbogen ist definiert durch **drei** Punkte (Anfangs- und Endpunkt sowie ein dazwischen liegender Punkt auf dem Kreisumfang).
- Ein weiterer Kreisbogen ist ebenfalls definiert durch **drei** Punkte (Anfangs- End- und Mittelpunkt).

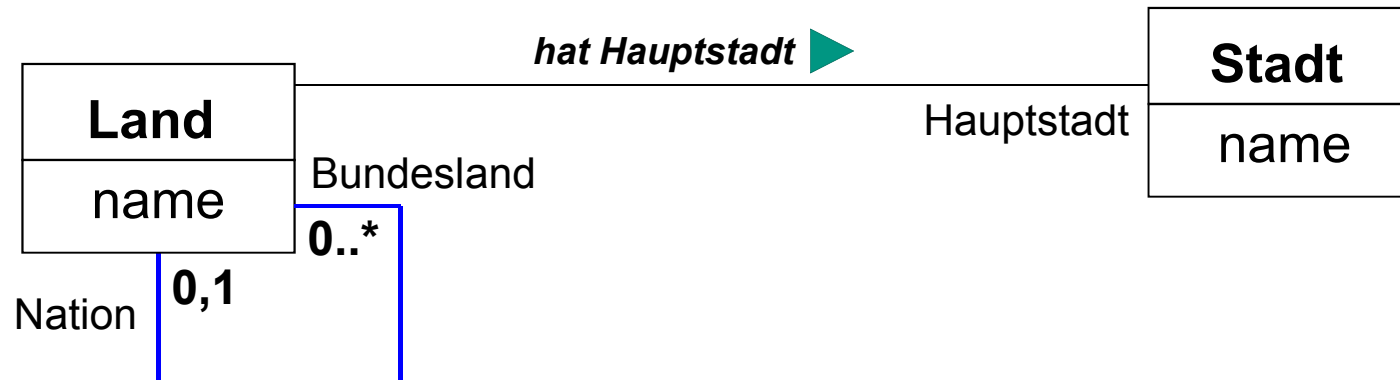




# Reflexive Assoziationen



# Reflexive Assoziationen: Mapping zu Java-Code



```
public class Land{  
  
    private List<Land> Bundeslaender;  
    private Land Nation;  
  
    private Stadt Hauptstadt;  
    . . .  
}
```

```
public class Stadt{  
  
    private Land land;  
    . . .  
}
```

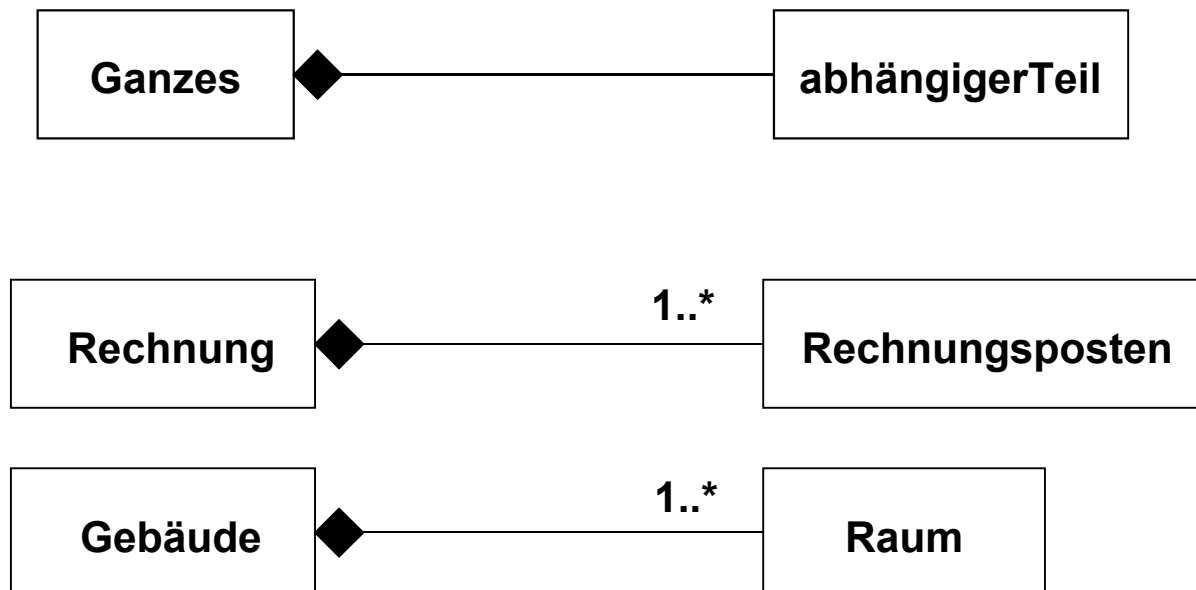
## Aggregation

- Assoziation, bei der hervorgehoben werden soll, dass zwischen den beteiligten Klassen eine Beziehung besteht, die durch „ist Teil von“ oder „besteht aus“ beschrieben werden kann.



# Komposition

- Eine Komposition ist eine besondere Form der Aggregation.
- Sie liegt vor, wenn die Einzelteile vom Aggregat (vom Ganzen) zusätzlich existenzabhängig sind.
- Sie sind durch eine gefüllte Raute gekennzeichnet:



# Aggregationen und Kompositionen im Java-Code

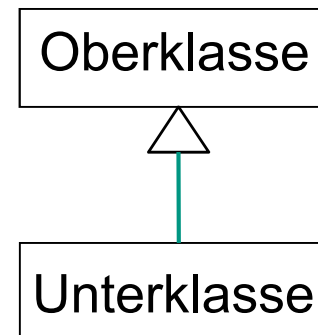
- Aggregation und Komposition werden **identisch** definiert (evtl. wird vom Modellierungstool ein Kommentar erzeugt)
- Existenzabhängigkeiten bei Kompositionen:
  - kaskadierendes Löschen (explizit oder mit Annotationen (z.B. JPA))

```
public class Gebaeude{  
  
    // besteht aus  
    private List<Raum> Raeume;  
    . . .  
}
```

## Vererbung (1)

Modellierungsspezifische Erweiterungen zum Thema Vererbung in „Grundlagen der Objektorientierung“:

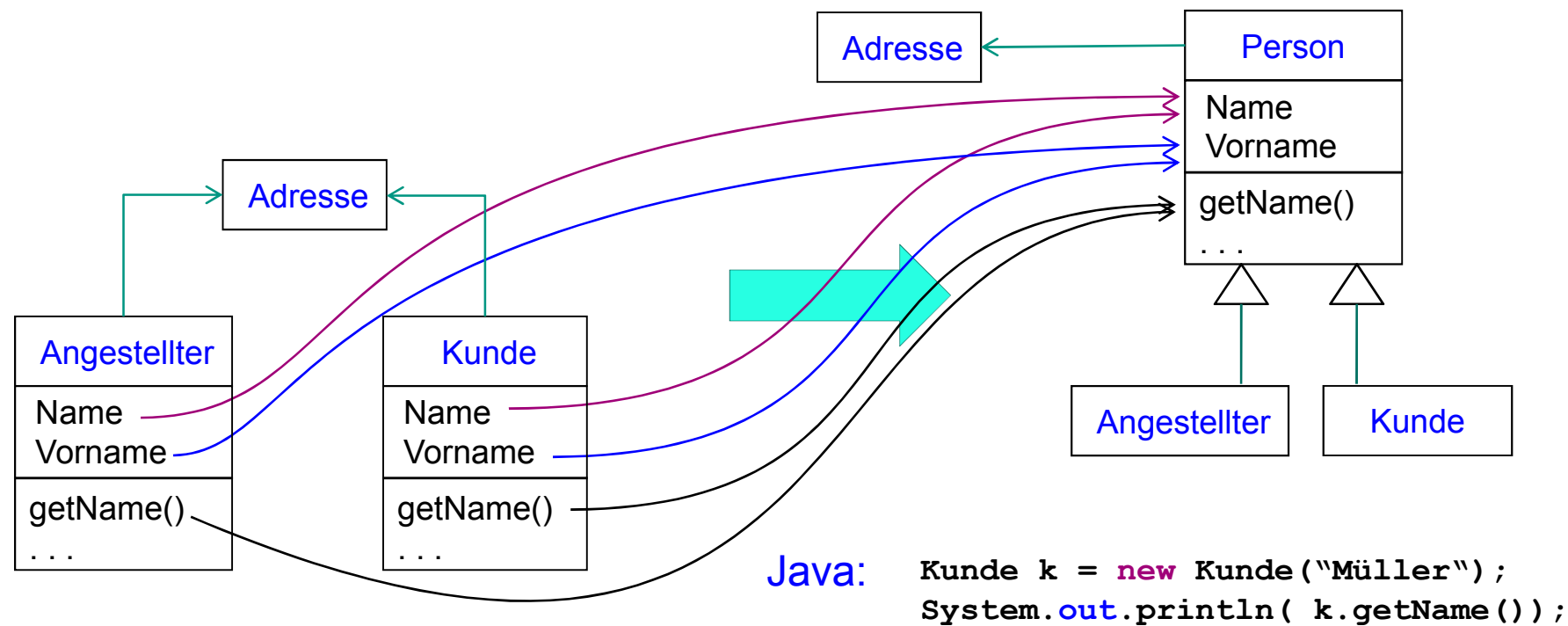
- Vererbung bzw. Generalisierung gelten über eine beliebige Zahl von Ebenen hinweg.
- Unterklasse kann durch „*ist ein ...*“ beschrieben werden
- UML-Darstellung:  
Pfeil mit Dreieck an der Spitze



## Vererbung (2)

Modellierungsspezifische Erweiterungen zum Thema Vererbung aus „Grundlagen der Objektorientierung“:

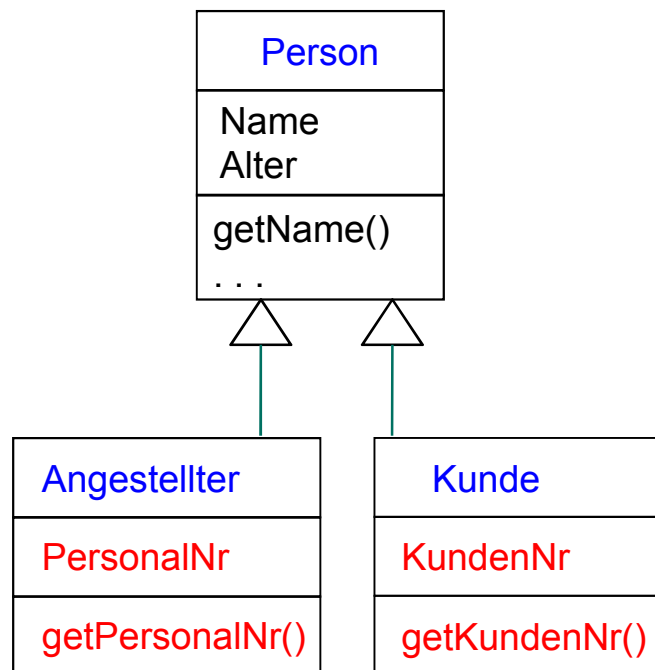
- Operationen, Attribute und Referenzen, die für eine Gruppe von Unterklassen gelten, werden der Oberklasse zugewiesen und von den einzelnen Unterklassen gemeinsam genutzt



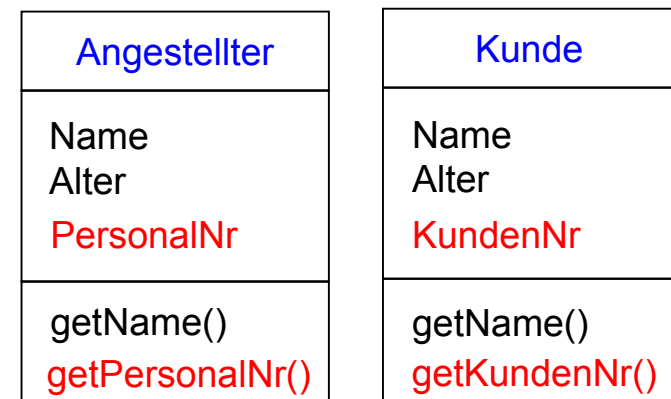
## Vererbung (3)

Modellierungsspezifische Erweiterungen zum Thema Vererbung in „Grundlagen der Objektorientierung“:

- Jede Unterklasse erbt alle Merkmale ihrer Oberklasse(n) und fügt ihre eigenen individuellen Eigenschaften und Methoden hinzu



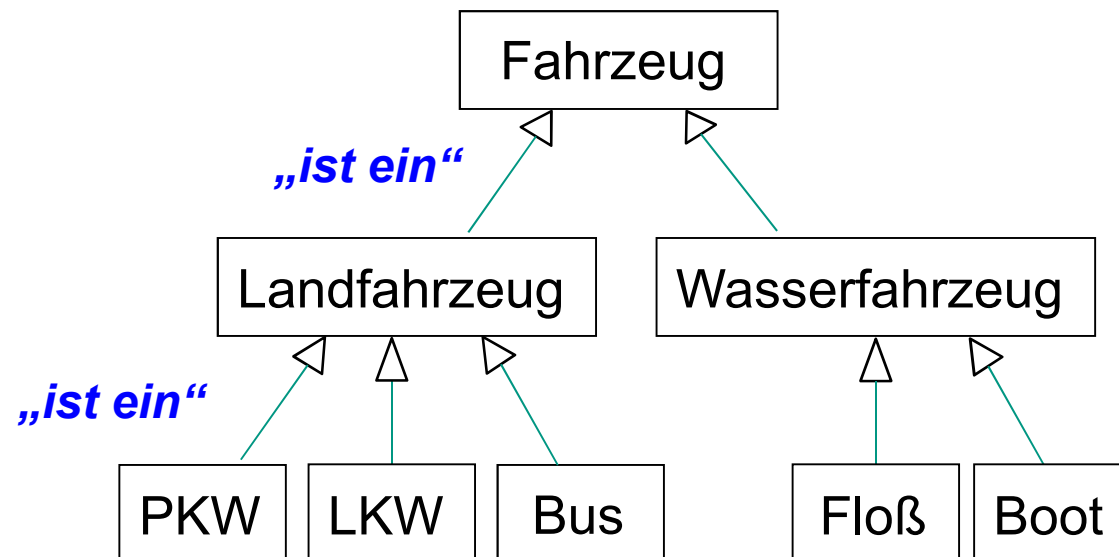
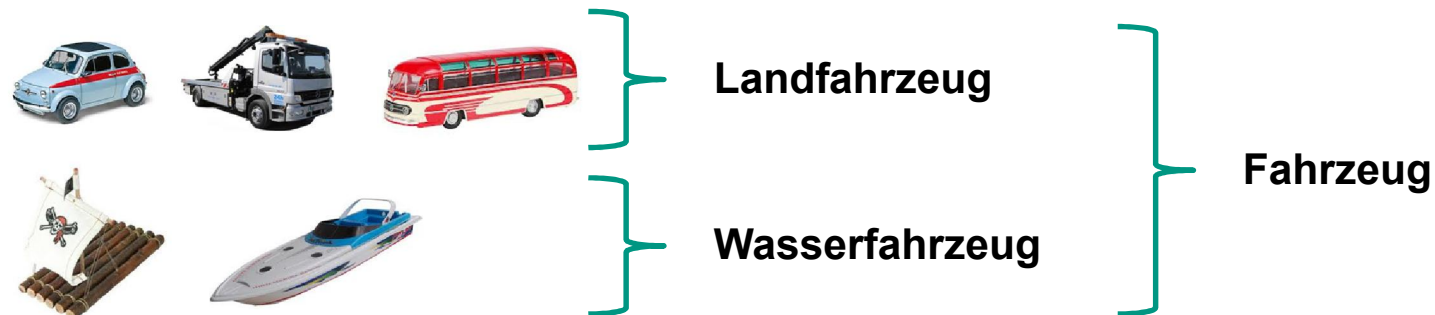
Ohne Oberklasse:





## Beispiel für Vererbung:

- unsere bereits bekannten Fahrzeuge



## Beispiel für Vererbung (Klassendiagramm zur Erinnerung)

| PKW                                                                 |
|---------------------------------------------------------------------|
| hoehe: int<br>breite: int<br>laenge: int<br>• • •<br>offen: boolean |
| fahren(): void<br>stehen(): void<br>• • •                           |



| LKW                                                                 |
|---------------------------------------------------------------------|
| hoehe: int<br>breite: int<br>laenge: int<br>• • •<br>offen: boolean |
| fahren(): void<br>stehen(): void<br>• • •                           |

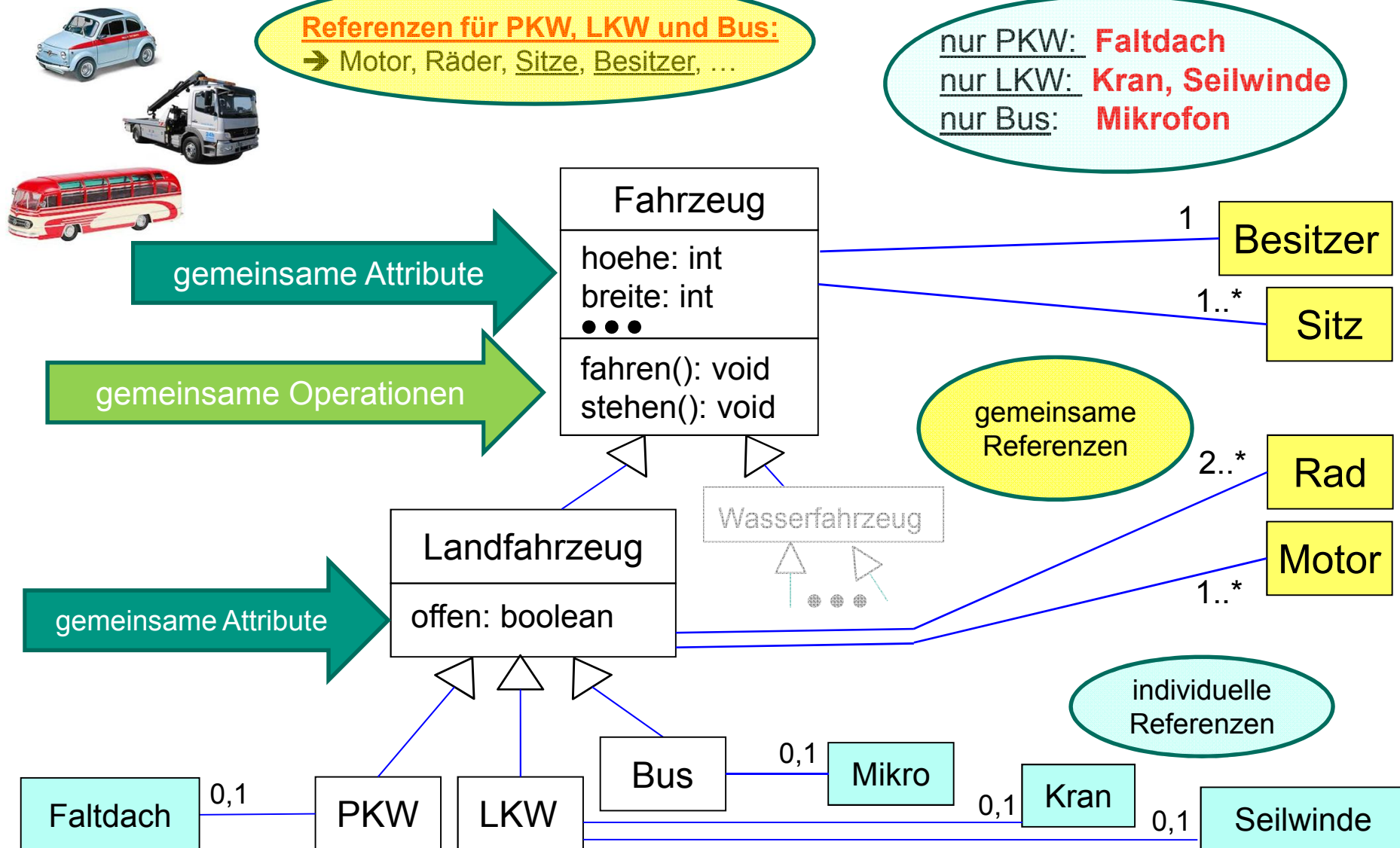


| Bus                                                                 |
|---------------------------------------------------------------------|
| hoehe: int<br>breite: int<br>laenge: int<br>• • •<br>offen: boolean |
| fahren(): void<br>stehen(): void<br>• • •                           |

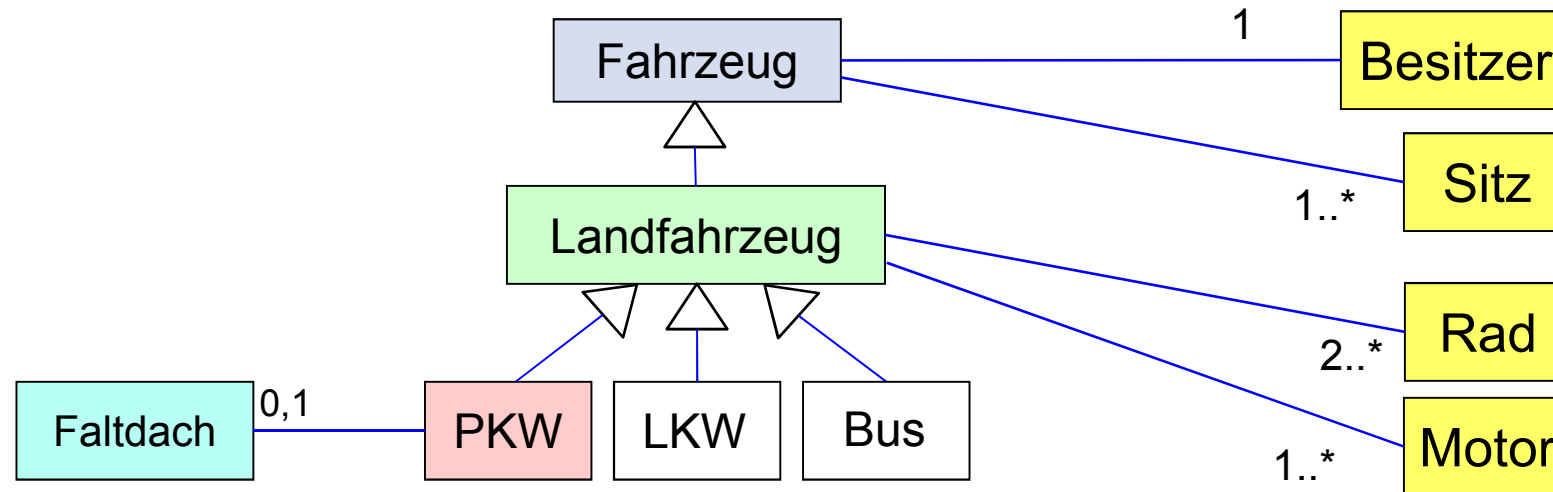


➔ **lauter identische Attribute und Operationen!**

# Beispiel für Vererbung (Forts. für Landfahrzeuge)



# Vererbung: Mapping zu Java-Code



```
public class Fahrzeug{
    private Besitzer besitzer;
    private List<Sitz> sitze;
    . . .
}
```

```
public class PKW extends Fahrzeug{
    private Faltdach faltdach;
    . . .
}
```

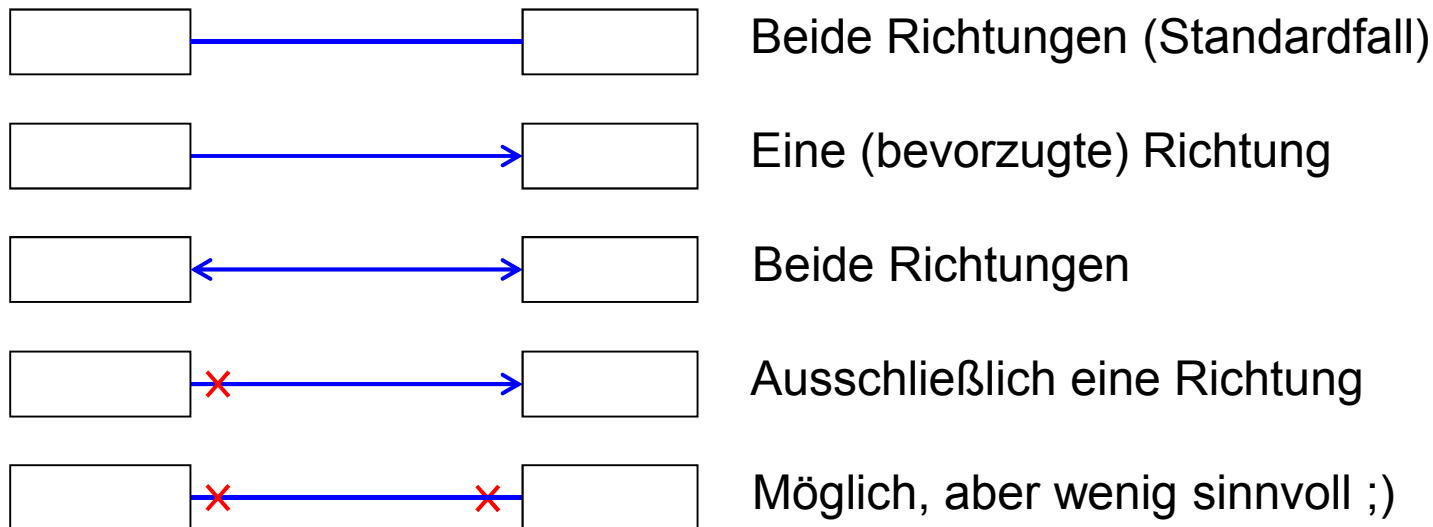
```
public class Landfahrzeug
    extends Fahrzeug{
    private List<Rad> raeder;
    private List<Motor> motoren;
    . . .
}
```

## Aufgabe 5: Grafikeditor (mit Vererbung)

Zeichnen Sie ein Klassendiagramm für einen Graphikeditor, der das Konzept der Gruppierung unterstützt. Folgendes sei angenommen:

- Ein Graphikdokument enthält beliebig viele Blätter.
- Jedes Blatt enthält Grafikobjekte (z.B. Text, geometrische Objekte sowie Gruppen (→ d.h. eine Gruppe ist hier ein Grafikobjekt!)).
- Geometrische Objekte sind Kreise, Ellipsen, Rechtecke, Linien und Quadrate.
- Eine Gruppe ist einfach eine Menge von Grafikobjekten, die ihrerseits Gruppen enthalten kann.
- Eine Gruppe muss mindestens zwei Grafikobjekte enthalten.
- Ein Grafikobjekt kann ein direktes Mitglied von höchstens einer Gruppe sein.

Darstellung durch Pfeilspitze und Kreuz (einfaches Linienende: *unspezifiziert*)



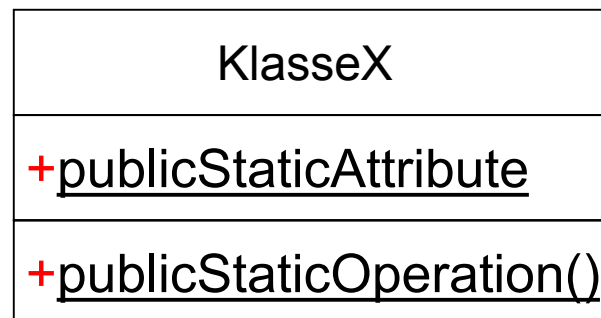
- Zusätzlich können an beiden (!) Enden Multiplizitäten eingetragen werden.
- Modelle möglich, die so nicht sinnvoll implementiert werden können / müssen
- IDEs: Details für Quellcode-Generierung müssen dort bestimmt werden

# Sichtbarkeit und Eigenschaften von Attributen und Methoden

Sichtbarkeitszeichen vor den Namen des Attributs bzw. der Operation

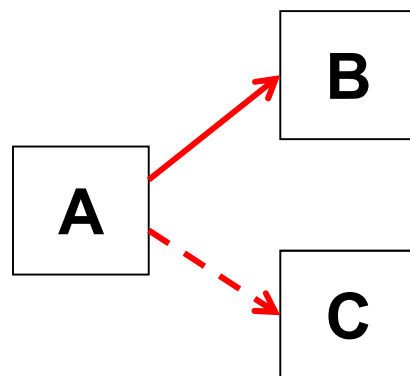
- |             |          |
|-------------|----------|
| - private   | -iD      |
| + public    | +name    |
| # protected | #vorname |
| ~ package   | ~nochWas |

Statische Attribute bzw. Operationen werden unterstrichen:



## Abhängigkeit vs. Assoziation

- Klasse **B** im Quellcode von **A** als Klassenattribut bzw. Klassenvariable modelliert: **Referenz** (Assoziation, Aggregation oder Komposition)
- Klasse **C** im Quellcode von **A** nur „kurzfristig“ in einer Methode oder als Rückgabeobjekt verwendet: **Abhängigkeit** (gestrichelter Pfeil)
- Abhängigkeiten sollen aufzeigen, welche Klassen von anderen Klassen benutzt werden => vollständige Übersicht möglich (Analyse, Entwurf)



```

public class A{
    B bb;          => Klassenvariable (Referenz)

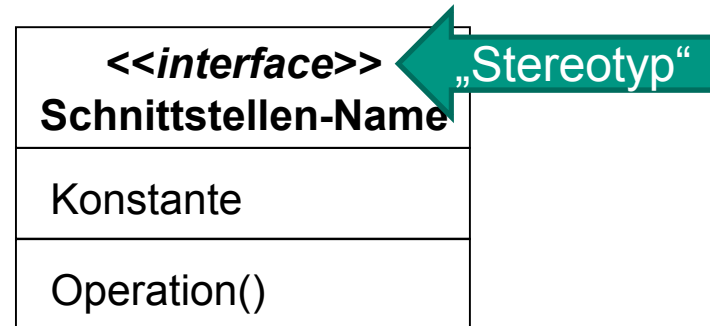
    public A( B b {
        this.bb = b;
        . . .
    }
    public C aMethod( C c ){
        C cc = new C();
        . . .
    }
}
  
```

} => Abhängigkeit

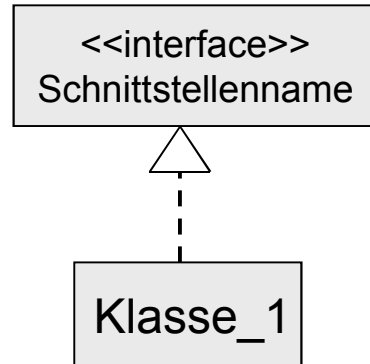


# Schnittstellen (Interfaces)

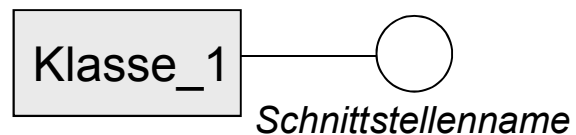
Notation:



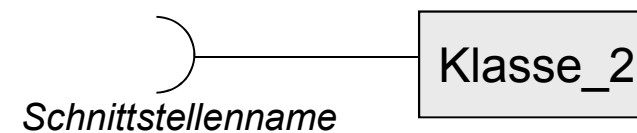
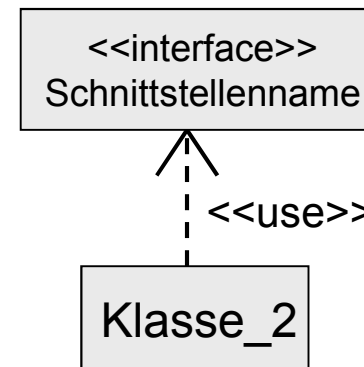
Implementierung:



alternativ:

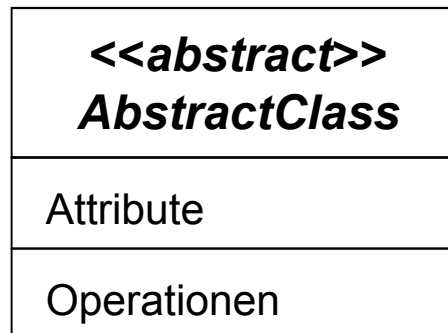


Verwendung:



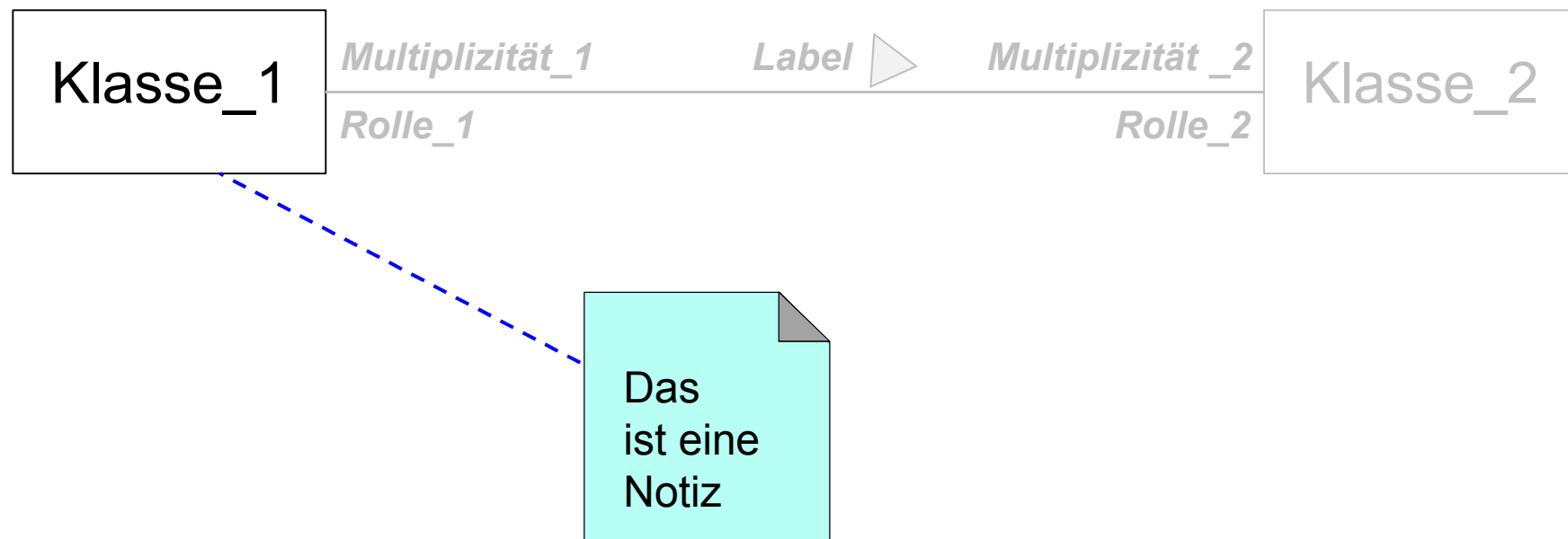
# Abstrakte Klassen

- Können (dürfen) nicht instanziiert werden
- Können Methoden implementieren und den Unterklassen zur Verfügung stellen
- Können (ebenfalls abstrakte) Methoden deklarieren, die von den Unterklassen implementiert werden **müssen** (analog zu Interfaces)
- **UML:** Klassensymbol mit Stereotyp `<<abstract>>` und/oder kursiv geschriebenem Klassennamen



## Kommentar, Notiz

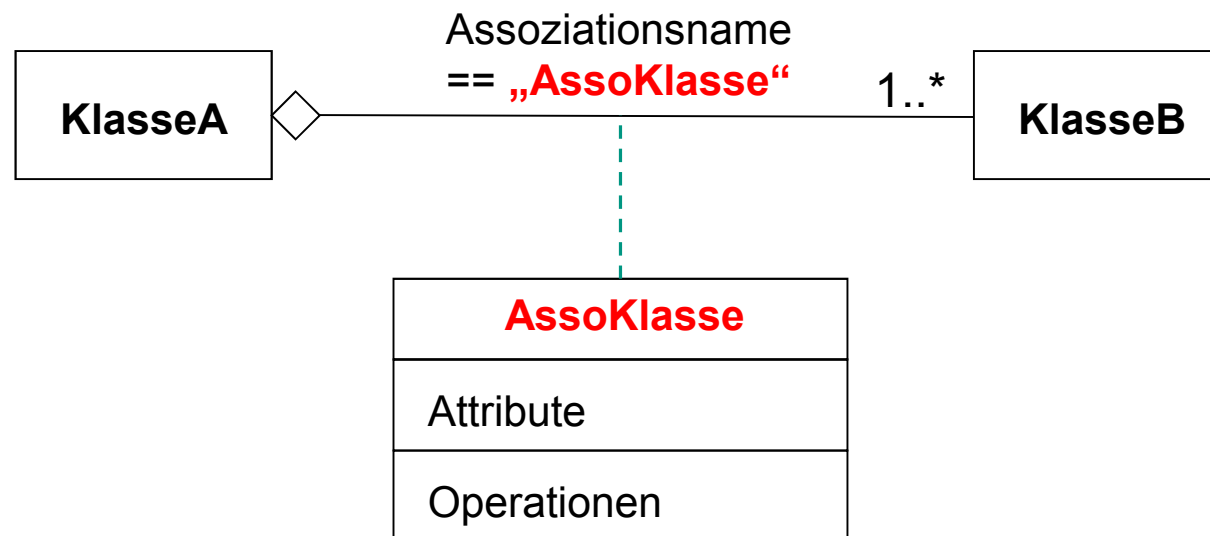
- Kommentare bzw. Notizen können mithilfe des Notiz-Symbols in das Diagramm gezeichnet werden
- Das Notizsymbol kann durch eine gestrichelte Linie mit einem beliebigen Diagrammelement verbunden werden



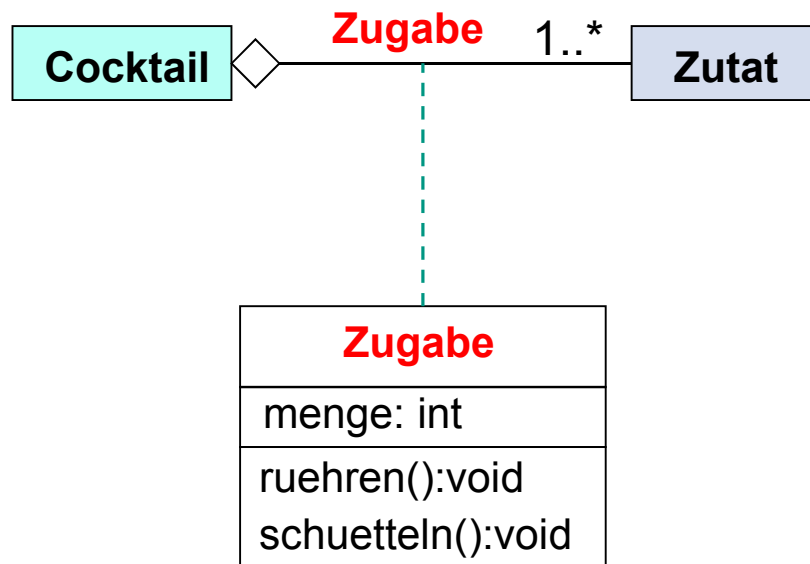
# Besonderheiten bei UML-Klassendiagrammen

# Assoziationsklassen

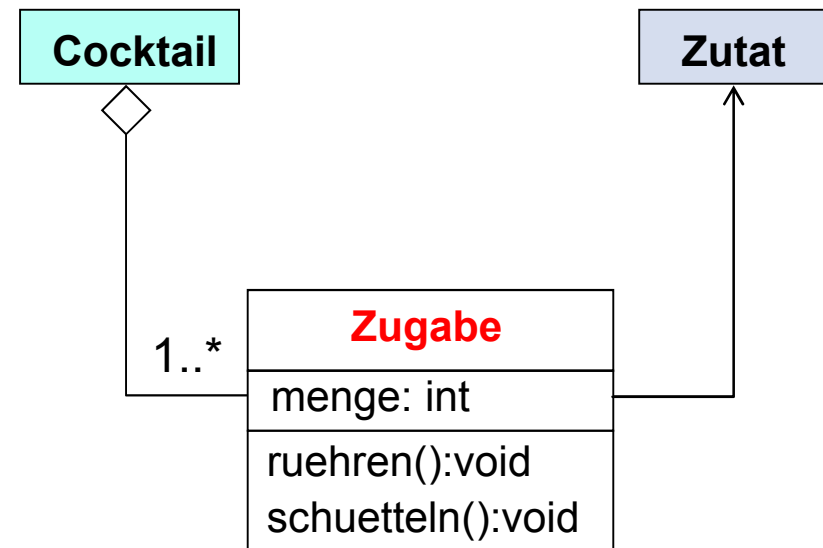
- Vereint die Eigenschaften der Klasse und der Assoziation in sich
- Sind mit einer Assoziation verbunden
- Dient dazu, Eigenschaften näher zu beschreiben, die keiner der beiden beteiligten Klassen sinnvoll zuzuordnen sind



- Beispiel (aus „UML 2 glasklar“) Modell:



- Nach Code-Generierung (VisualParadigm™, 2014):



# Generalisierungsmengen und -eigenschaften

- Bei Vererbungen/Generalisierungen können zusätzlich Bedingungen angegeben werden:

## Eigenschaftswerte:

### complete:

Vollständige Spezialisierung, keine Unterklassen mehr möglich / erlaubt

### incomplete:

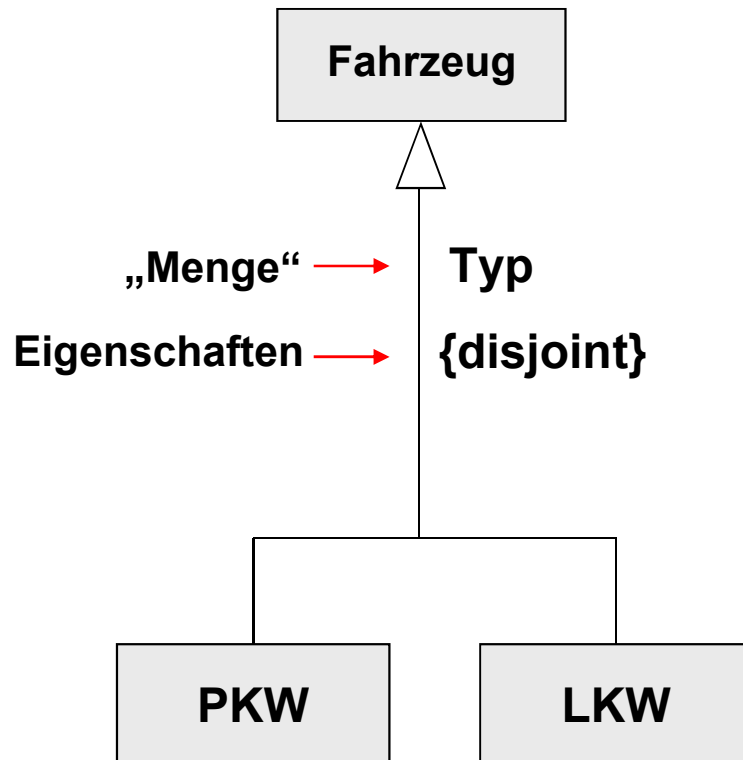
weitere Spezialisierungen möglich

### disjoint:

Unterklassen werden klar voneinander unterschieden

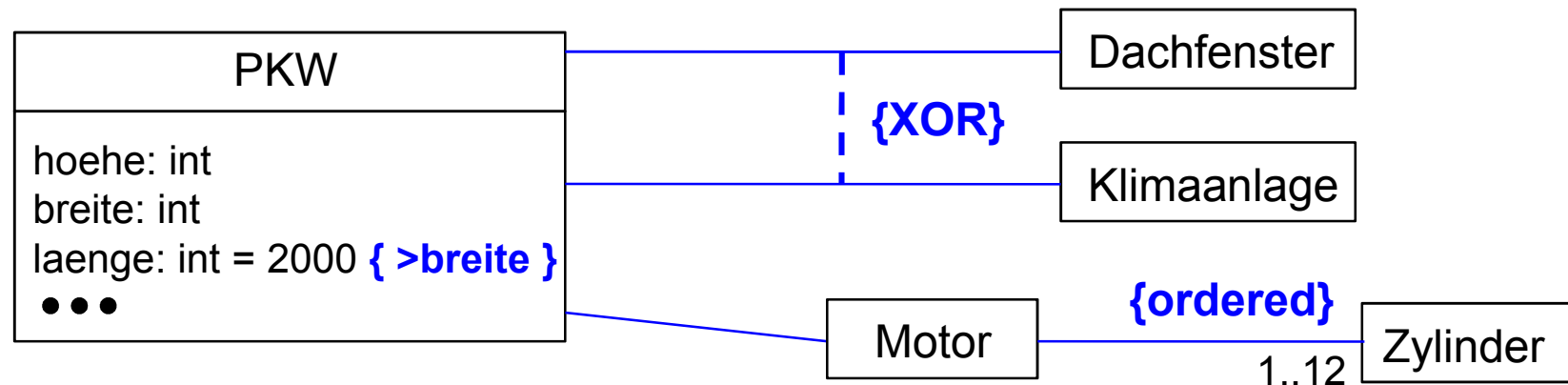
### overlapping:

Eine nachfolgende Klasse kann Unterklasse von mehreren der mit *overlapping* gekennzeichneten Unterklassen sein



## Einschränkungen (*Constraints*)

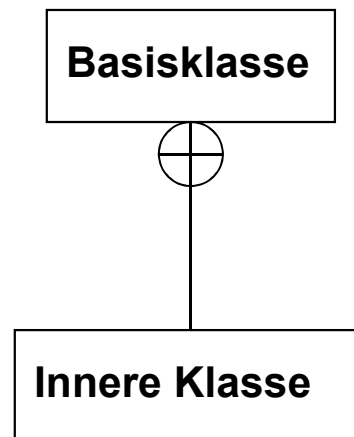
- Oft auch *Zusicherungen* genannt
- Bedingungen, die für ein Element erfüllt sein müssen. Es kann sich dabei um einen OCL-Ausdruck handeln (*Object Constraint Language*), zugelassen ist aber auch ein sprachlicher Ausdruck.
- Constraints können an beliebige Modellelemente angefügt werden (Attribute, Operationen, Klassen, Assoziationen, ...)
- Sie werden zwischen geschweiften Klammern geschrieben
- Beispiele:





## Innere Klassen

- Innere Klassen besitzen denselben Namensraum wie ihre Elternklasse
- Sie werden bei Java innerhalb der Klassendeklaration definiert



- Mario Jeckle: UML 2.0, Die neue Version der Standardmodellierungssprache;  
<http://www.jeckle.de/files/umltutorial.pdf>
- UML 2 glasklar  
Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, Stefan Queins  
Hanser Verlag München Wien, 2004
- UML 2.0 in a Nutshell  
Dan Pilone, Neil Pitman  
O'Reilly Verlag, 2006