Karlsruher Institut für Technologie

DHBW
Duale Hochschule
Baden-Württemberg

Software Engineering I

Grundlagen der objektorientierten Datenmodellierung mit UML
Instanzen- und Klassendiagramme

Institut für Angewandte Informatik (IAI)

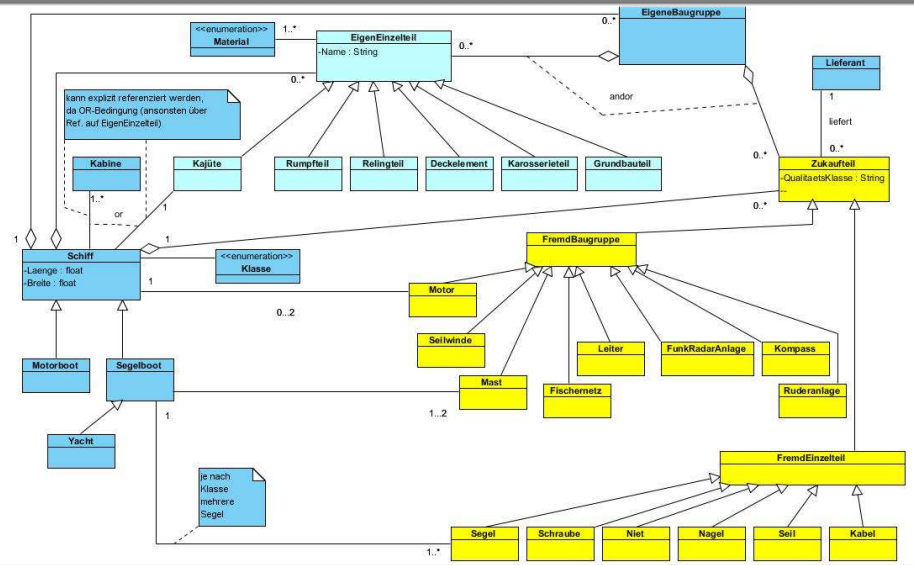


Diagramme der Unified Modeling Language (UML)

Karlsruher Institut für Technologie

Die meistgenutzten und wichtigsten Diagramme der UML 2

- Strukturdiagramme
  - Objekt- bzw. Instanzendiagramm
  - Klassendiagramm
  - Paketdiagramm
- Verhaltensdiagramme
  - Use-Case-Diagramm (Anwendungsfalldiagramm)
  - Sequenzdiagramm
  - Aktivitätsdiagramm
  - Zustandsdiagramm

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

Karlsruher Institut für Technologie

- Objekte (alternativ: Instanzen):
  - reale Elemente in Alltag, Problemumfeld und Programm
    ➔ Inhalt des Arbeitsspeichers
- 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 (Implementierung, Quellcode) einer Operation
  - Wird oft auch als alternativer Begriff für Operation verwendet

Darstellung von Klassen und Instanzen

Klasse

Instanz

UML-Notation für Instanznamen:

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

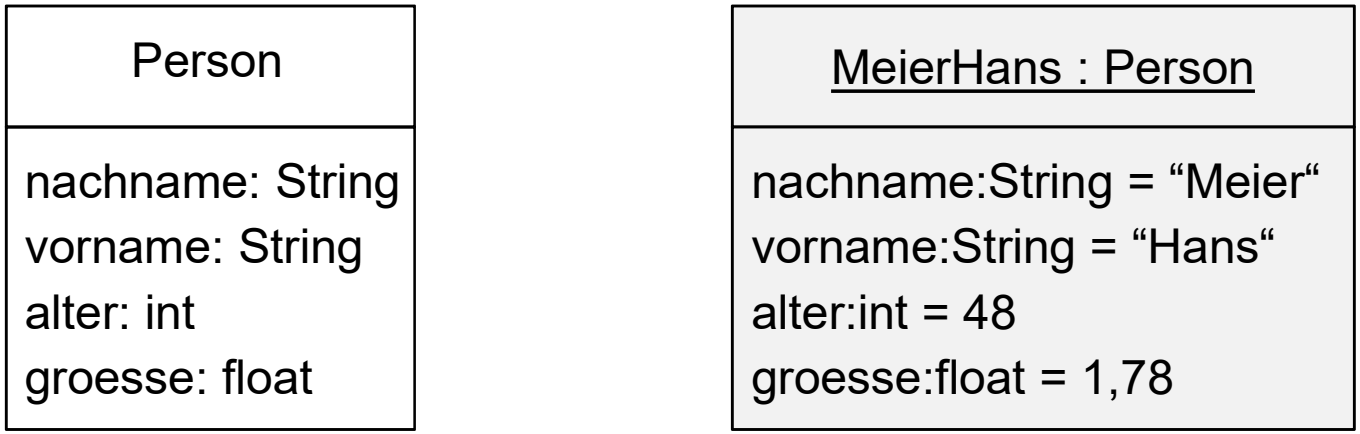
(Java):
```
Person MeierHans = new Person();
```

: Klasse bei einer anonymen Instanz nur Klassenname mit Doppelpunkt

Instanzname wenn nur der Instanzname ausreicht ()

Der Instanzname wird unterstrichen

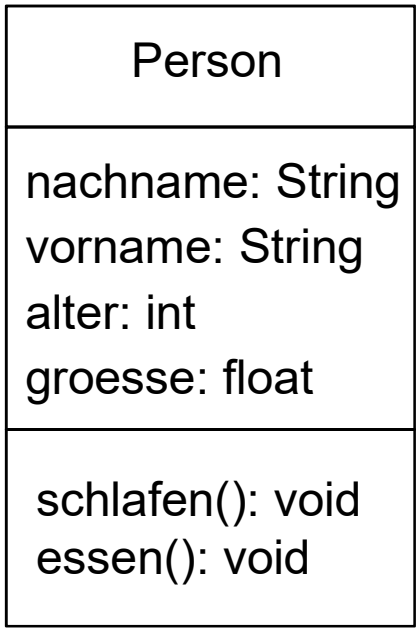
Darstellung von Klassen- und Instanz-Attributen



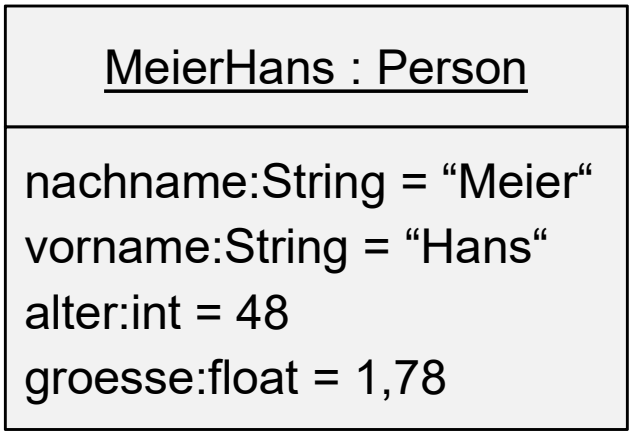
Klasse

Instanz

Darstellung von Klassen- und Instanz-Methoden



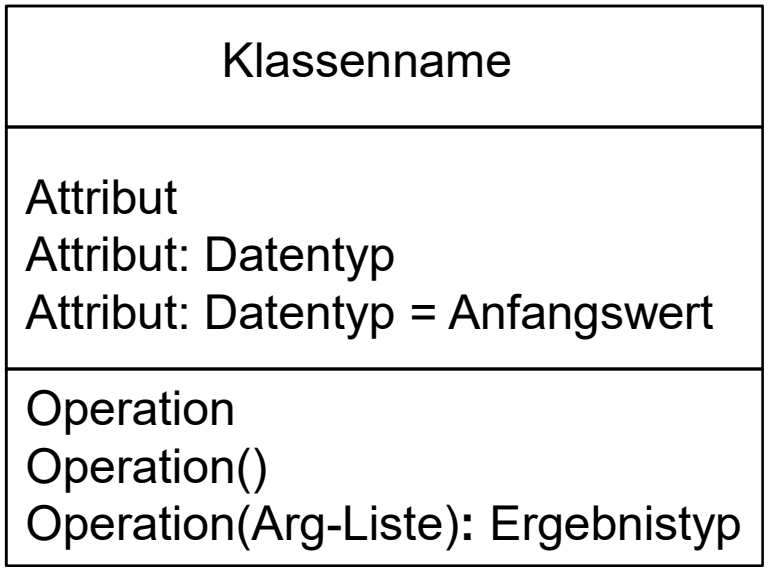
Klasse



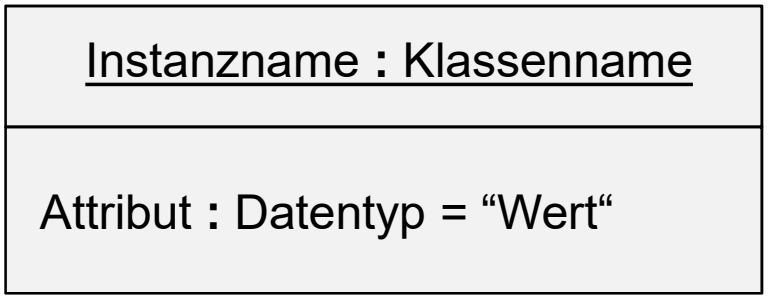
(keine Methoden)

Instanz

Klassen und Instanzen: allgemeine Darstellung

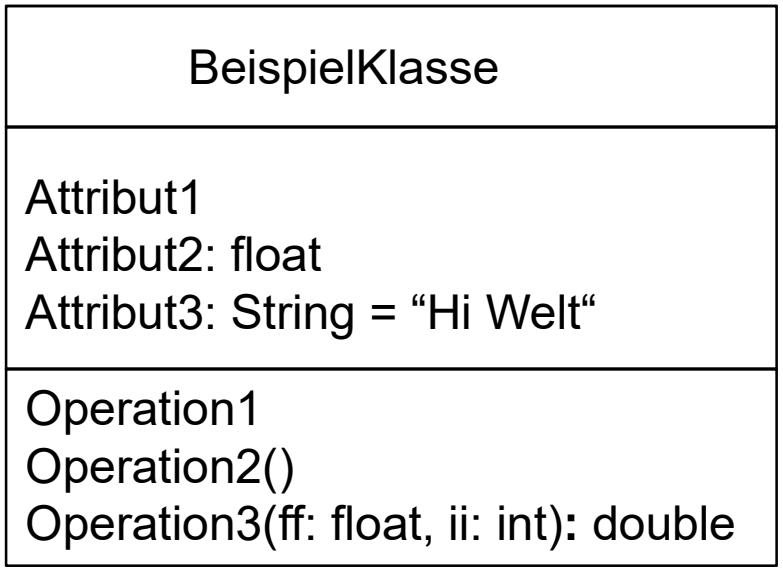


Klasse



Instanz

Klassen und Instanzen: Mapping zu Java-Code



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 Objekte und deren Attribute
  (S. Grundlagen der Objektorientierung)

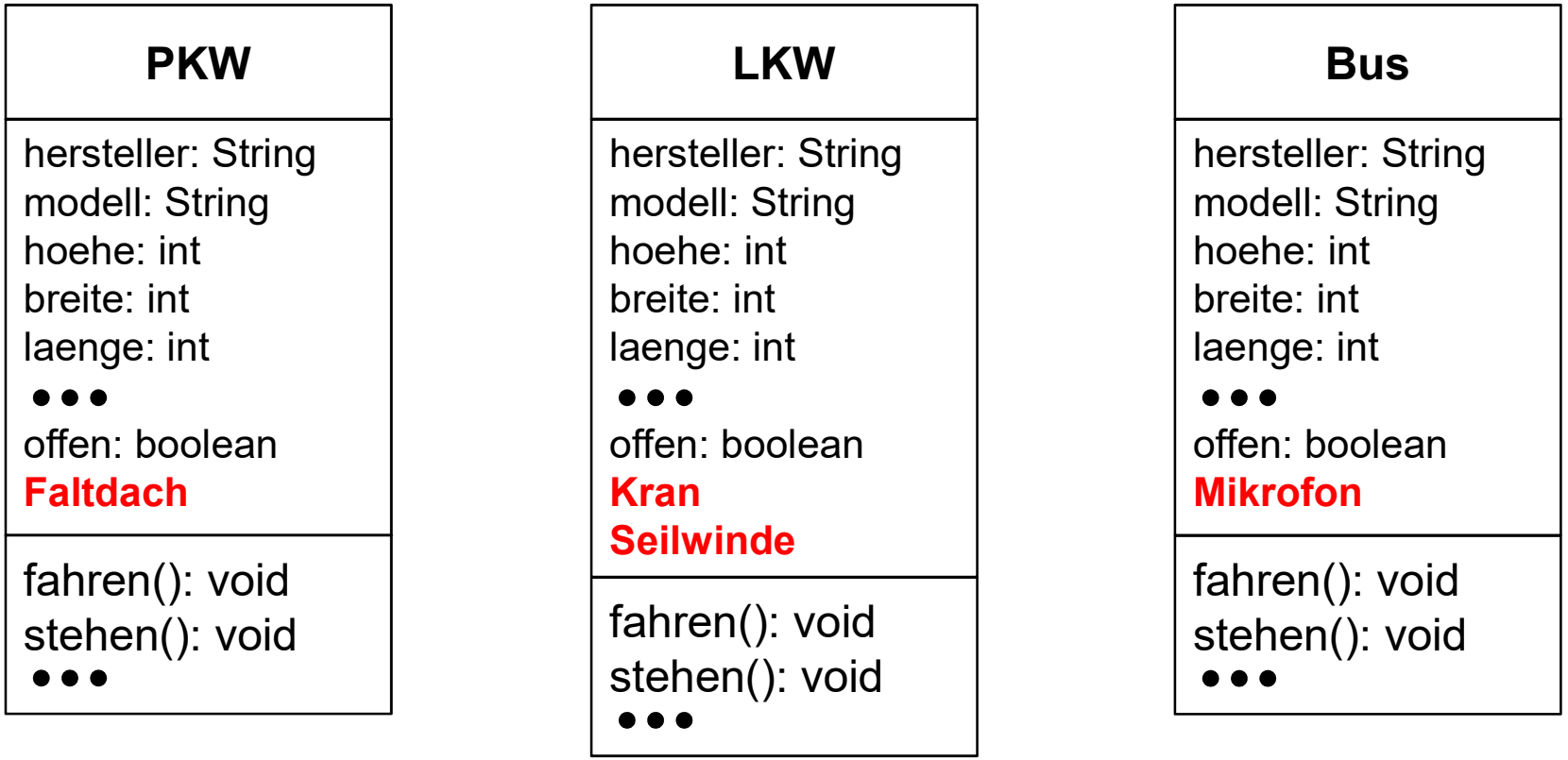
<u>PKW:</u>
Hersteller, Modell, Höhe, Breite, Länge, Leistung,
Verbrauch, Gewicht, Farbe, an/aus, offen/zu, Sitzplätze, …
fahren, stehen, tanken, beschleunigen, …
➔ Motor, Räder, Sitze, Besitzer, **Faltdach**, …

<u>LKW:</u>
Hersteller, Modell, Höhe, Breite, Länge, Leistung,
Verbrauch, Gewicht, Farbe, an/aus, offen/zu, Sitzplätze, …
fahren, stehen, tanken, beschleunigen, …
➔ Motor, Räder, Sitze, Besitzer, **Kran, Seilwinde,** …

<u>Bus:</u>
Hersteller, Modell, Höhe, Breite, Länge, Leistung,
Verbrauch, Gewicht, Farbe, an/aus, offen/zu, Sitzplätze, …
fahren, stehen, tanken, beschleunigen, …
➔ Motor, Räder, Sitze, Besitzer, **Mikrofon**, …

Beispiel zur Klassen-Darstellung

Karlsruher Institut für Technologie



Beispiel zur Instanzen-Darstellung

Fiat500 : PKW
hersteller = Fiat modell = 500 hoehe : int = 1440 breite: int = 1350 laenge: int = 3240 ●●● offen : boolean = true

DB-ASW : LKW
hersteller = M-Benz modell = ASW hoehe : int = 2950 breite : int = 2490 laenge : int = 6858 ●●● offen: boolean = false

DB-O321 : Bus
hersteller = M-Benz modell = O321 hoehe : int = 2800 breite : int = 2450 laenge : int = 11250 ●●● offen : boolean = false

Klassendiagramm

- Grafische Notation, um **Klassen** und deren Relationen zu anderen Klassen darzustellen.
- Schema, Muster oder Template (Schablone) zur Beschreibung vieler möglichen Objektinstanzen.
- Jede Klasse ist nur einmalig vorhanden.

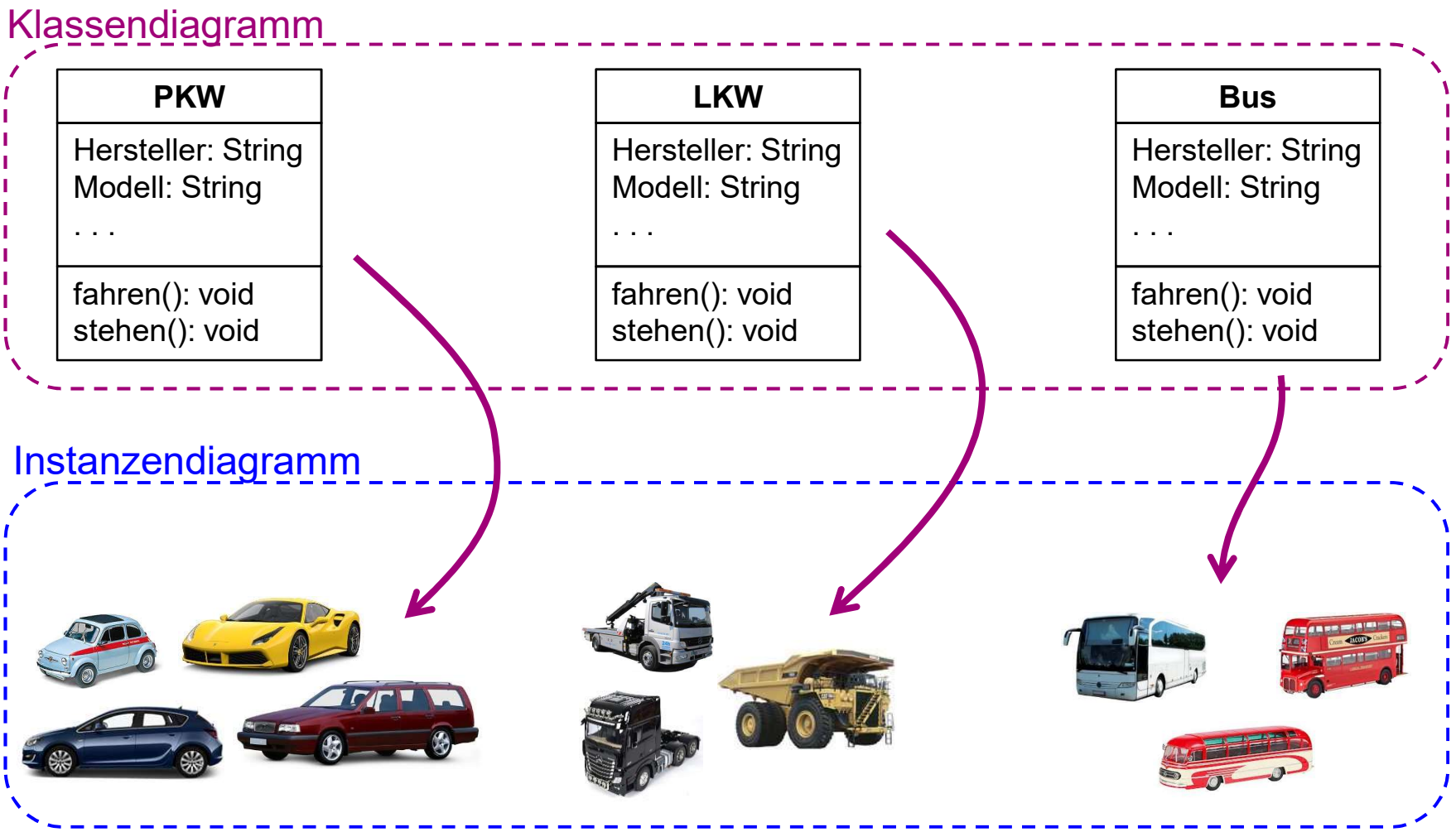
PKW
hersteller: String modell: String . . .
fahren(): void stehen(): void

LKW
hersteller: String modell: String . . .
fahren(): void stehen(): void

Bus
hersteller: String modell: String . . .
fahren(): void stehen(): void

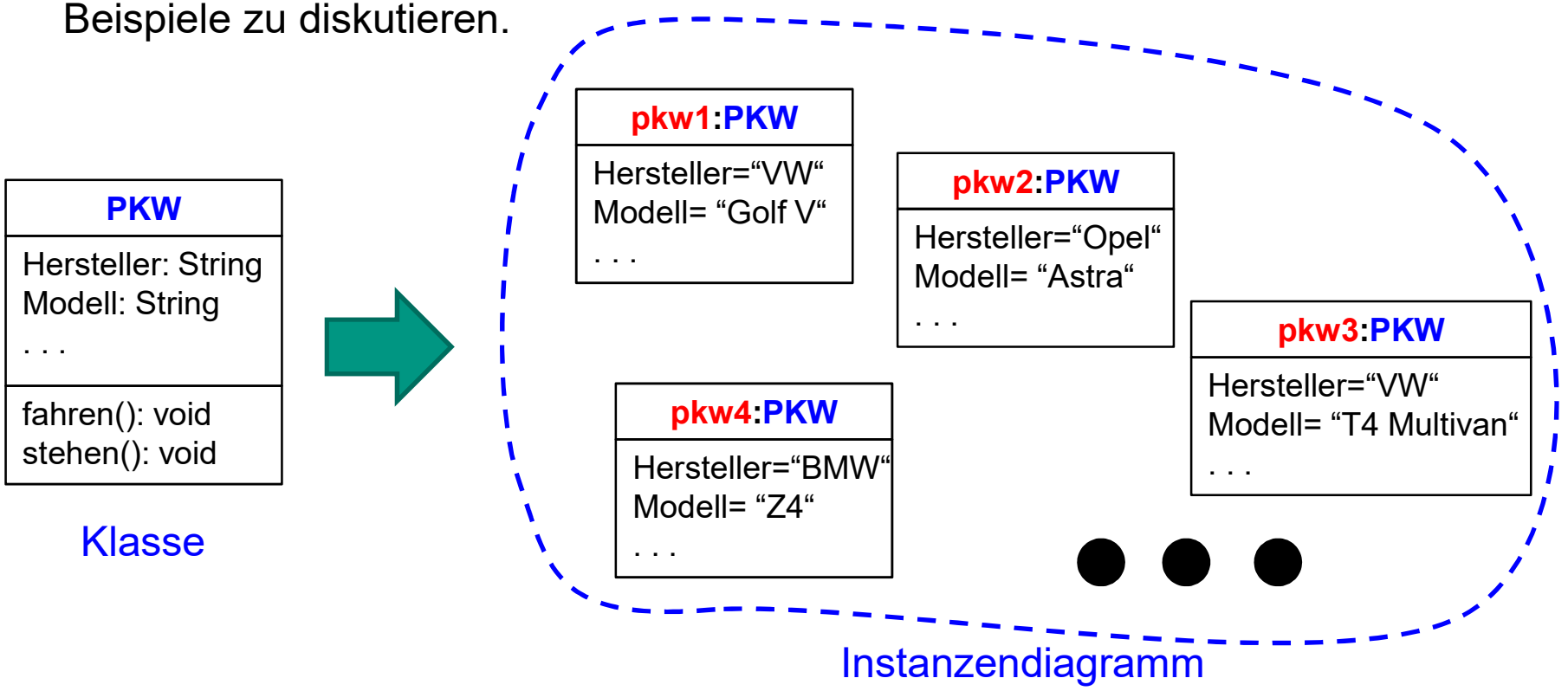
Klassendiagramm (2)

- Ein Klassendiagramm ist Basis für unendlich viele Instanzendiagramme



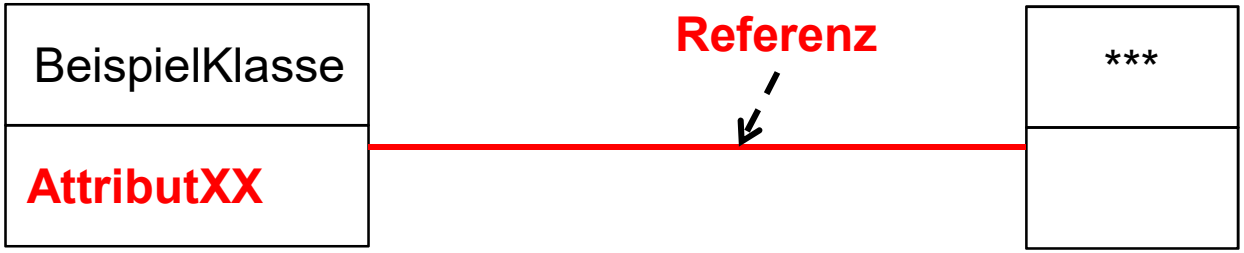
Instanzendiagramm (Objektdiagramm)

- 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.



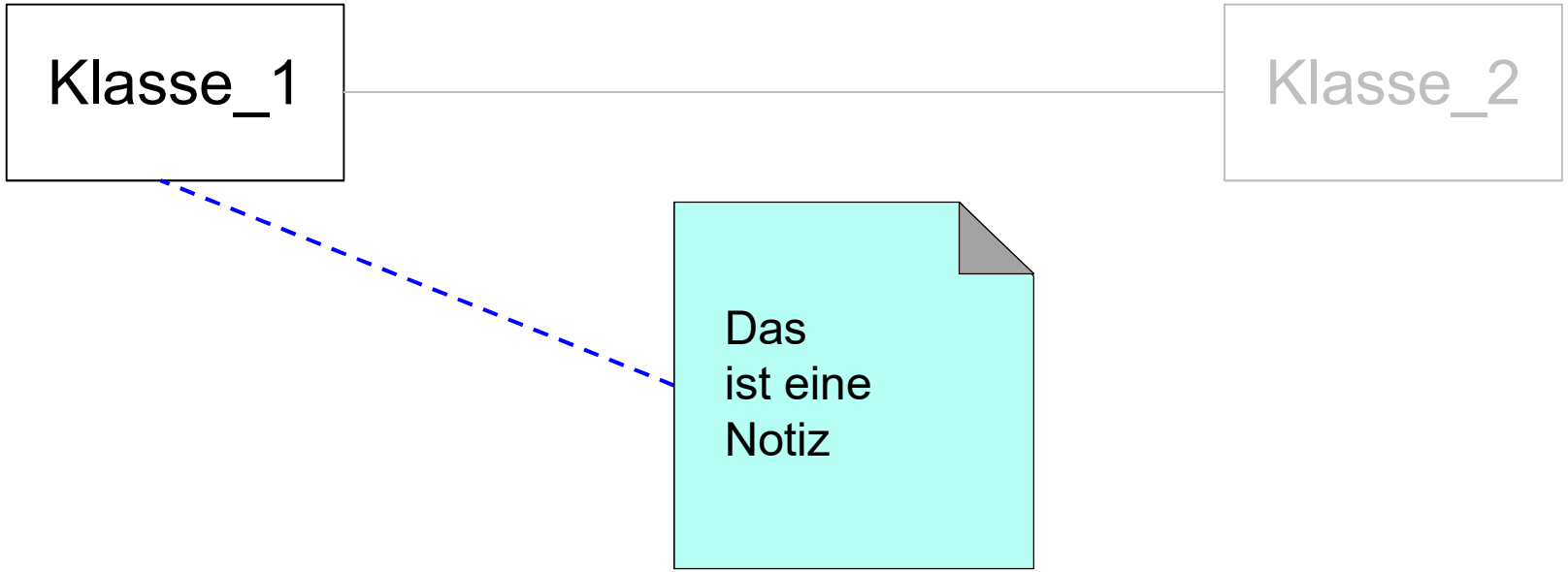
Begrifflichkeit: *Attribut* vs. *Referenz*

- **„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)



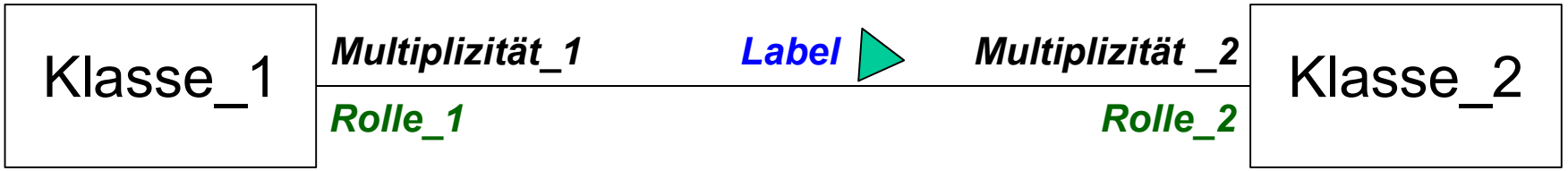
Kommentar, Notiz

- 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



Referenzen zwischen zwei Klassen bzw. Instanzen:

Assoziationen (allgemeine Darstellung)



Label: Beliebiger Bezeichner der Assoziation über/unter der Mitte der Linie. Beispiel: „*wird verwendet von*“
Label hat **keinen** Einfluss auf den späteren Quellcode!

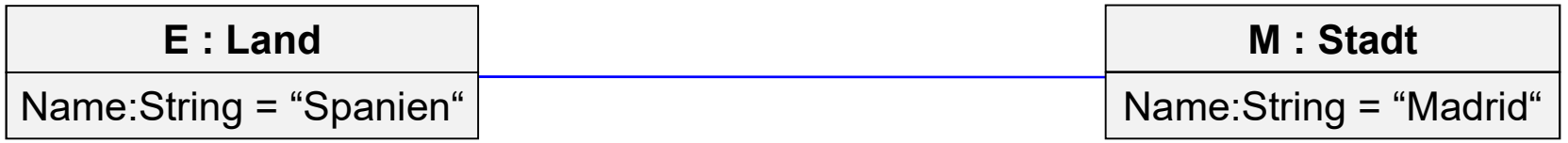
Rolle: Genaue Beschreibung der verknüpften Klasse aus der Sicht der jeweils anderen Klasse. Steht am Ende der Linie.
(„*Rolle_2* ist die Rolle, die *Klasse_2* aus Sicht von *Klasse_1 spielt*“)

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

Assoziation: Beispiel

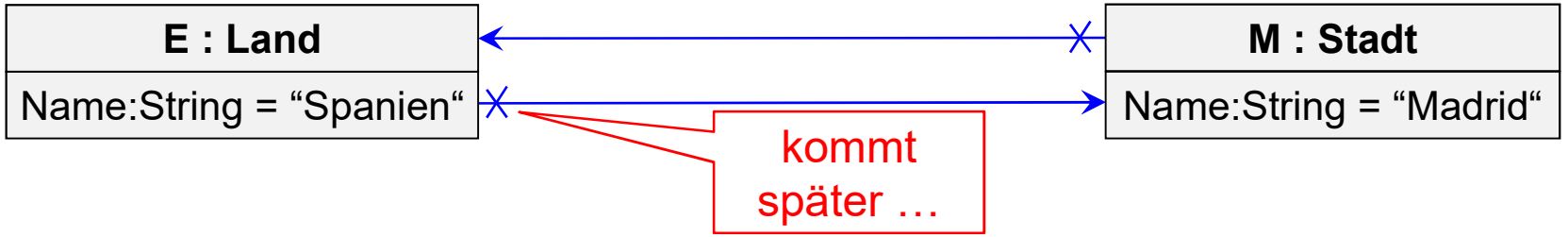
Assoziationen

- Bemerkungen zur Darstellung der Beziehung zwischen zwei **Instanzen**



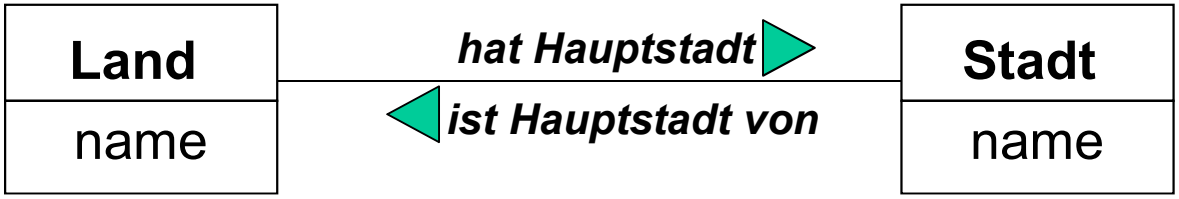
Bidirektionale Assoziationen ohne Beschriftung sind bei **Instanzendiagrammen** nicht eindeutig, meist liegt tatsächlich eine Unidirektionale Beziehung vor! (Programmiersprachen kennen nur unidirektionale Assoziationen)

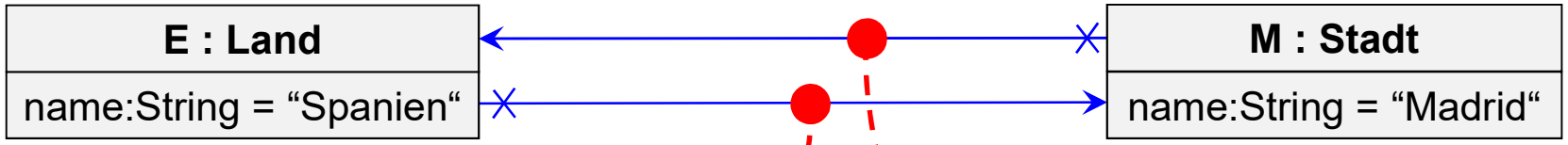
➔ Eindeutiger sind dann gerichtete Assoziationen:



➡ In SWE1: unidirektionale Assoziationen verwenden!

Assoziationen: Mapping zu Java-Code



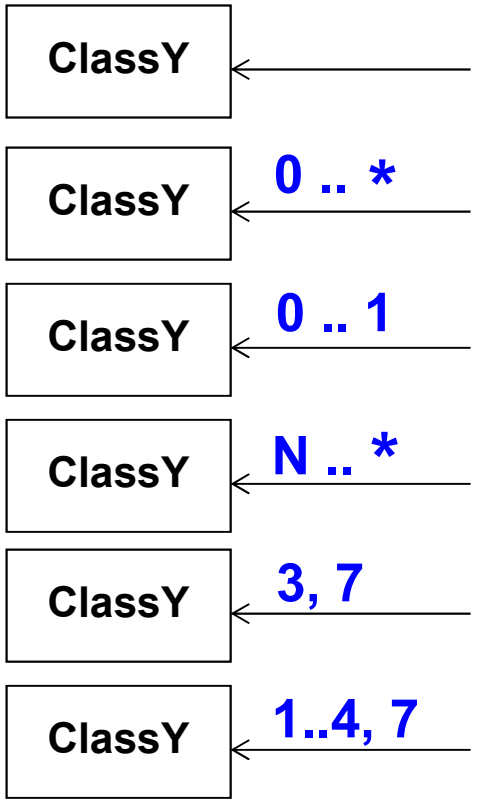



```
public class Land{

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

```
public class Stadt{

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

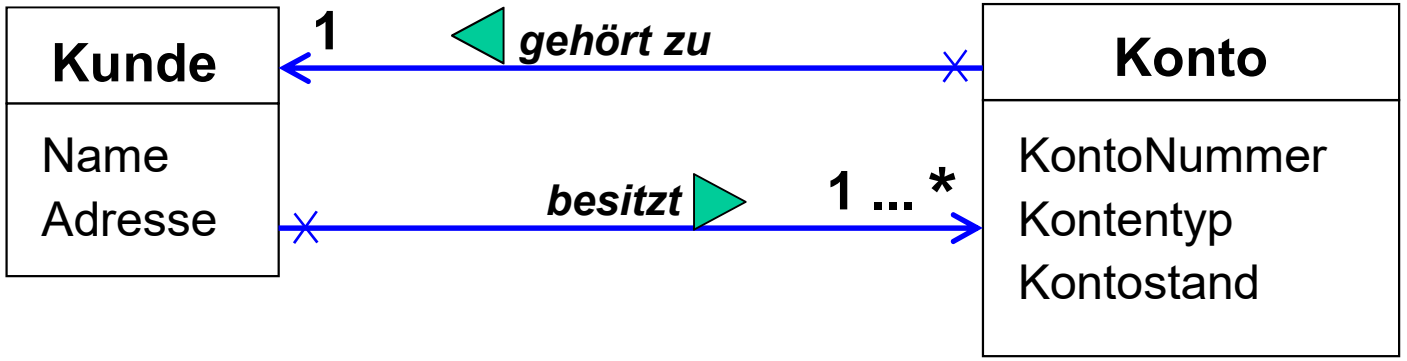
Multiplizitäten

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



Multiplizitäten: Beispiel

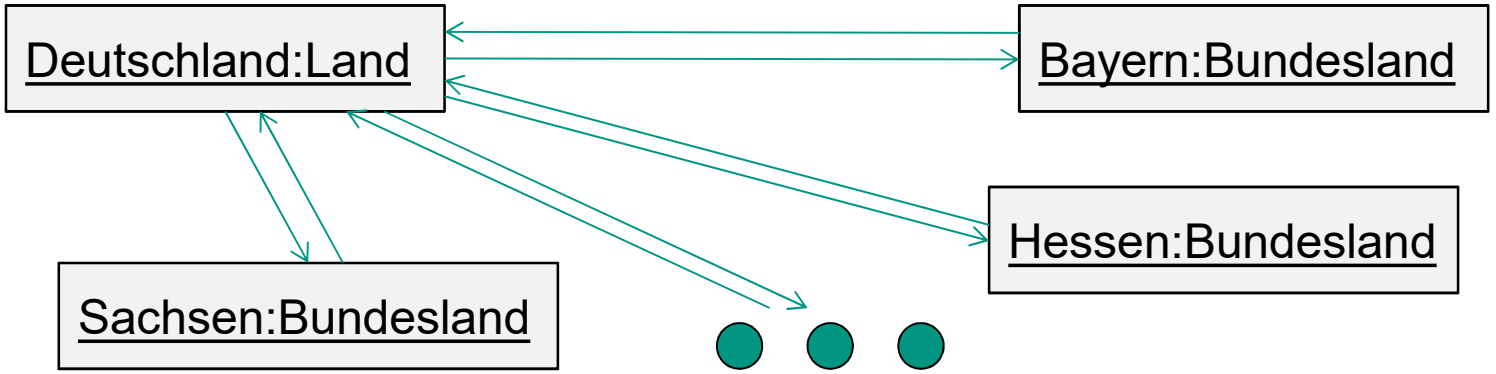
Alternativ: gerichtete Assoziationen



Multiplizitäten: Beispiel Klassen- u. Instanzendiagramm

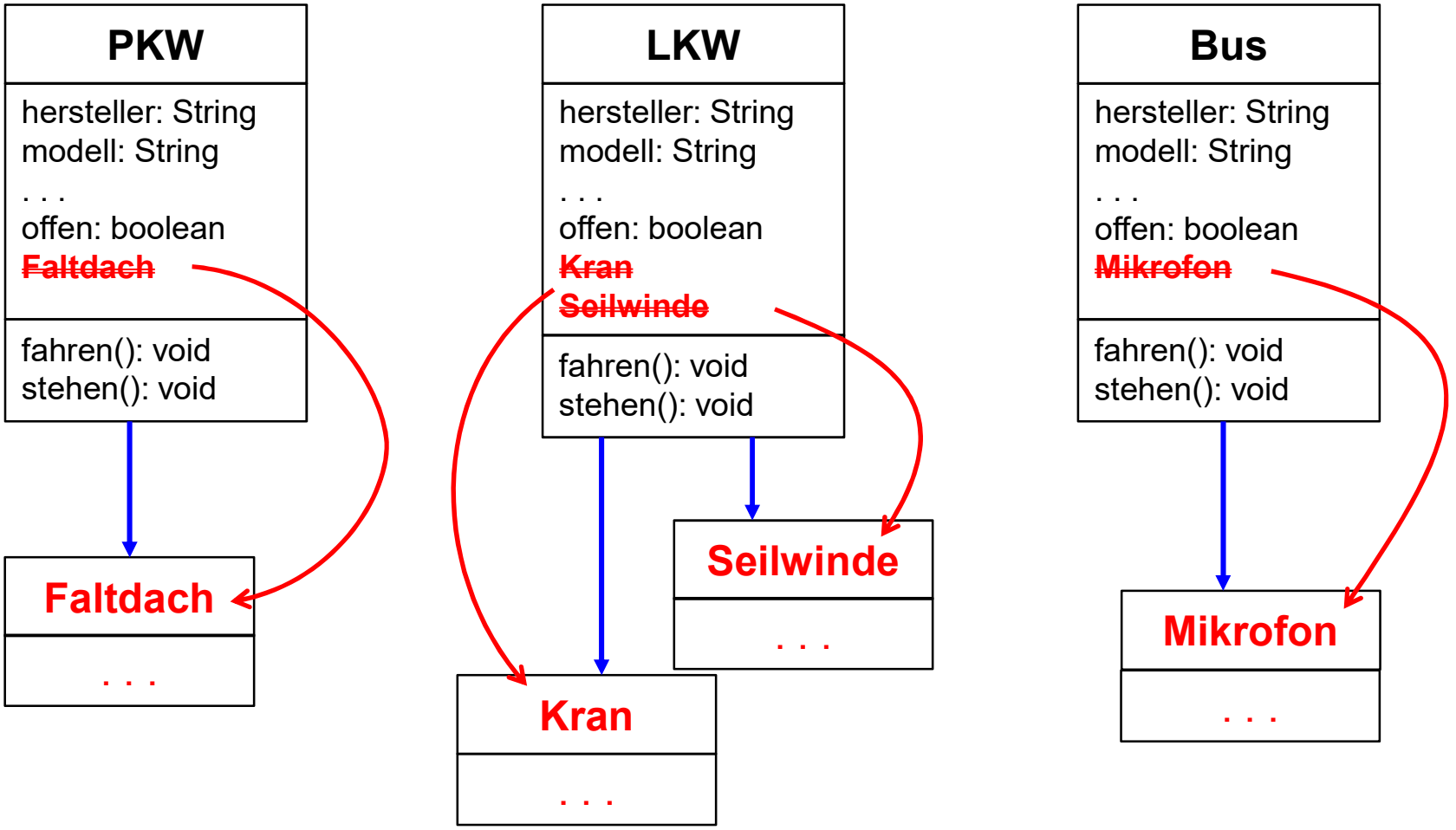
- Klassendiagramm:

- Instanzendiagramm:



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

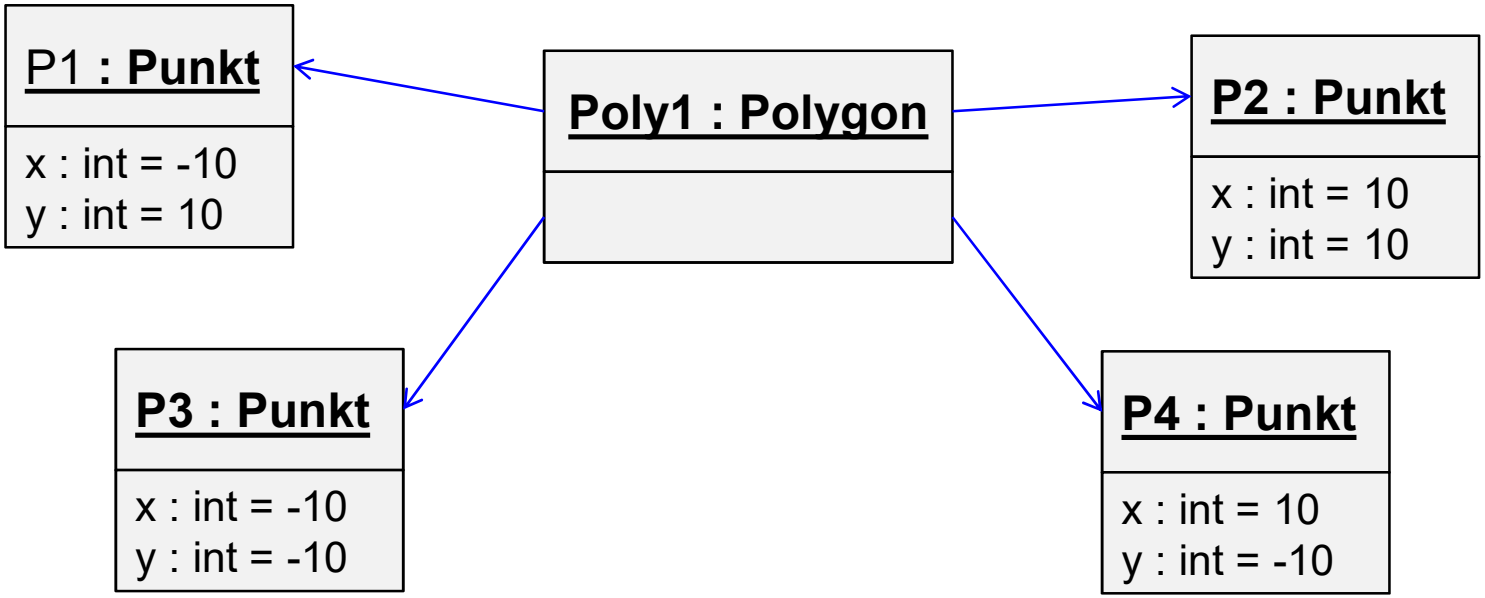
Unsere Fahrzeug-Klassen nun mit Referenzen



Übungsaufgaben

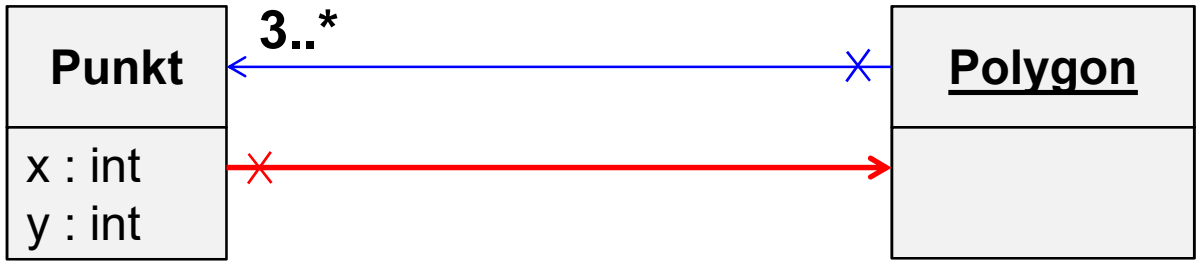
Aufgabe 1: Instanzendiagramm ➔ Klassendiagramm

- 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 1: Lösungsvorschlag




```
public class Punkt{

  private int x;
  private int y;

  . . .
}
```

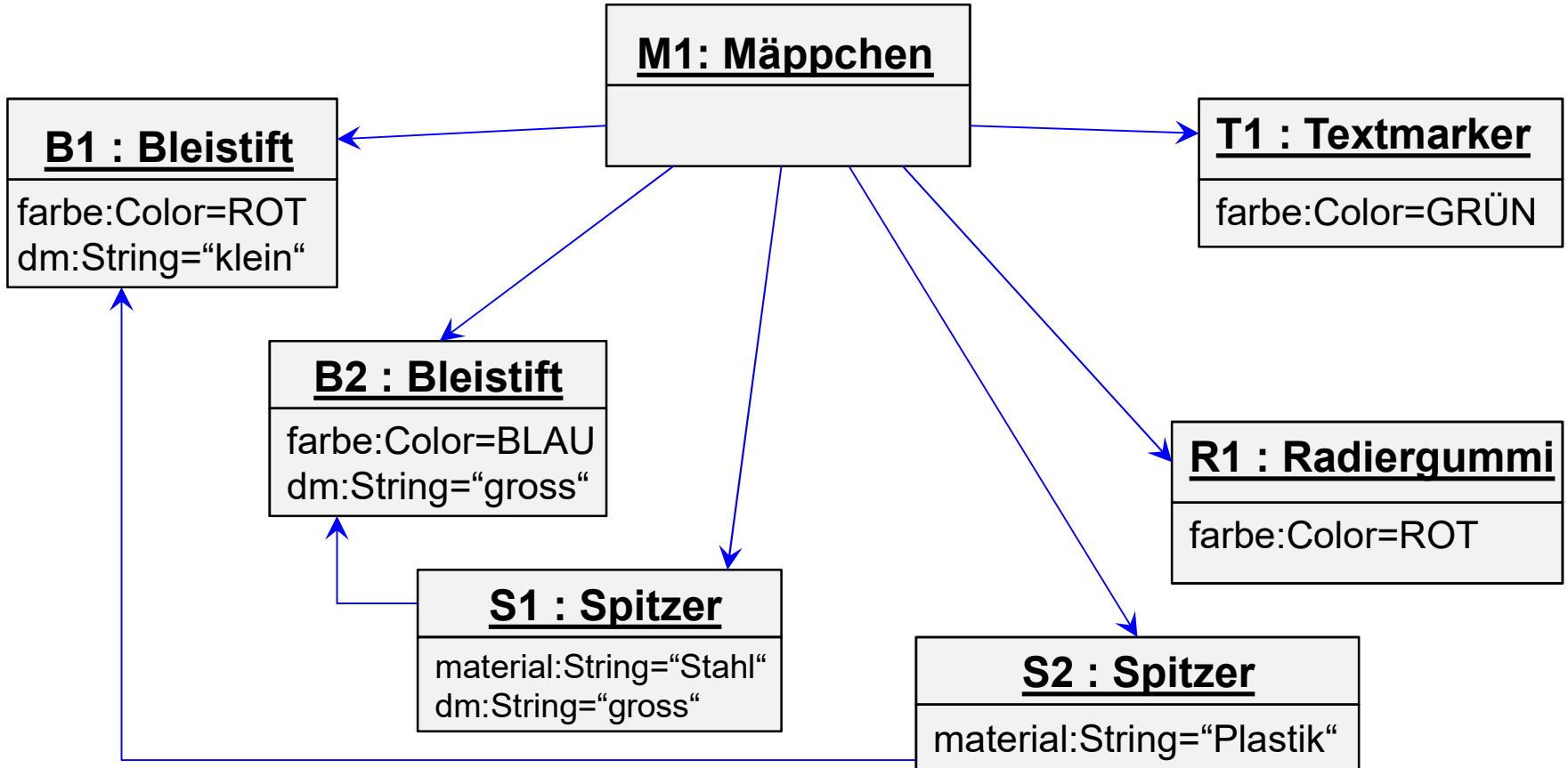
```
public class Polygon{

  private List<Punkt> Punkte;
  . . .
}
```

bidirektionale Verbindung:

- Punkt „kennt“ Polygon (nicht erforderlich ➔ Lastenheft)
- Punkt schlecht wiederverwendbar ➔ eher nicht bidirektional modellieren

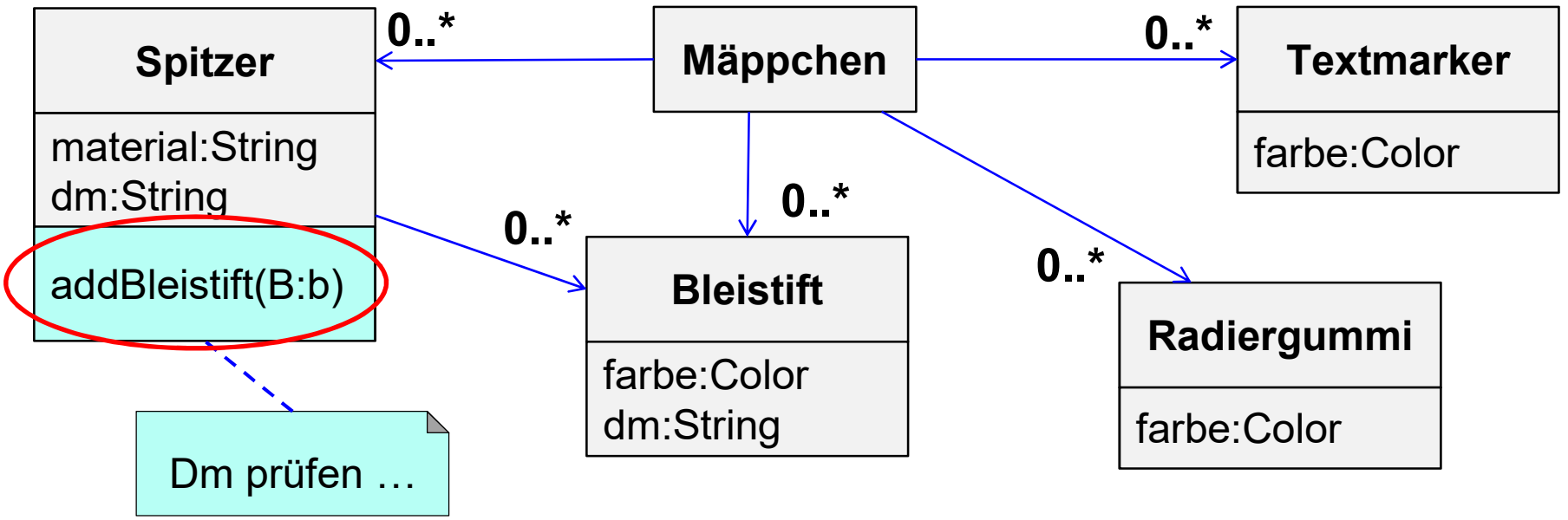
Aufgabe 2: Instanzendiagramm ➔ Klassendiagramm

- Entwickeln Sie ein Klassendiagramm aus folgendem Instanzendiagramm



dm = Durchmesser

Aufgabe 2: Lösungsvorschlag



Problematik: prüfen, ob Bleistift und Spitzer zusammenpassen

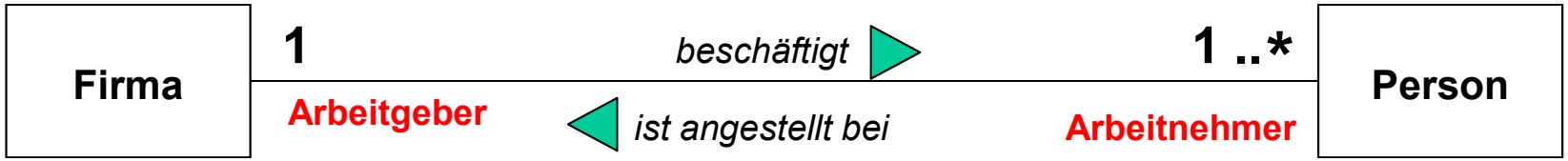
- Verwendung einer Methode, welche das Prüfen (mit) übernimmt (*addBleistift(*)*)
- (plus) Kommentar

Wichtig: Aus welcher Sicht wird modelliert?

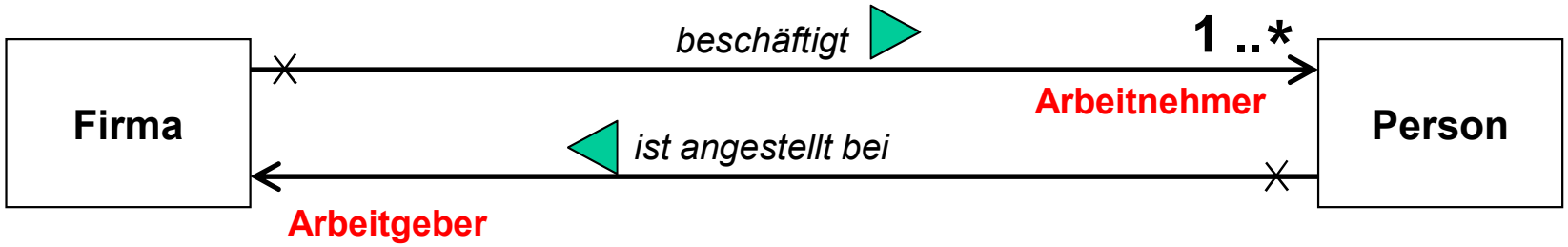
- Mäppchenhersteller oder beliebiges reales Mäppchen ➔ siehe Lastenheft!

Rollennamen

- 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

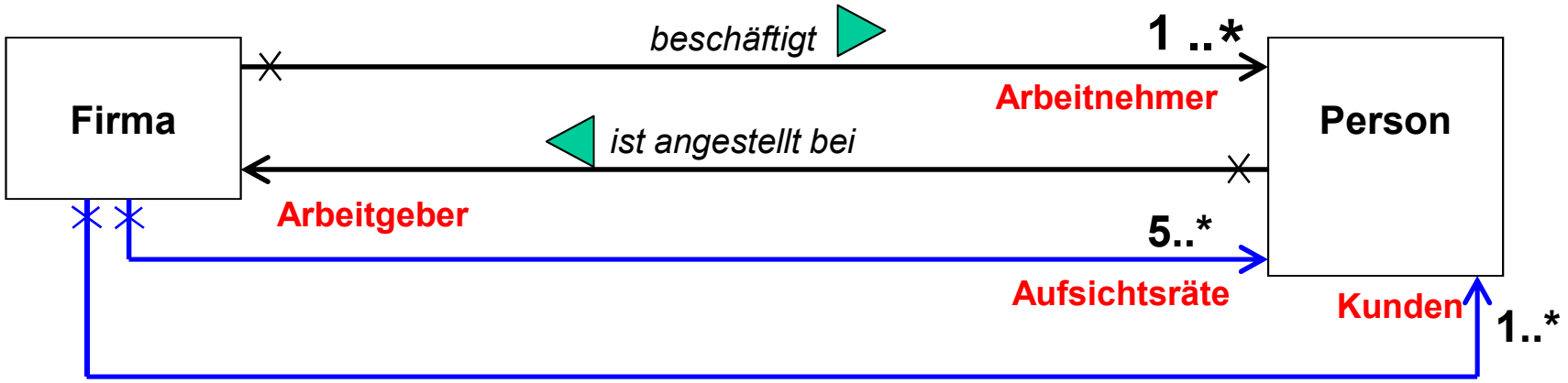


Alternativ: gerichtete Assoziationen

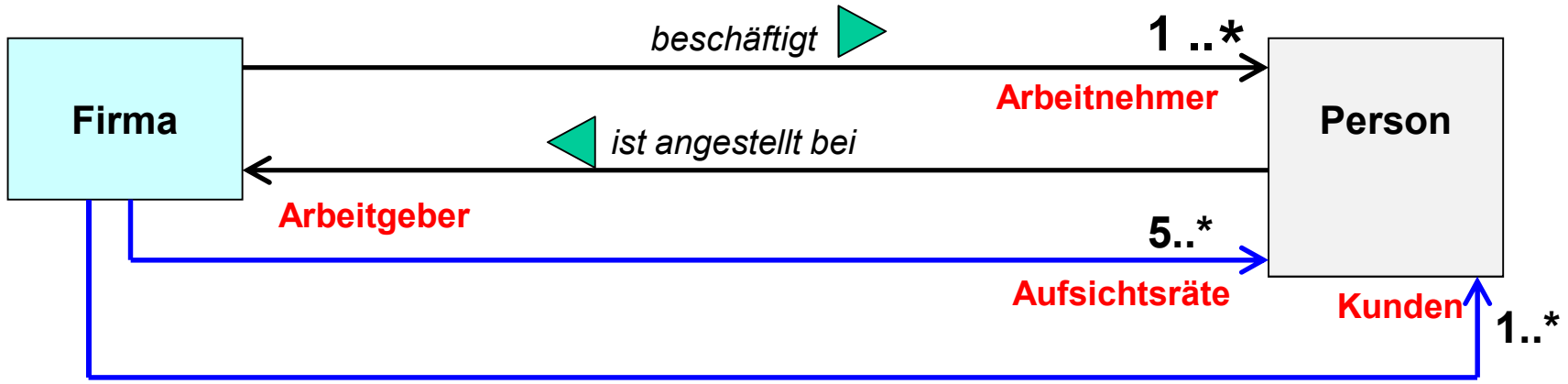


Rollennamen

- 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 meist als Referenznamen verwendet)



Multiplizitäten + Rollen: Mapping zu Java-Code

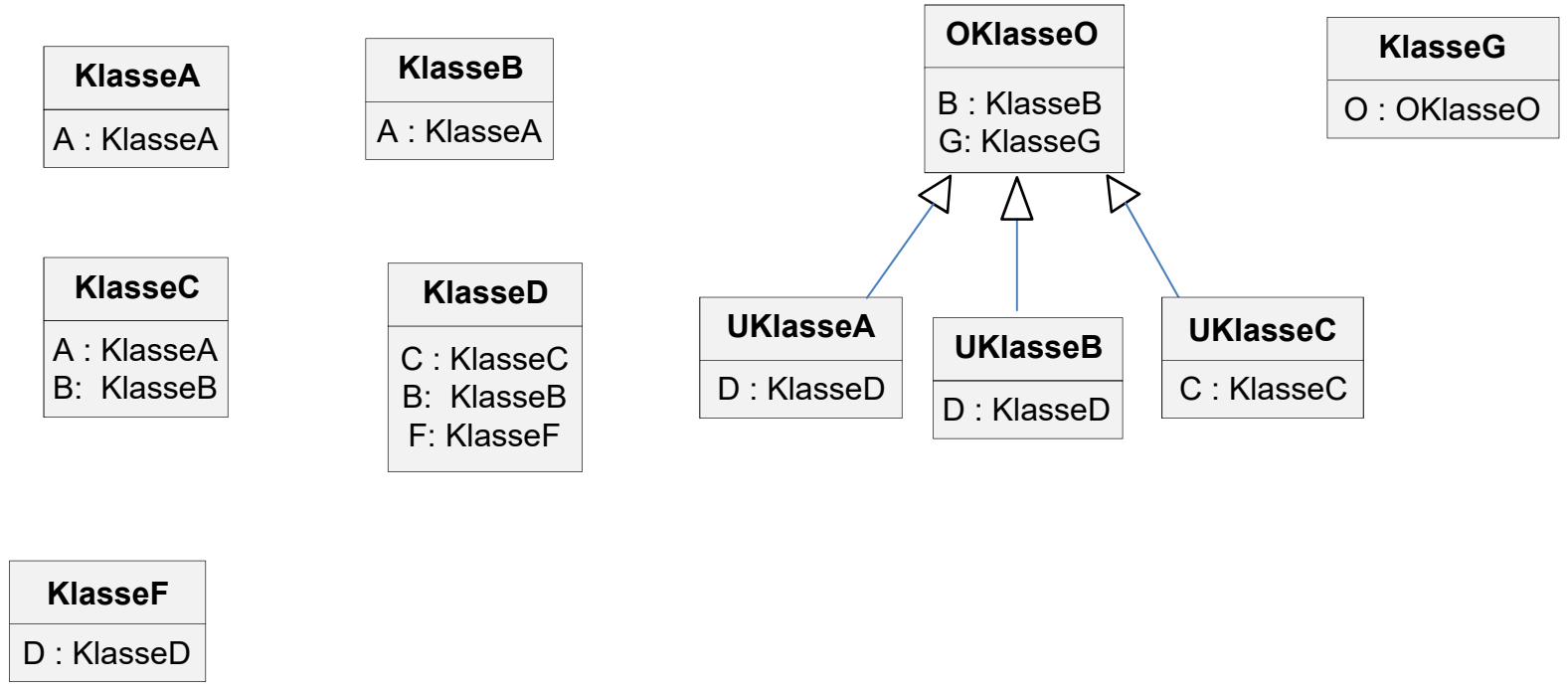



```java
public class Firma{

  // beschaeftigt
  private List<Person> Arbeitnehmer;
  private List<Person> Aufsichtsräte;
  private List<Person> Kunden;
  . . .
}
```

```java
public class Person{

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

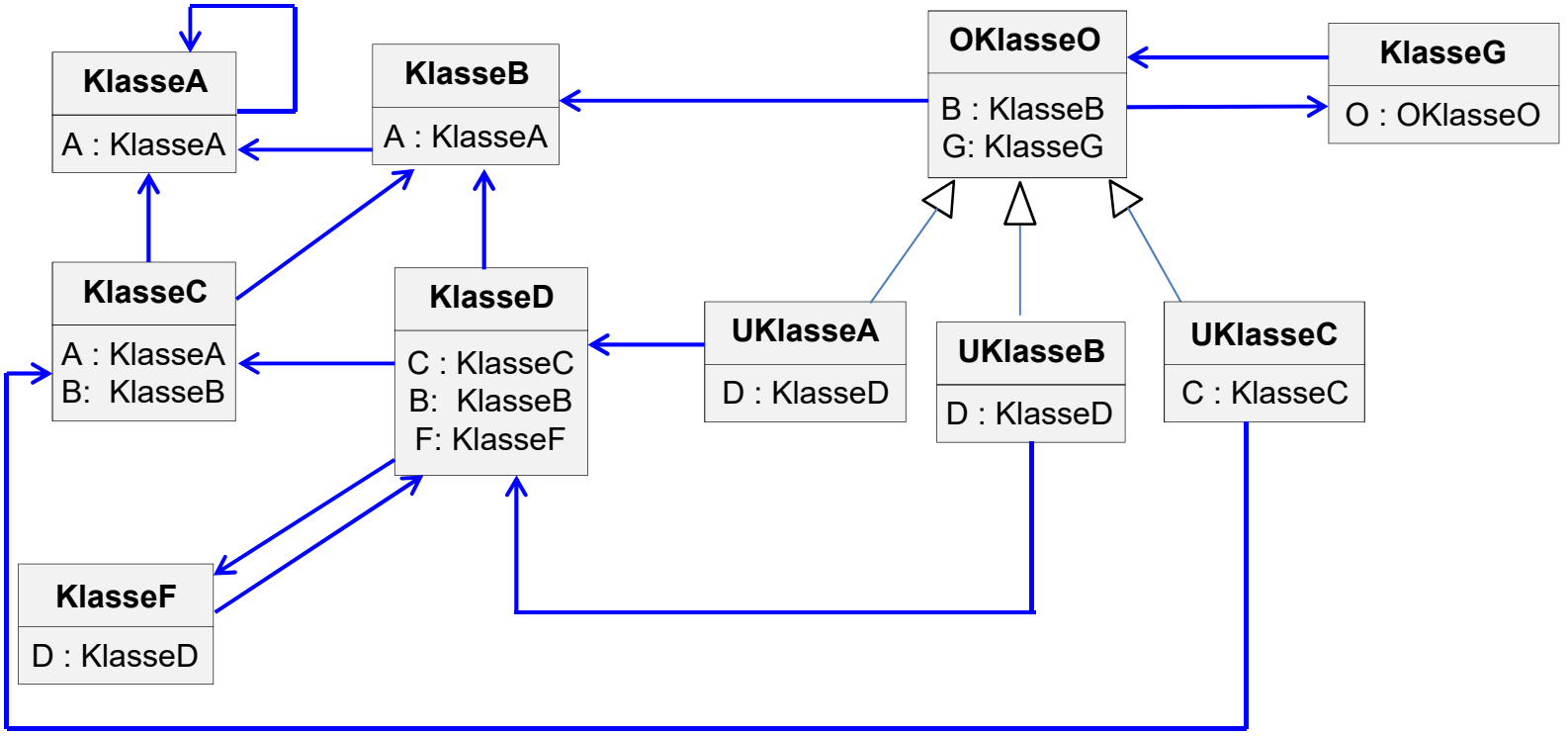
Assoziationen und Rollen? Reichen nicht Attributnamen?

Klassendiagramm ausschließlich mit Attributdarstellung



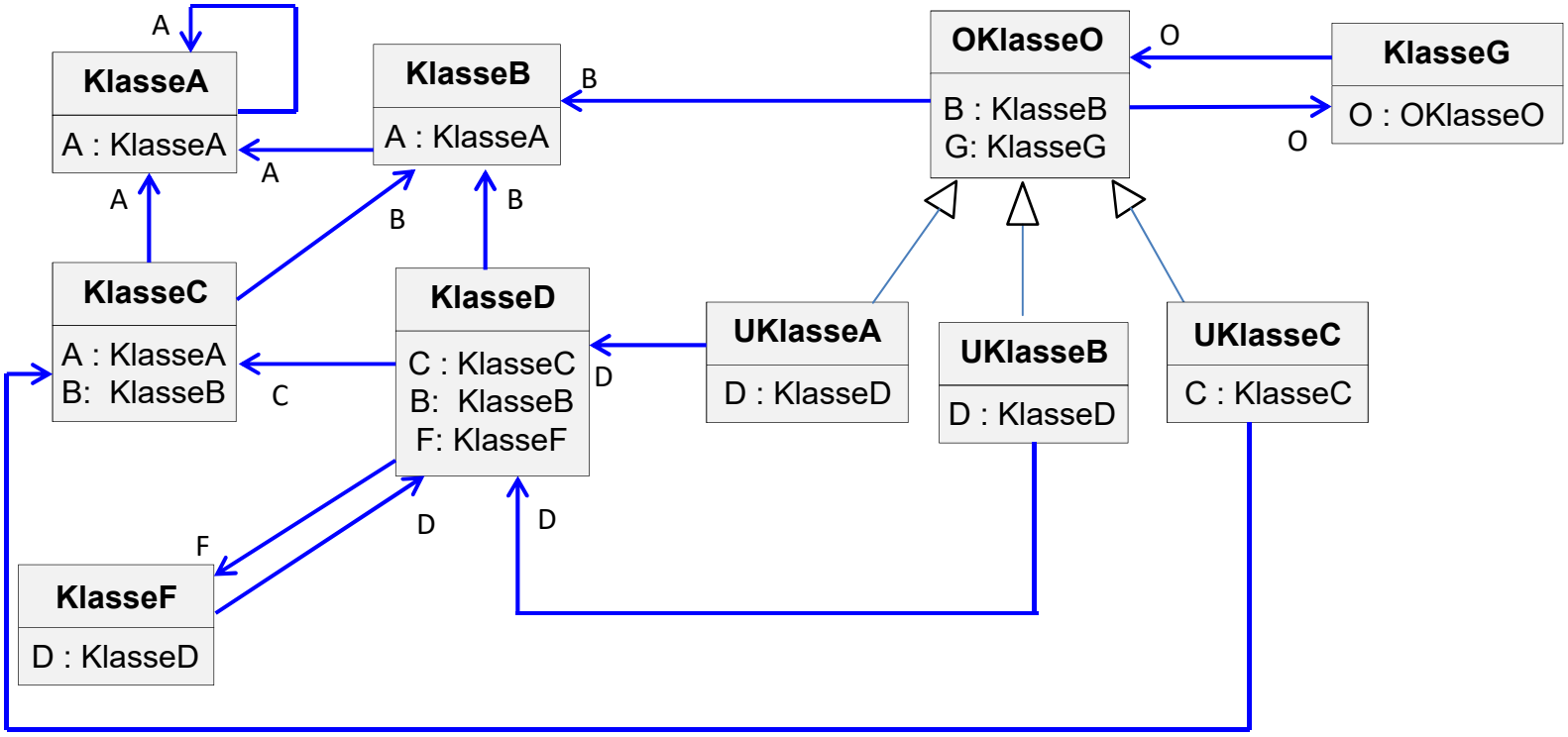
Warum Assoziationen, reichen nicht Attributnamen?

Klassendiagramm mit „zusätzlicher“ Referenzdarstellung



Warum Assoziationen, reichen nicht Attributnamen?

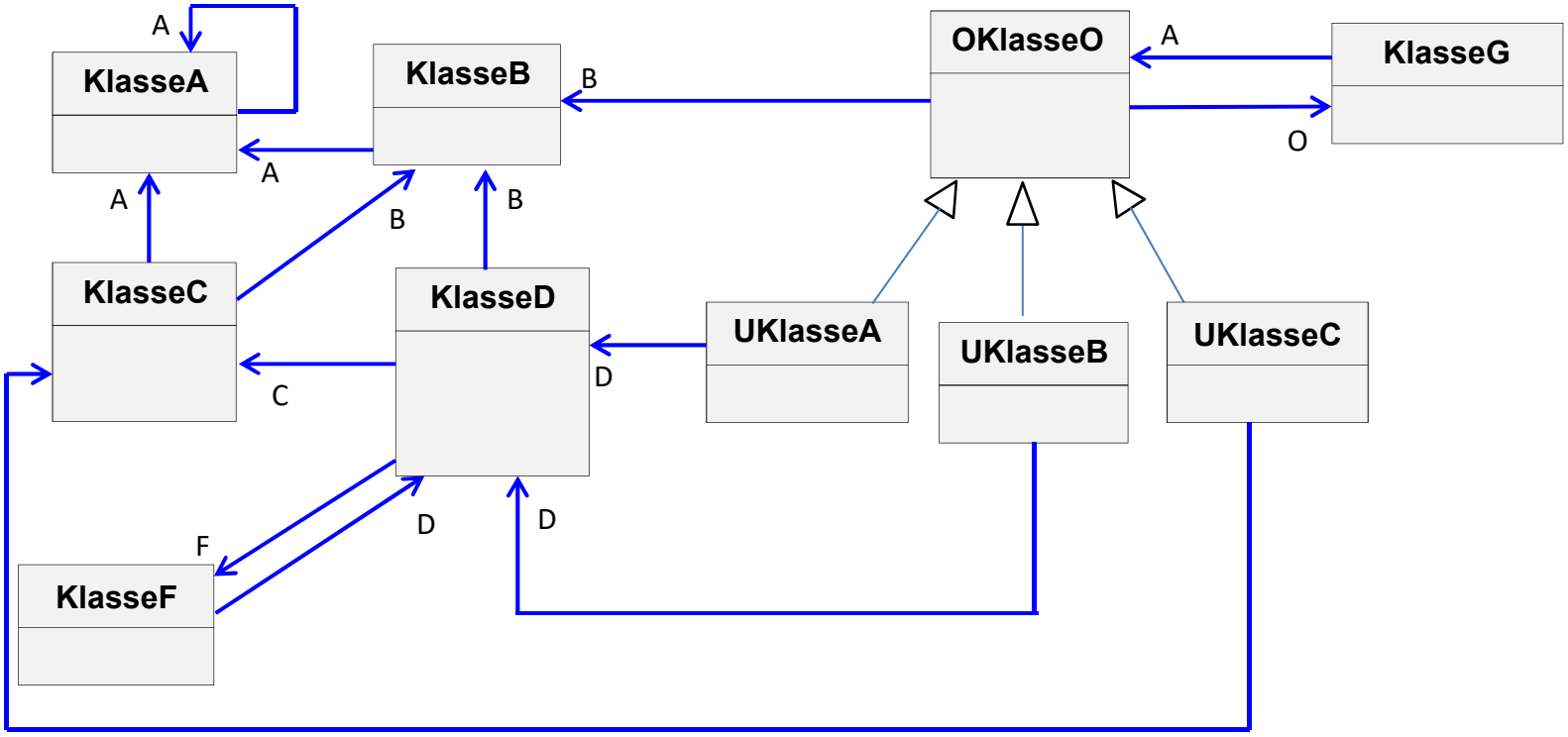
Klassendiagramm mit Referenzdarstellung plus Rollennamen



Warum Assoziationen, reichen nicht Attributnamen?

Klassendiagramm mit Referenzdarstellung plus Rollennamen

➔ ersetzt Attributdarstellung

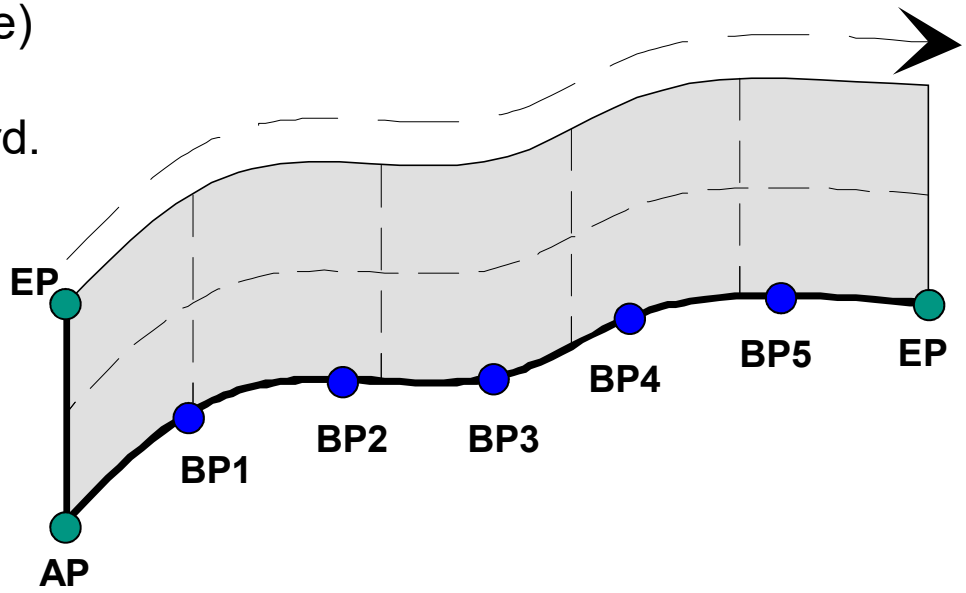


Übungsaufgaben

Aufgabe 3: Klassendiagramm für Translationsfläche

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

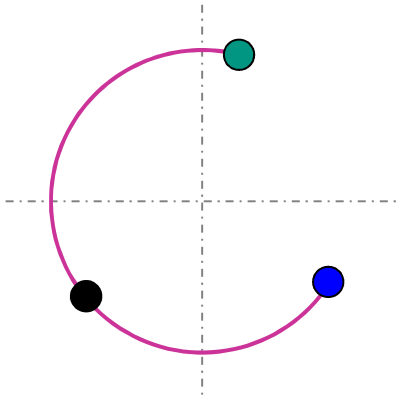
- Eine erste Fläche (Translationsfläche) ist definiert durch eine Linie, die entlang einer Raumkurve "gezogen" wird.
- Die Linie selbst verweist auf einen Anfangs- und einen Endpunkt.
- Die Raumkurve verweist ebenfalls auf einen Anfangs- und einen Endpunkt sowie auf eine Liste von (z.B. 5) Basispunkten.



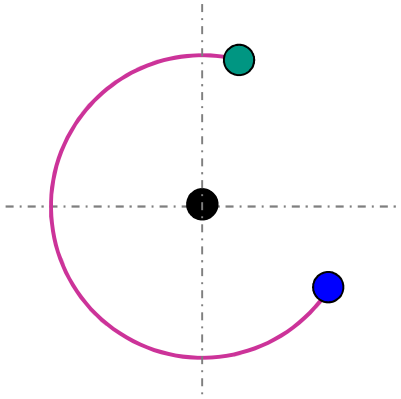
Aufgabe 4: Klassendiagramme für Kreisbögen

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

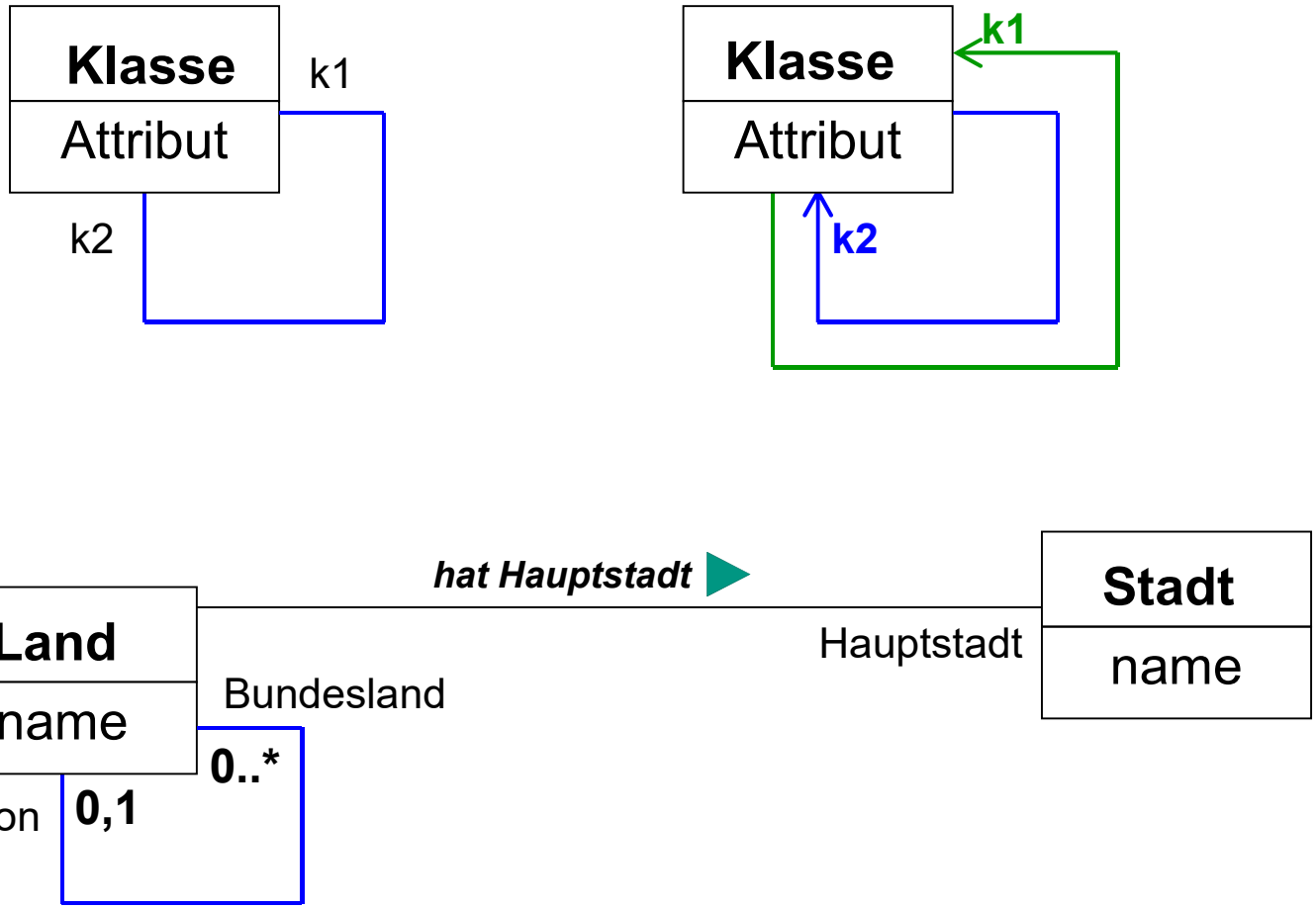
- Ein Kreisbogen ist definiert durch drei Punkte (Anfangs- und Endpunkt sowie ein dazwischen liegender Punkt auf dem Kreisumfang).



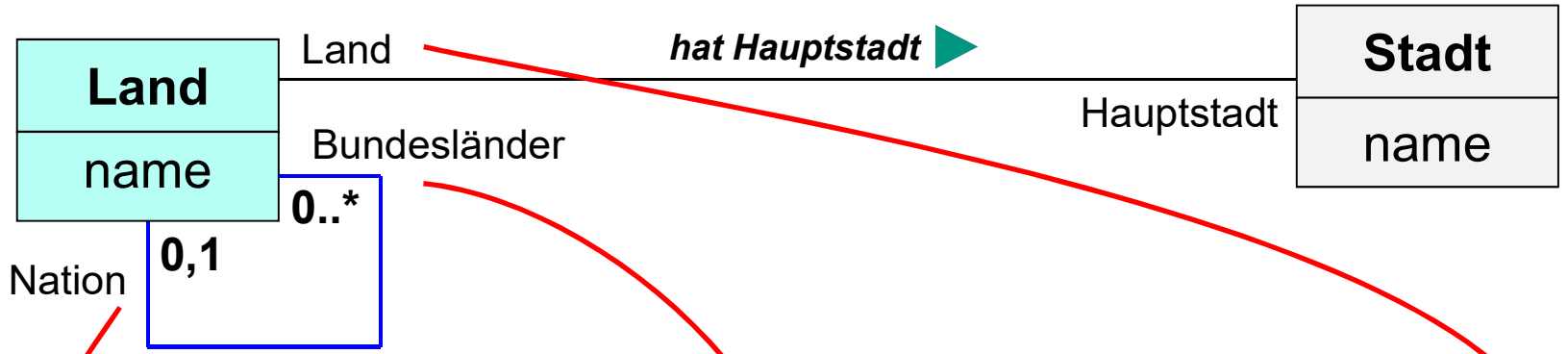
- Ein weiterer Kreisbogen ist ebenfalls definiert durch drei Punkte (Anfangs- End- und Mittelpunkt).



Reflexive Assoziationen



Reflexive Assoziationen: Mapping zu Java-Code

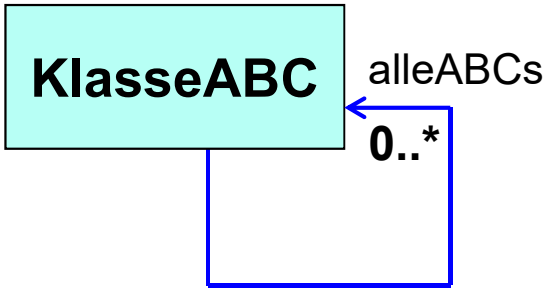



```
public class Land{

  private Stadt Hauptstadt;

  private List<Land> Bundesländer;
  private Land Nation;
  . . .
}
```

```
public class Stadt{

  private Land Land;
  . . .
}
```

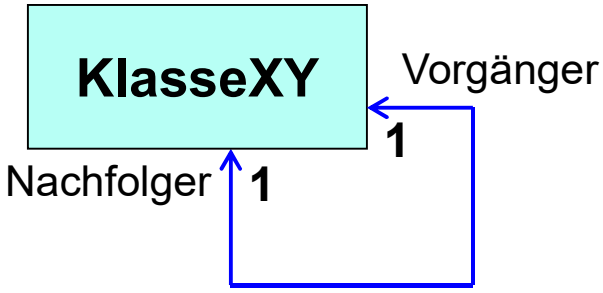
Reflexive Assoziationen: Anwendungsbeispiele

Verwaltung aller Instanzen einer Klasse (siehe *Multiton-Pattern* und *Listenklassen*)




```
public class KlasseABC{

  private static List<KlasseABC> alleABCs;
  . . .
}
```

Verkettete Listen



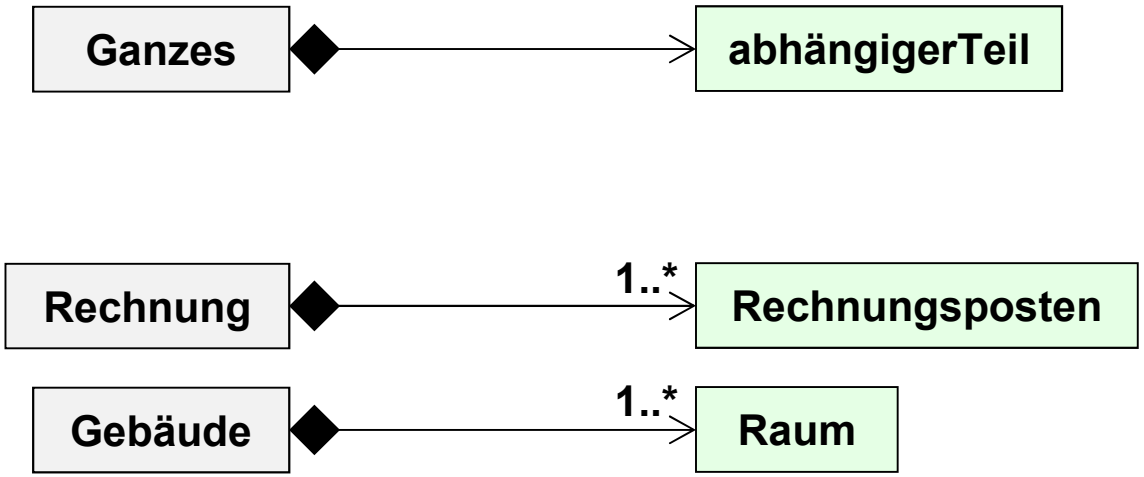

```
public class KlasseXY{

  private KlasseXY nachfolger;
  private KlasseXY vorgänger;
  . . .
}
```

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 sind durch eine gefüllte Raute gekennzeichnet:
- Sie liegt vor, wenn die Einzelteile vom Aggregat (vom Ganzen) zusätzlich existenz**ab**hängig sind.



Aggregationen und Kompositionen im Java-Code

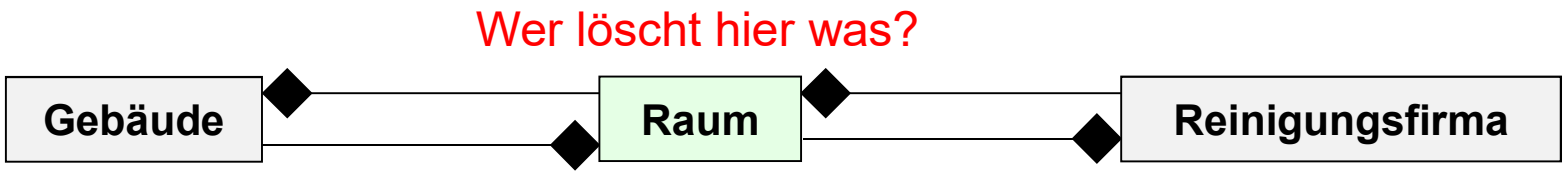
- Aggregation und Komposition werden im Code **identisch** definiert
  (bestenfalls wird vom Modellierungstool ein Kommentar erzeugt)

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

- Existenzabhängigkeiten bei Kompositionen:
  - kaskadierendes Löschen (explizit oder mit Annotationen (z.B. JPA))

Kompositionen: Problematik der Existenzabhängigkeit

- kaskadierendes Löschen:



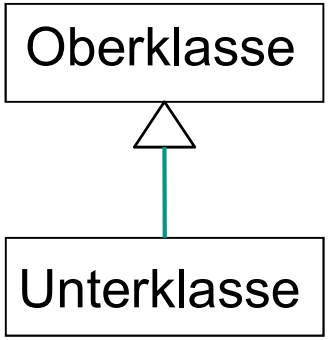
Beim Löschen der Reinigungsfirma wird auch das Gebäude gelöscht!
(und umgekehrt)

- „Sparsam“ verwenden und Konsistenz beachten!

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



Aufgabe 5: Grafikeditor (mit Vererbung)

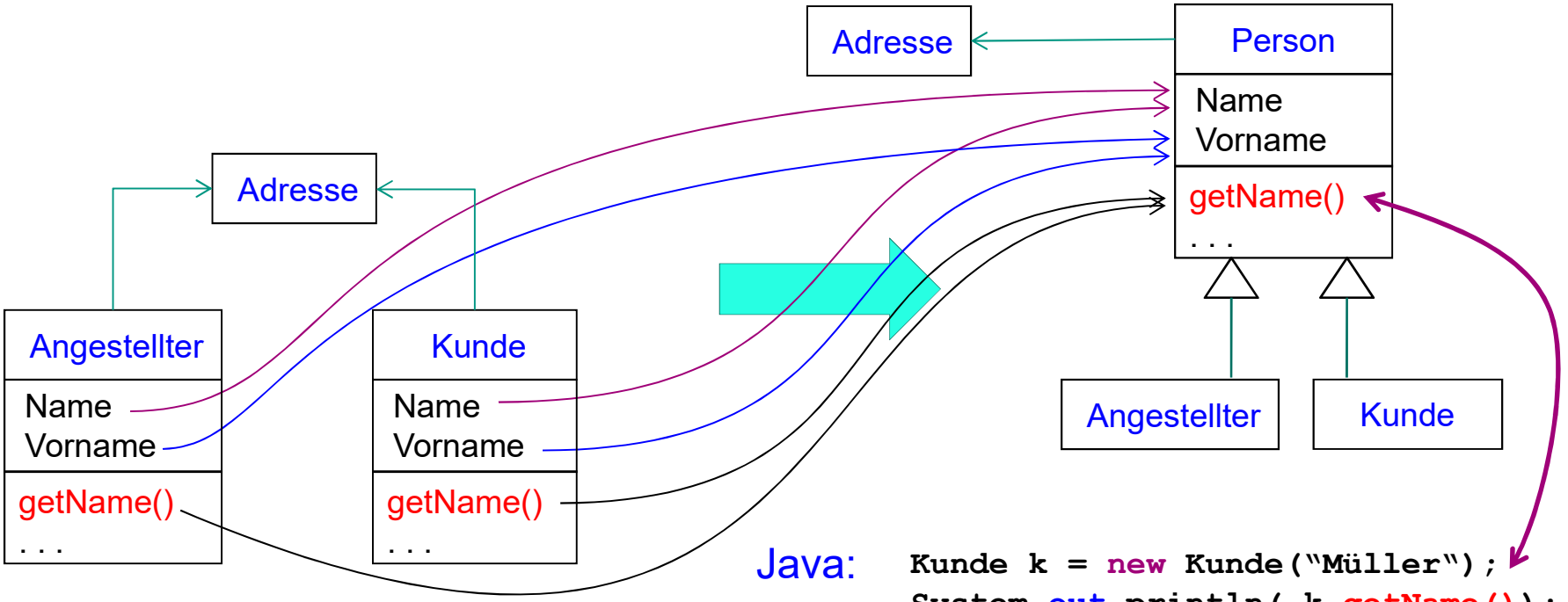
Karlsruher Institut für Technologie

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.

Vererbung (2)

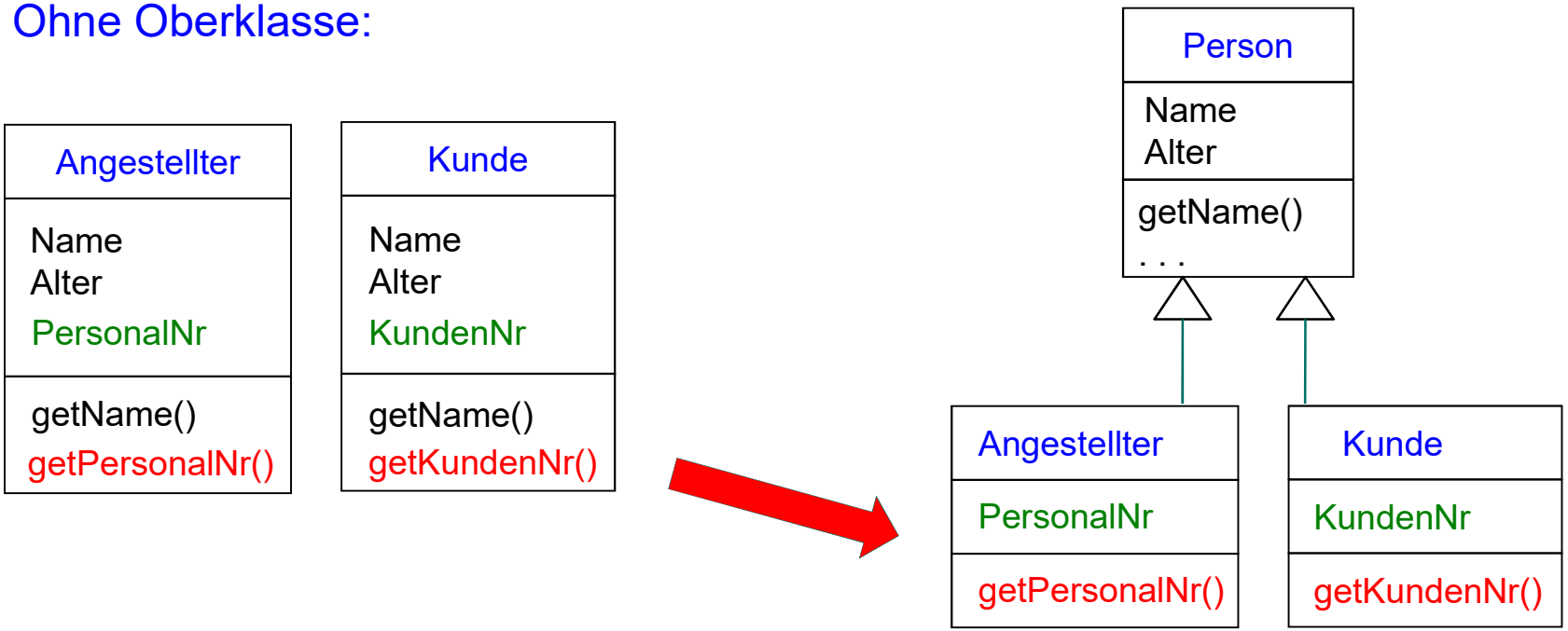
- Operationen, Attribute und Referenzen, die für eine Gruppe von Unterklassen gelten, werden
  - der Oberklasse zugewiesen
  - von den einzelnen Unterklassen gemeinsam genutzt



Vererbung (3)

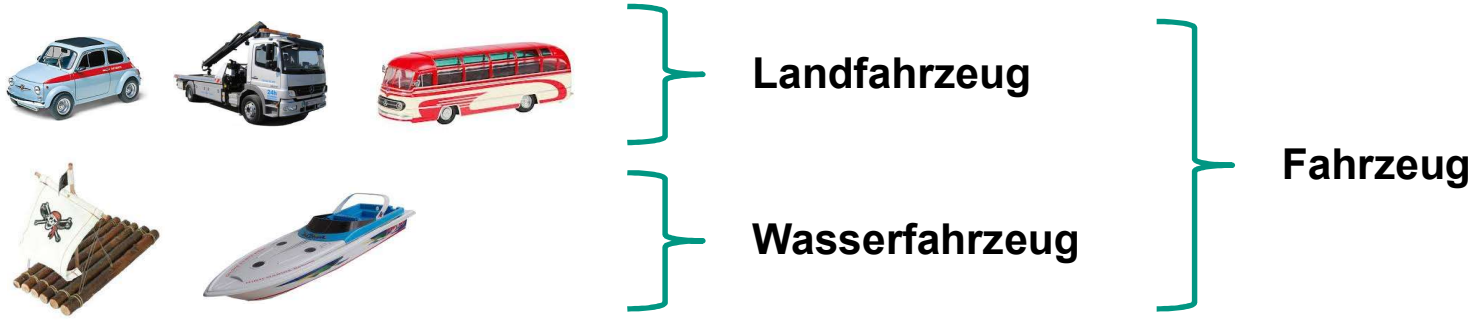
- Jede Unterklasse
  - erbt alle Attribute und Operationen ihrer Oberklasse(n)
  - und fügt ihre eigenen Attribute und Operationen hinzu

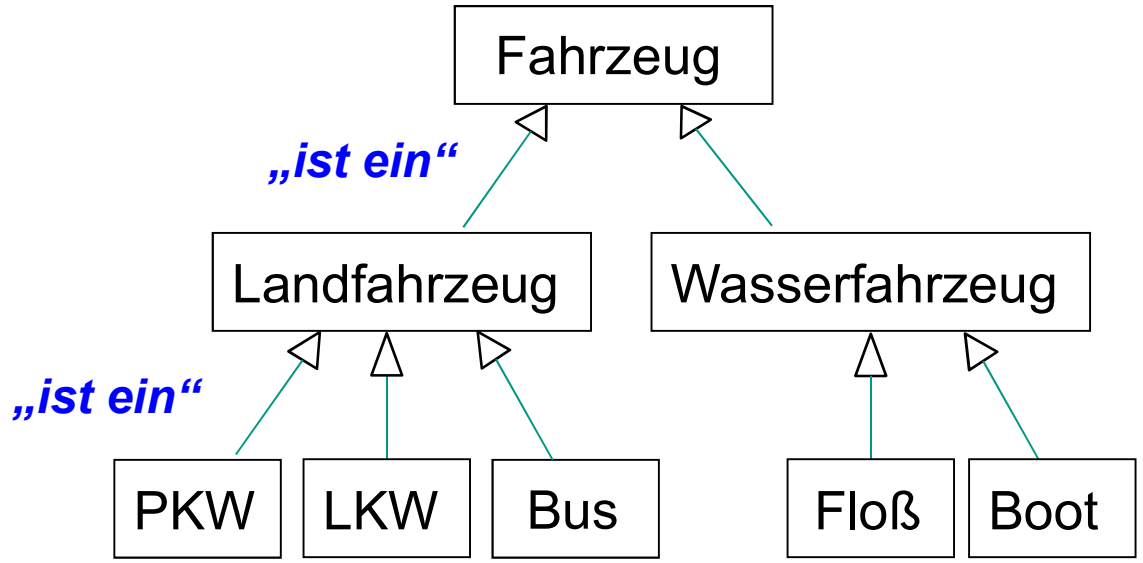
Ohne Oberklasse:



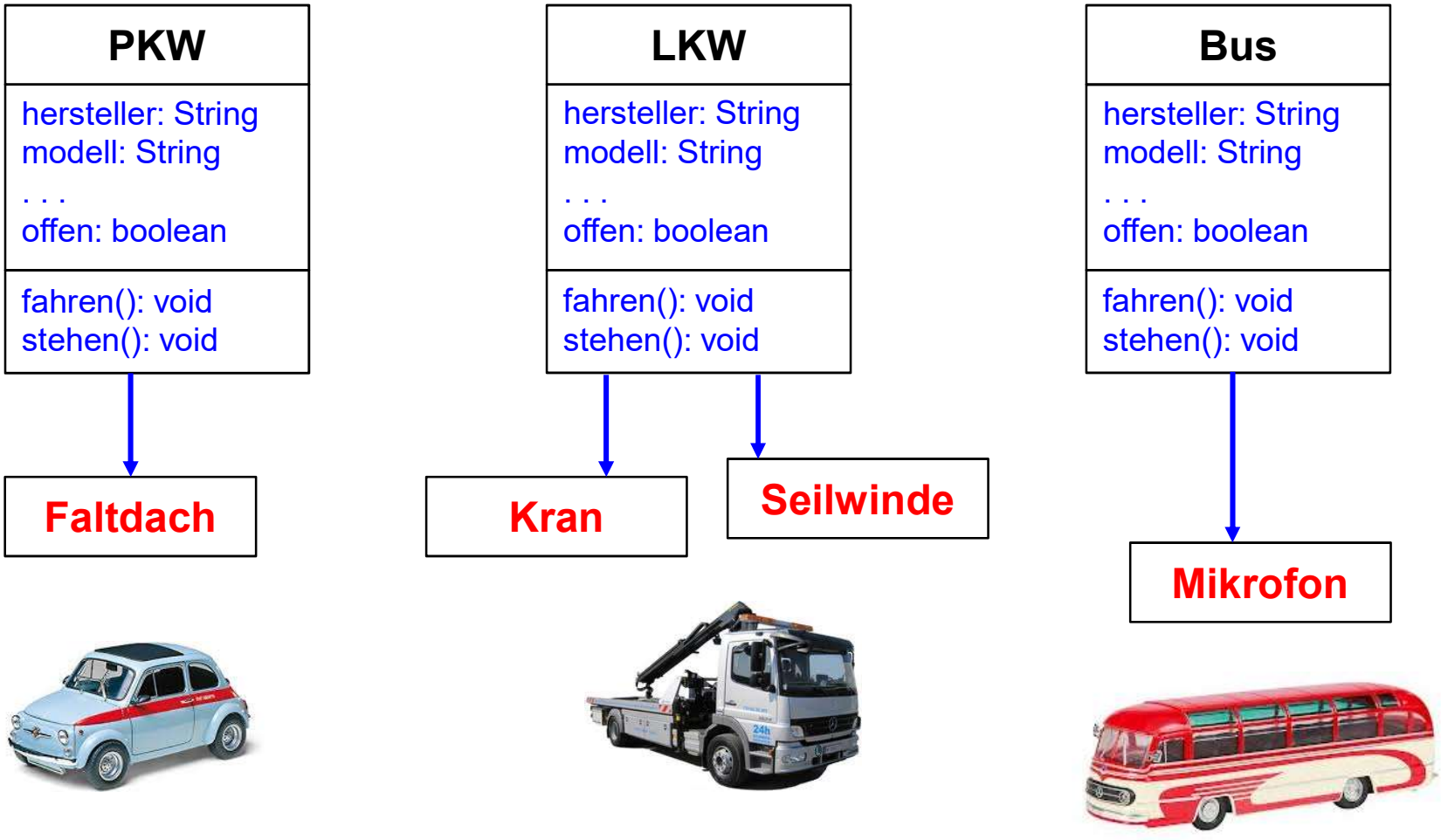
Beispiel für Vererbung:

- unsere bereits bekannten Fahrzeuge



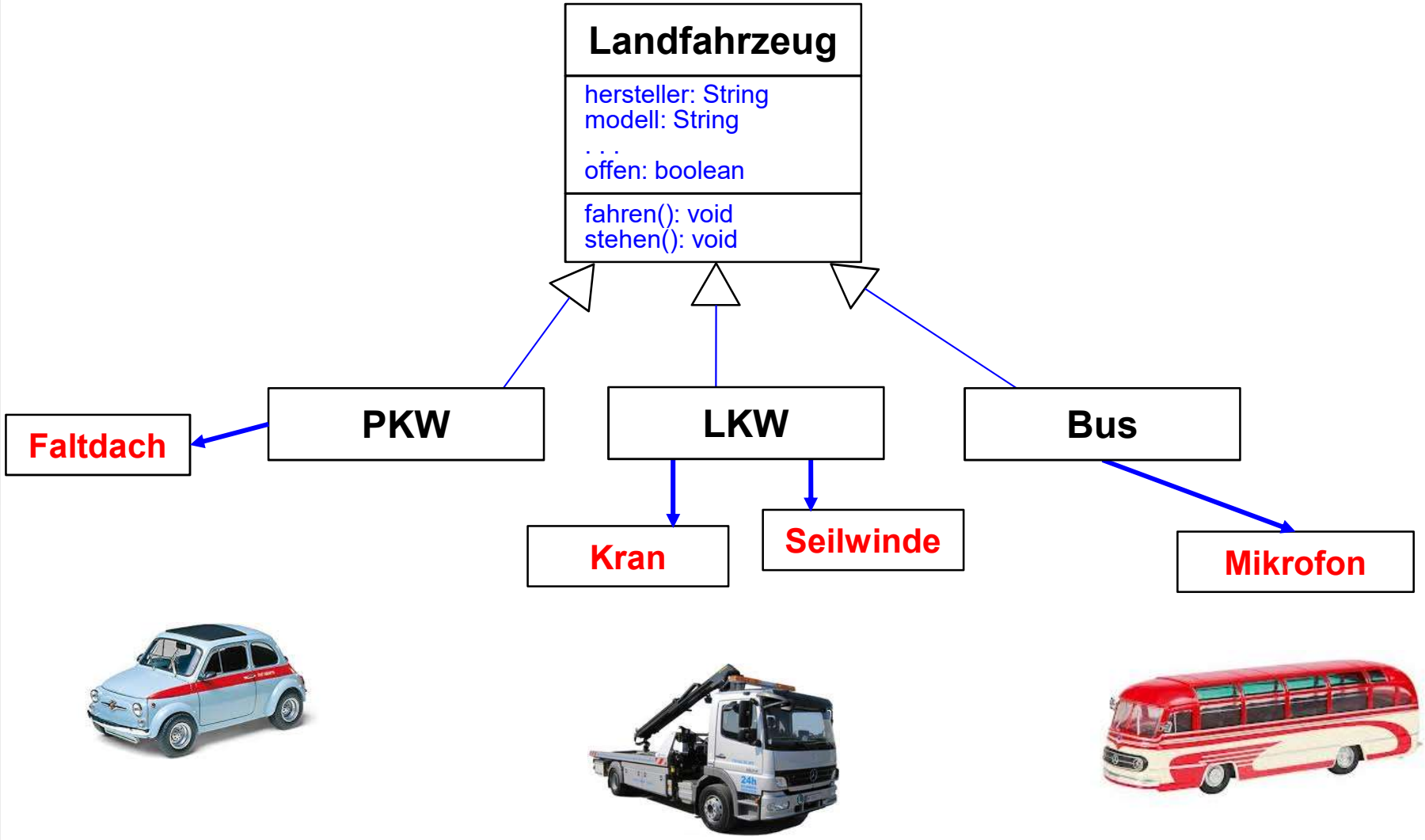


Beispiel für Vererbung (Klassendiagramm zur Erinnerung)



➔ **lauter identische Attribute und Operationen!**

Beispiel für Vererbung



Beispiel für Vererbung (Forts. für Landfahrzeuge)

Referenzen für PKW, LKW und Bus:
➔ Motor, Räder, Sitze, Besitzer, …

nur PKW: ➔ **Faltdach**
nur LKW: ➔ **Kran, Seilwinde**
nur Bus: ➔ **Mikrofon**

gemeinsame Attribute

gemeinsame Operationen

Fahrzeug
hoehe: int
breite: int
●●●
fahren(): void
stehen(): void

gemeinsame Referenzen

Motor 1
Rad 5
Sitz 2..*

Wasserfahrzeug

gemeinsame Attribute

Landfahrzeug
offen: boolean

individuelle Referenzen

Faltdach 0,1 PKW
LKW
Bus 0,1 Mikro
0,1 Kran
0,1 Seilwinde

Beispiel für Vererbung (Forts. für Landfahrzeuge)

Referenzen für PKW, LKW und Bus:
➔ Motor, Räder, Sitze, Besitzer, …

nur PKW: **Faltdach**
nur LKW: **Kran, Seilwinde**
nur Bus: **Mikrofon**

gemeinsame Attribute

gemeinsame Operationen

Fahrzeug
hersteller: String
modell: String
●●●
fahren(): void
stehen(): void

1 — Besitzer
1..* — Sitz

gemeinsame Referenzen

2..* — Rad
1..* — Motor

Wasserfahrzeug
●●●

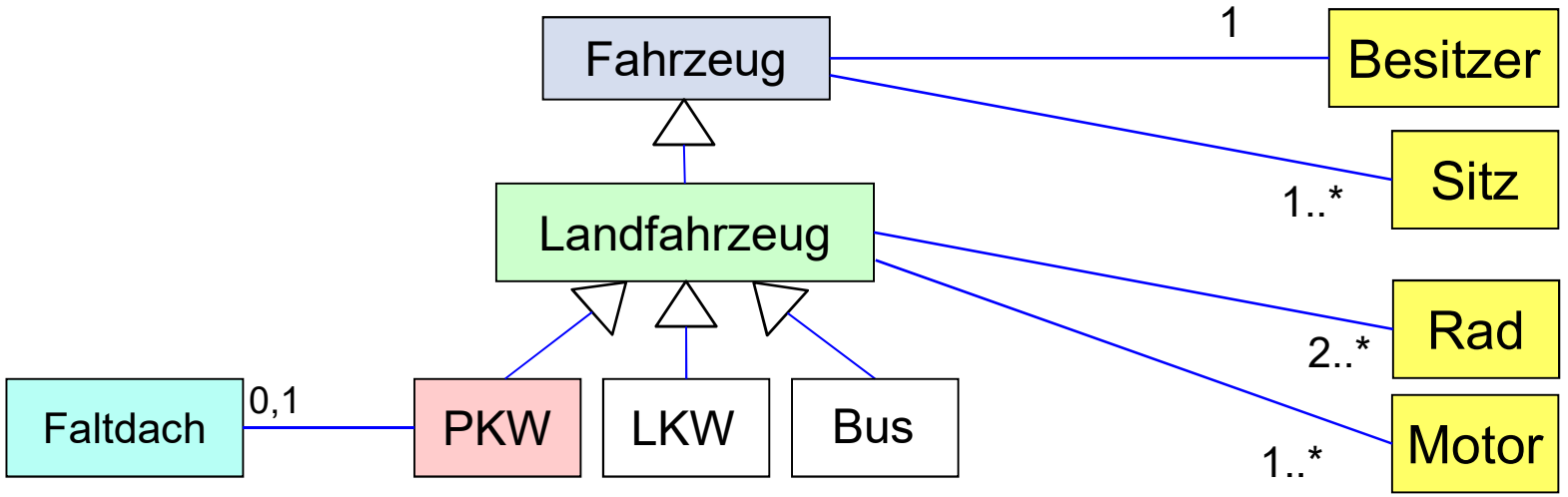
gemeinsame Attribute

Landfahrzeug
offen: boolean

individuelle Referenzen

Faltdach — 0,1 — PKW
LKW
Bus — 0,1 — Mikro
0,1 — Kran
0,1 — Seilwinde

Vererbung: Mapping zu Java-Code



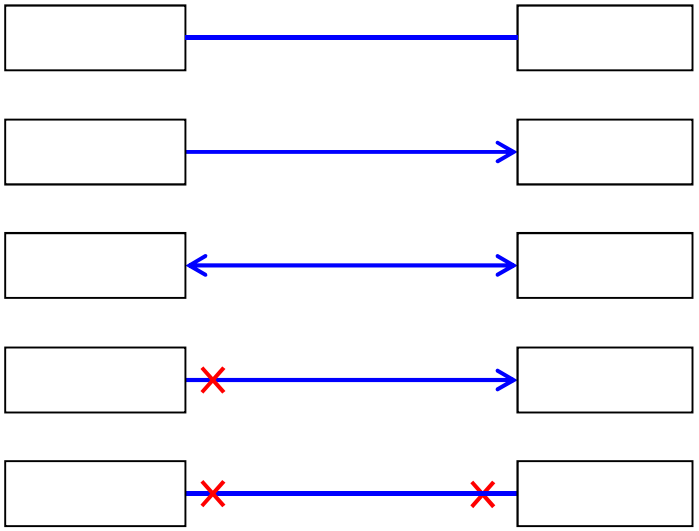

```
public class Fahrzeug{
  private Besitzer besitzer;
  private List<Sitz> sitze;
   . . .
}
```

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

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

Navigierbarkeit von Assoziationen

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



Beide Richtungen (Standardfall)

Eine (bevorzugte) Richtung

Beide Richtungen

Ausschließlich eine Richtung

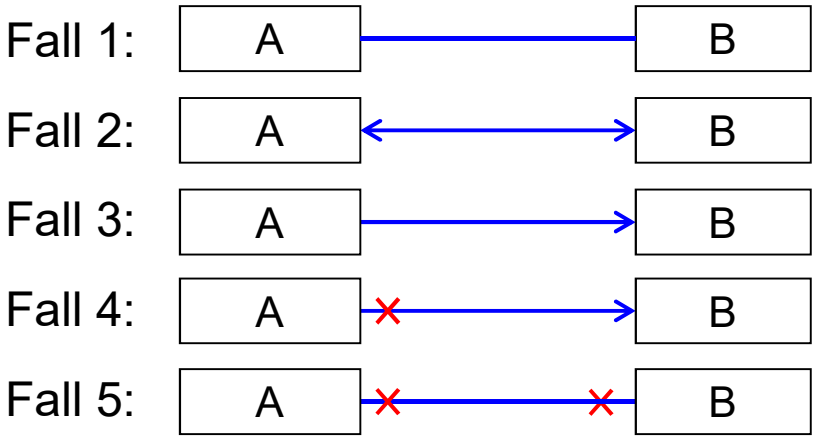
Möglich, aber wenig sinnvoll ;)

- 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

Navigierbarkeit von Assoziationen

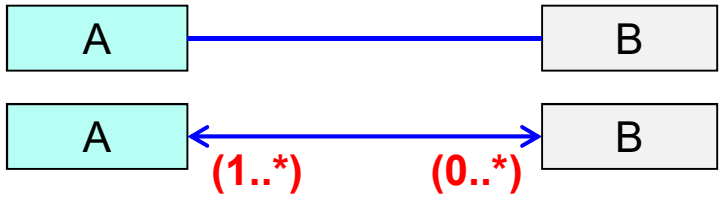
Navigierbarkeit:

- Darstellung durch Pfeilspitze und optionales Kreuz
- Pfeilspitze: Klasse A „kennt“ Klasse B



- 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

Navigierbarkeit (Java-Mapping, Fall 1+2)



Beide Richtungen (Standardfall, *unspezifiziert*)

Beide Richtungen mit Navigationsangabe

```
public class A{

  private B bb;
  . . .
}
```

```
public class B{

  private A aa;
  . . .
}
```

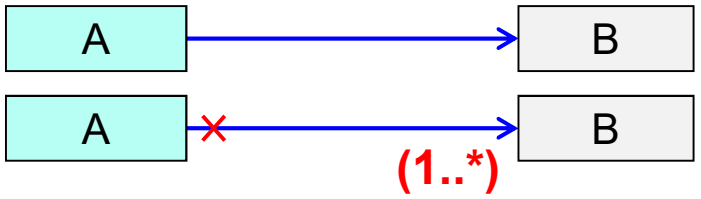
Einfache Referenz

```
public class A{

  private List<B> Bs;
  . . .
}
```

```
public class B{

  private List<A> As;
  . . .
}
```

Referenz mit Multiplizität 0..*/1..*

Navigierbarkeit (Java-Mapping, Fall 3+4)



Eine (bevorzugte) Richtung

Ausschließlich eine Richtung

Einfache Referenz:

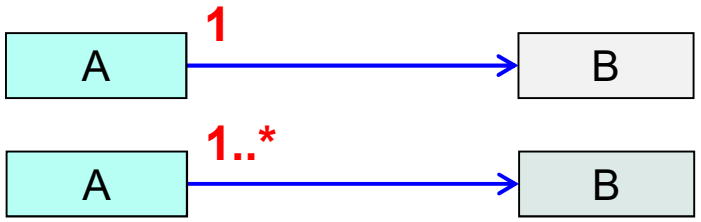
```
public class A{
  private B bb;
  . . .
}
```

```
public class B{
. . .
}
```

Referenz
mit Multiplizität „1..*“
an Pfeilspitze:

```
public class A{
  private List<B> Bs;
  . . .
}
```

Navigierbarkeit (Java-Mapping, Fall 3a)



Eine (bevorzugte) Richtung

```
public class A{
   private B bb;
   . . .
}
```

```
public class B{
 private A aa;
. . .
}
```

Referenz mit Multiplizität „1“

```
public class B{
 private List<A> aAs;
. . .
}
```

Referenz mit Multiplizität „1..*“

Navigierbarkeit (Vereinbarung)

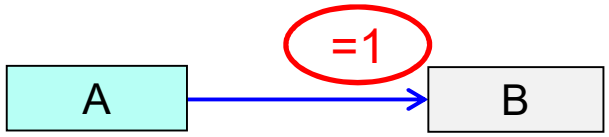
Multiplizitäten am Pfeilende tragen meist mehr zur Verwirrung als zum Verständnis bei

=> Vereinbarungen:

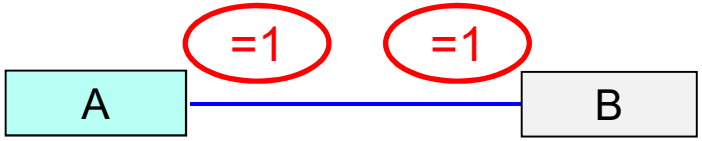
1. Multiplizitäten ausschließlich an der Pfeilspitze angeben.

2. Existiert eine Rück-Referenz (Backpointer), dann diese als bidirektionalen Pfeil oder als separaten Pfeil eintragen

3. Keine Multiplizität an der Pfeilspitze bedeutet die Multiplizität 1



4. Assoziationen ohne Pfeilspitzen und Multiplizitäten bedeuten ebenfalls die Multiplizität 1

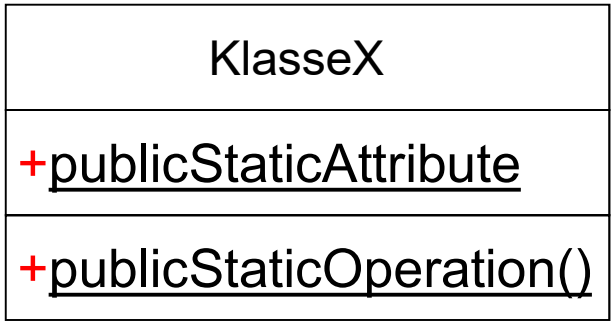


Sichtbarkeit und Eigenschaften von Attributen und Methoden

Sichtbarkeitszeichen vor den Namen des Attributs bzw. der Operation

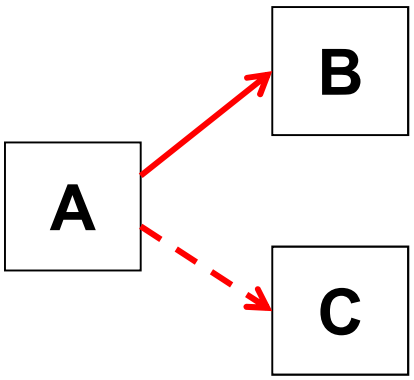
- private	-iD
+ public	+name
# protected	#vorname
~ package	~nochWas

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)

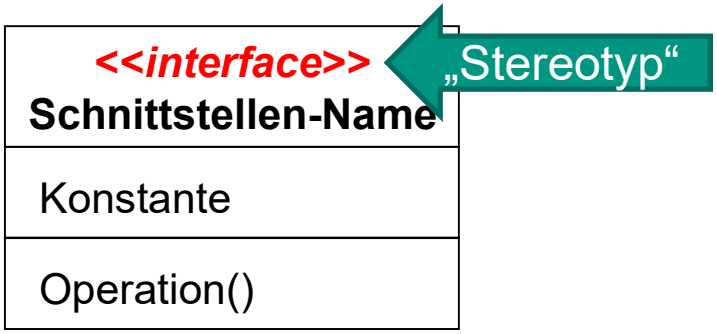



```
public class A{
   B bb;      => Klassenvariable (Referenz)

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

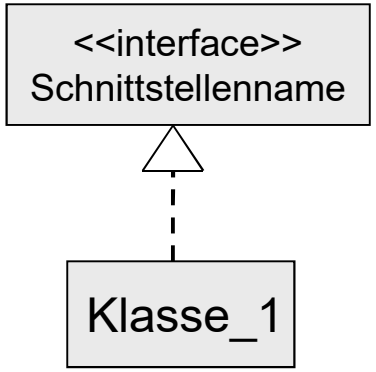
=> Abhängigkeit

Schnittstellen (Interfaces)

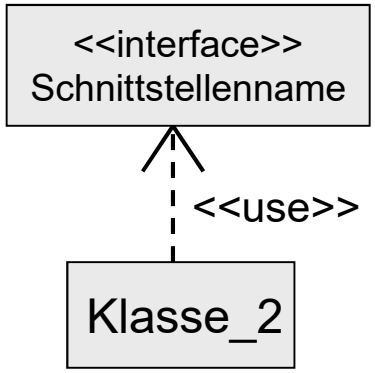
Notation:



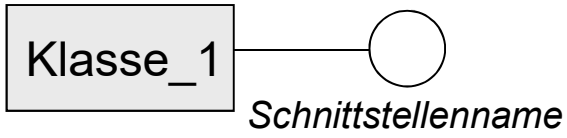
Implementierung:

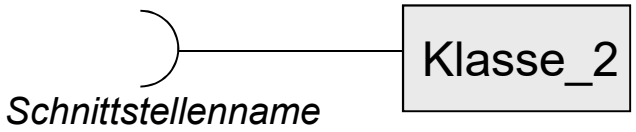


Verwendung:



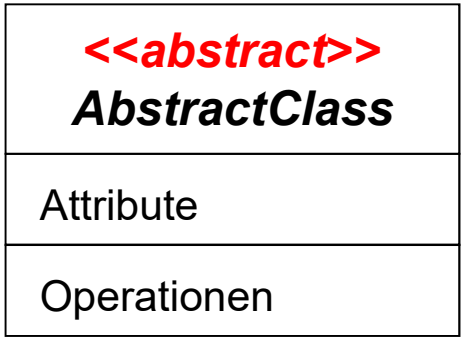
alternativ:





Abstrakte Klassen

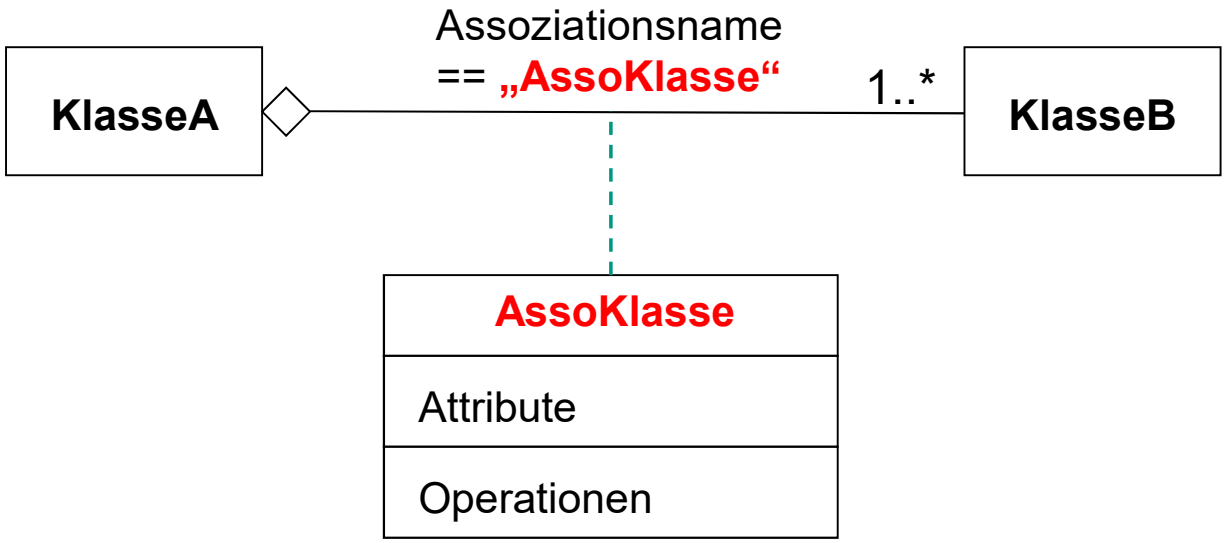
- Können (dürfen) nicht instanziiert werden
- Können Methoden implementieren und den Unterklassen zur Verfügung stellen
- Können (ebenfalls abstrakte) Methoden deklarieren, die von den Unterklassen implementiert werden **müssen** (analog zu Interfaces)
- UML: Klassensymbol mit Stereotyp *<<abstract>>*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



Besonderheiten bei UML-Klassendiagrammen

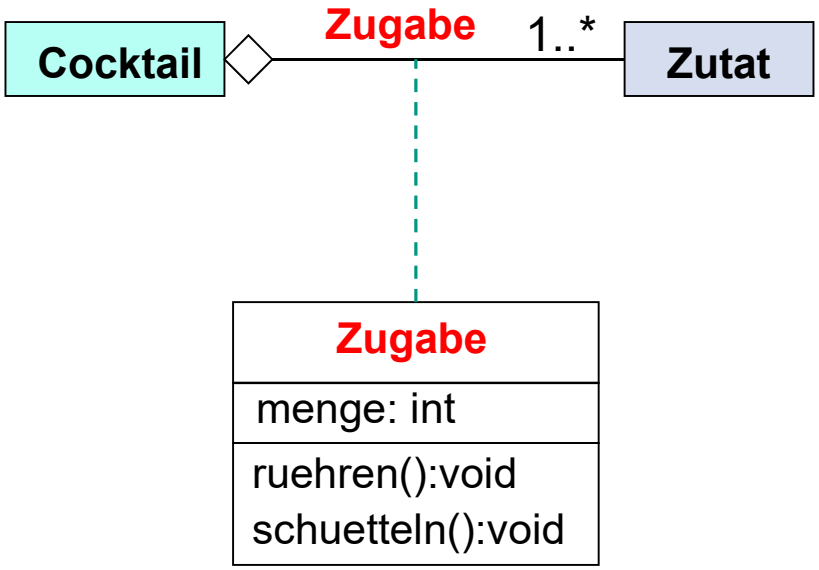
Assoziationsklassen

- 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

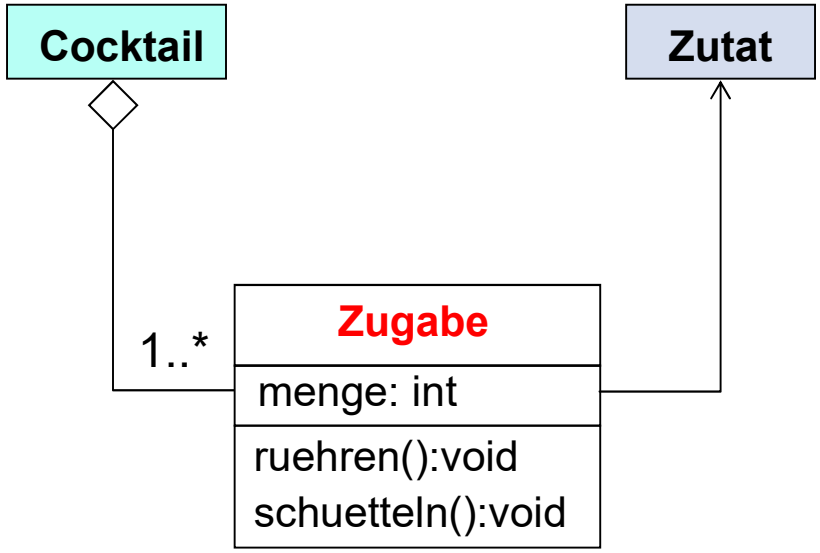


Assoziationsklassen – alternative Darstellung / Umwandlung

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

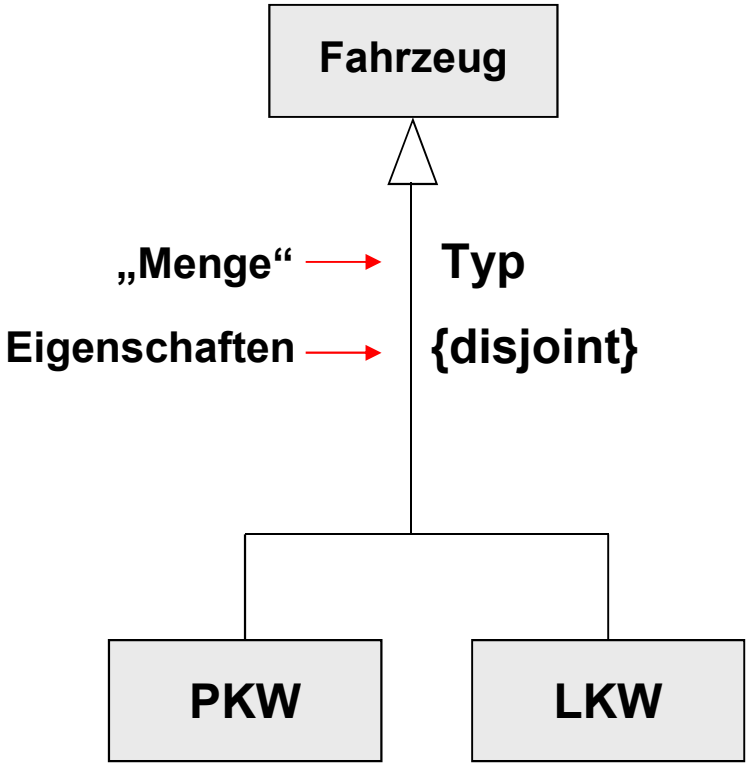


- Nach Code-Generierung (VisualParadigm™, 2014):



Generalisierungsmengen und -eigenschaften

- Bei Vererbungen/Generalisierungen können zusätzlich Bedingungen angegeben werden:



Eigenschaftswerte:

complete:
Vollständige Spezialisierung, keine Unterklassen mehr möglich / erlaubt

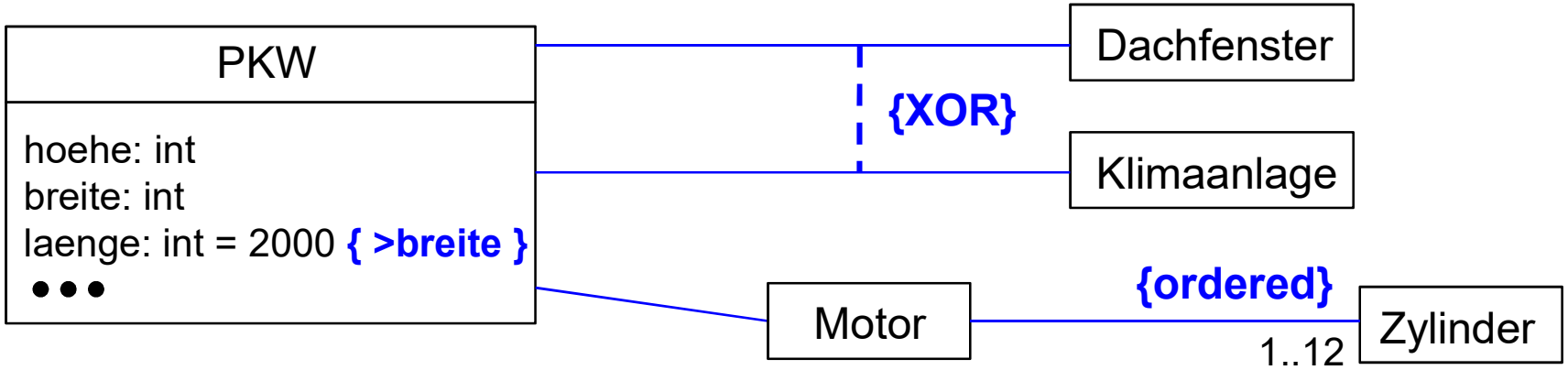
incomplete:
weitere Spezialisierungen möglich

disjoint:
Unterklassen werden klar voneinander unterschieden

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

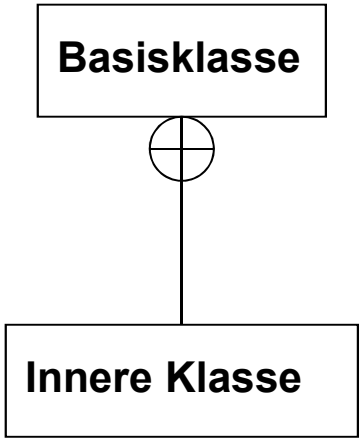
Einschränkungen (*Constraints*)

- 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:



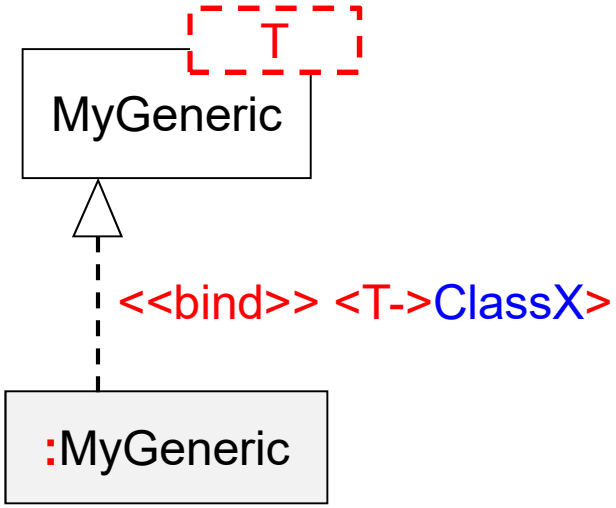
Innere Klassen

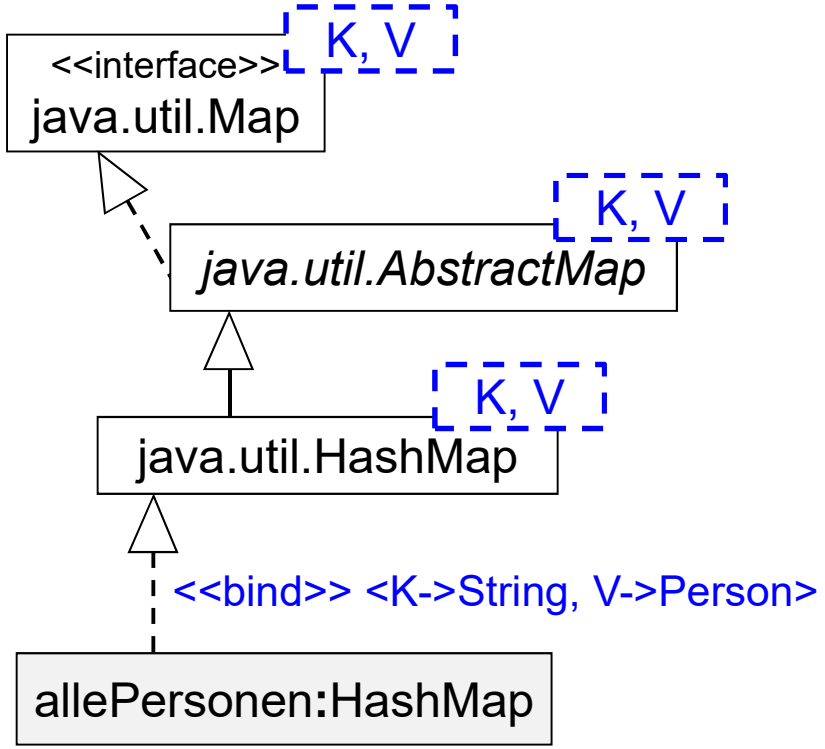
- Innere Klassen besitzen denselben Namensraum wie ihre Elternklasse
- Sie werden bei Java innerhalb der Klassendeklaration definiert



Parametrisierte Klassen (Generics)

- Definition durch gestricheltes Rechteck rechts oben am Klassensymbol
- Realisierung durch Implementierungspfeil mit Angabe der verwendeten Parameterklasse („*bind*“)
- Über mehrere Vererbungshierarchien möglich (s. *java.util.HashMap*)





Literatur

- 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

Zusammenfassung

- Erläuterung der Begriffe:
  - Objektorientierung: Objekt, Klasse, Instanz, Attribut, Operation, ..
  - Identifizieren von Objekten und Klassen
- Erklärung der 5 wichtigsten Aspekte der Objektorientierung
  - Identität
  - Klassifikation
  - Vererbung
  - Polymorphie (Polymorphismus)
  - Spätes Binden (*late binding*)