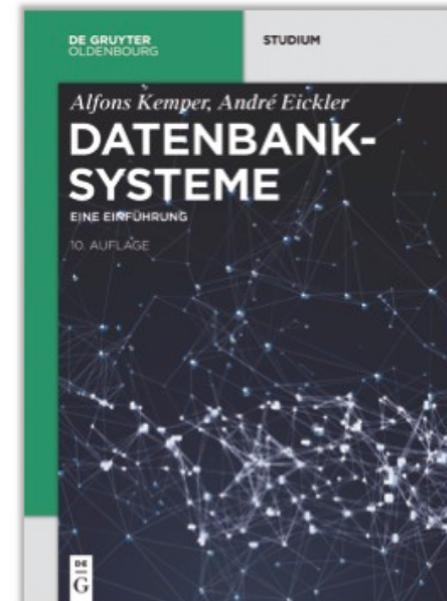
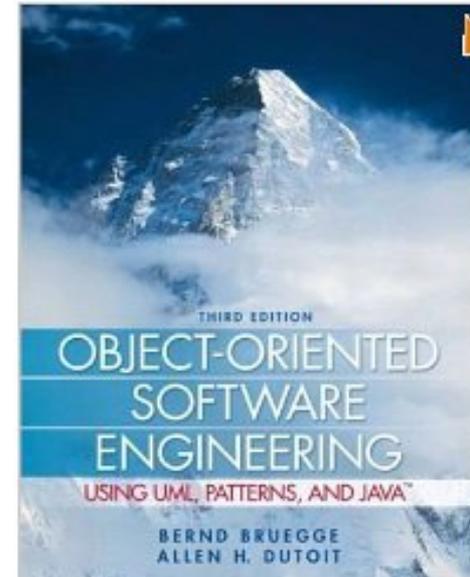


# Einführung in die Informatik II für Ingenieurwissenschaften (MSE)

- Prof. Alfons Kemper, Ph.D.
- Alexander van Renen
  
- **Teil 1:**
  - Objektorientierte Modellierung (in UML) und
  - Programmierung in Java
  
- **Teil 2:**
  - Datenbanksysteme: Eine Einführung
  - Alfons Kemper und Andre Eickler
  - Oldenbourg Verlag, 10. Auflage, 2016



# **Vorlesung: Prof. Alfons Kemper**

**alfons.kemper@in.tum.de**



# Übungsbetrieb

- Übungsleitung: Alexander van Renen ([renen@in.tum.de](mailto:renen@in.tum.de))
- Tutoren:
  - Aaron Tacke
    - E-Mail: [tacke@in.tum.de](mailto:tacke@in.tum.de)
    - BBB Room: <https://bbb.rbg.tum.de/ale-pt4-zhf>
  - Florian Gebauer
    - E-Mail: [florian.gebauer@tum.de](mailto:florian.gebauer@tum.de)
    - BBB Room: <https://bbb.rbg.tum.de/ale-kc2-9ve>



# Organisatorisches I

- Unsere Webseite: <https://db.in.tum.de/teaching/ss20/ei2>
- Vorlesung:
  - Video-Stream: Montags 10:00 in unserem YouTube Channel: <https://www.youtube.com/channel/UC6ubXOiMJfovYlwQcAMZLPw>
  - Download: Moodle, YouTube, und Vorlesungswebseite
- Übung:
  - Übung Live-Stream über Zoom: Montags 11:30
- Fragestunde:
  - Live-Streams über Big Blue Button: <bbb.rbg.tum.de>

# Organisatorisches II

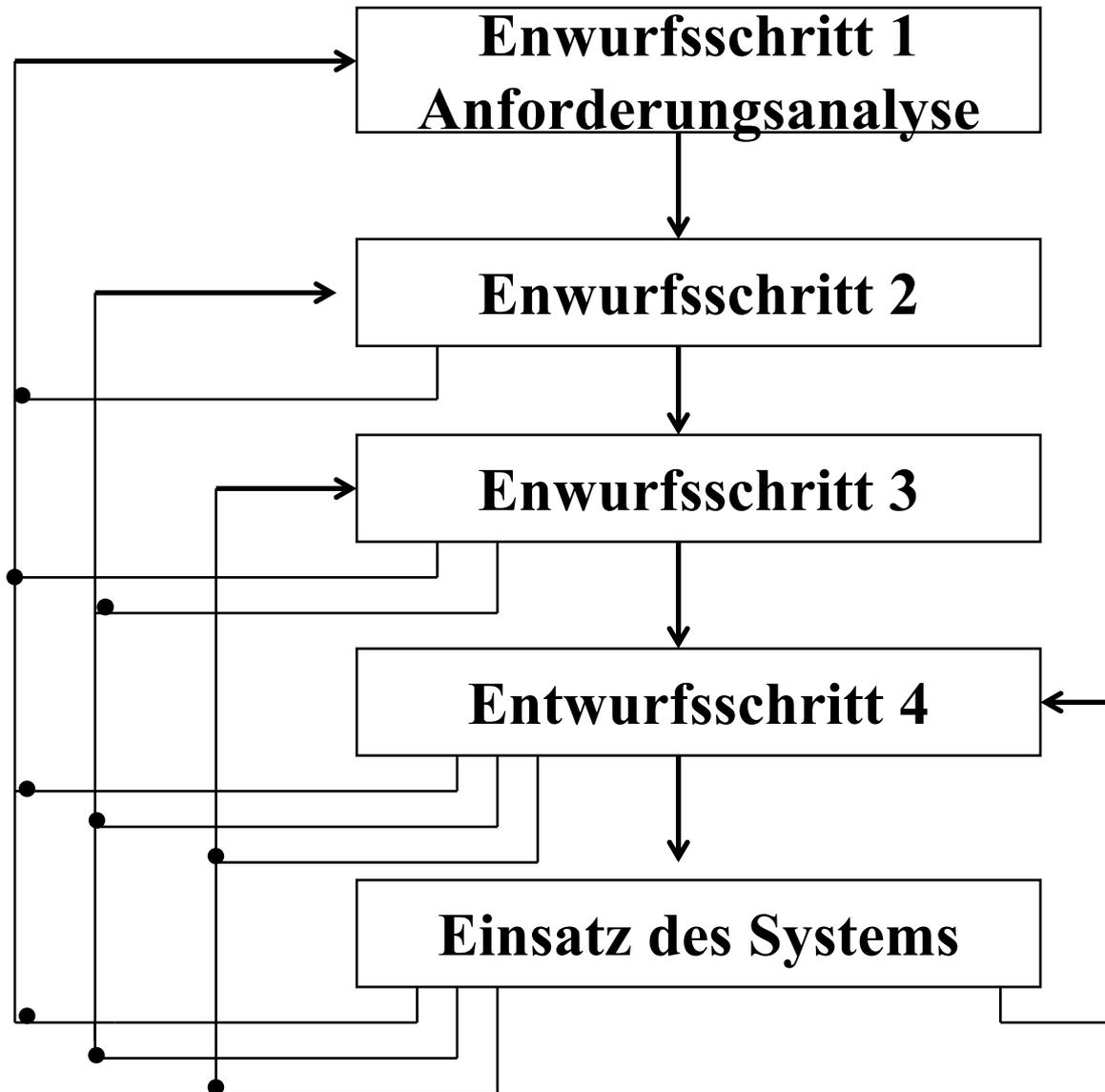
- Material (Folien & Übungsblätter):
  - Moodle und Vorlesungswebseite
- Chat Plattform (für Fragen und Diskussionen):
  - Setup: <https://wiki.in.tum.de/Informatik/Helpdesk/RIOT>
  - Raum: <https://riot.im/app/#/room/#mseinfo2:tum.de>
- Prüfungstermin:
  - Vorläufige Datum: Donnerstag, 27.08.2020 um 11:00

# Datenbankentwurf

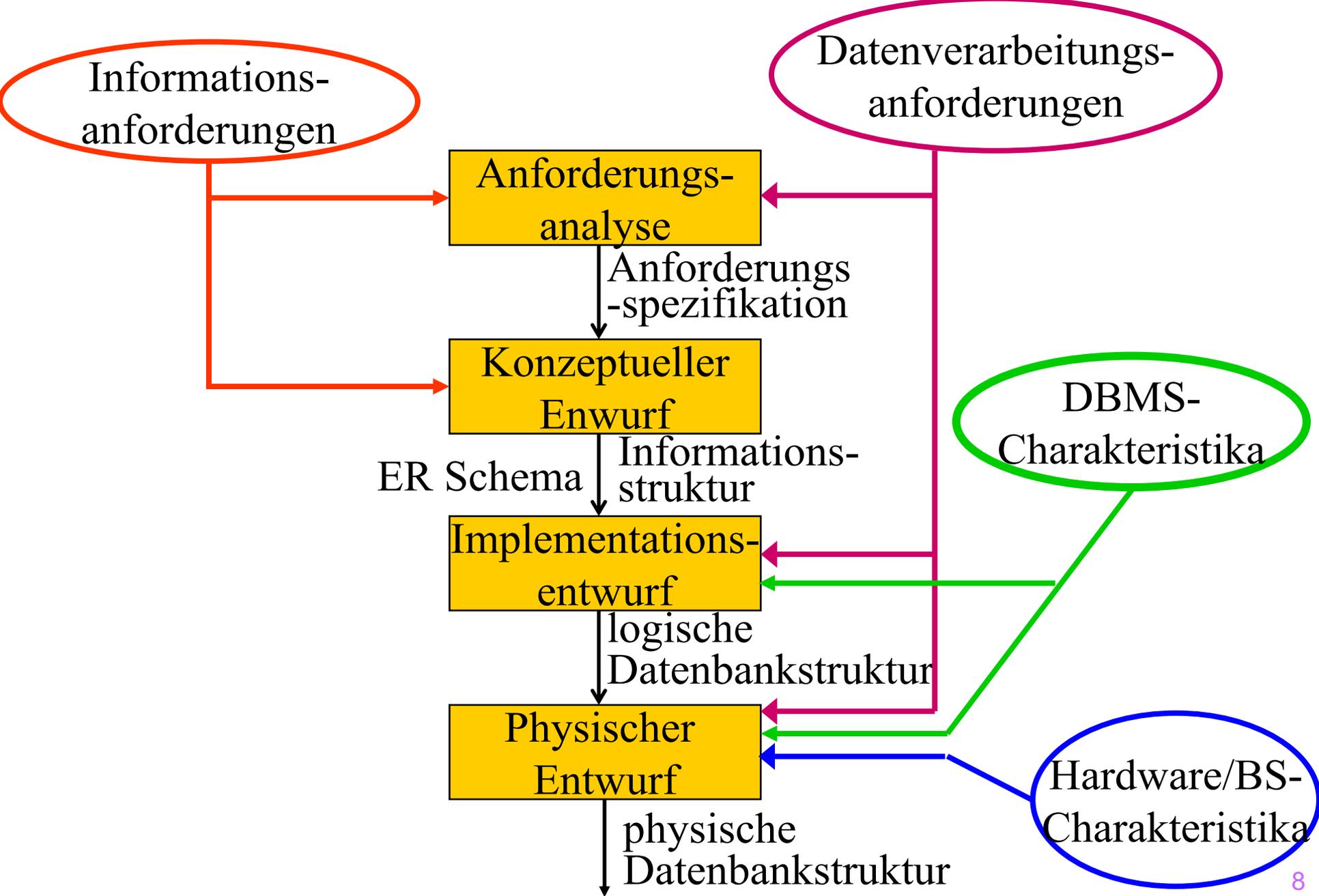
## Abstraktionsebenen des Datenbankentwurfs

1. Konzeptuelle Ebene
2. Implementationsebene
3. Physische Ebene

# Allgemeiner „top-down Entwurf“



# Phasen des Datenbankentwurfs



# Anforderungsanalyse

1. Identifikation von Organisationseinheiten
2. Identifikation der zu unterstützenden Aufgaben
3. Anforderungs-Sammelplan
4. Anforderungs-Sammlung
5. Filterung
6. Satzklassifikationen
7. Formalisierung

# Objektbeschreibung

## ● Uni-Angestellte

- Anzahl: 1000
- Attribute

### ❖ Personalnummer

- Typ: char
- Länge: 9
- Wertebereich:  
0...999.999.99
- Anzahl  
Wiederholungen: 0
- Definiertheit: 100%
- Identifizierend: ja

### ❖ Gehalt

- Typ: dezimal
- Länge: (8,2)
- Anzahl Wiederholung: 0
- Definiertheit: 10%
- Identifizierend: nein

### ❖ Rang

- Typ: String
- Länge: 4
- Anzahl Wiederholung: 0
- Definiertheit: 100%
- Identifizierend: nein

# Beziehungsbeschreibung: *prüfen*

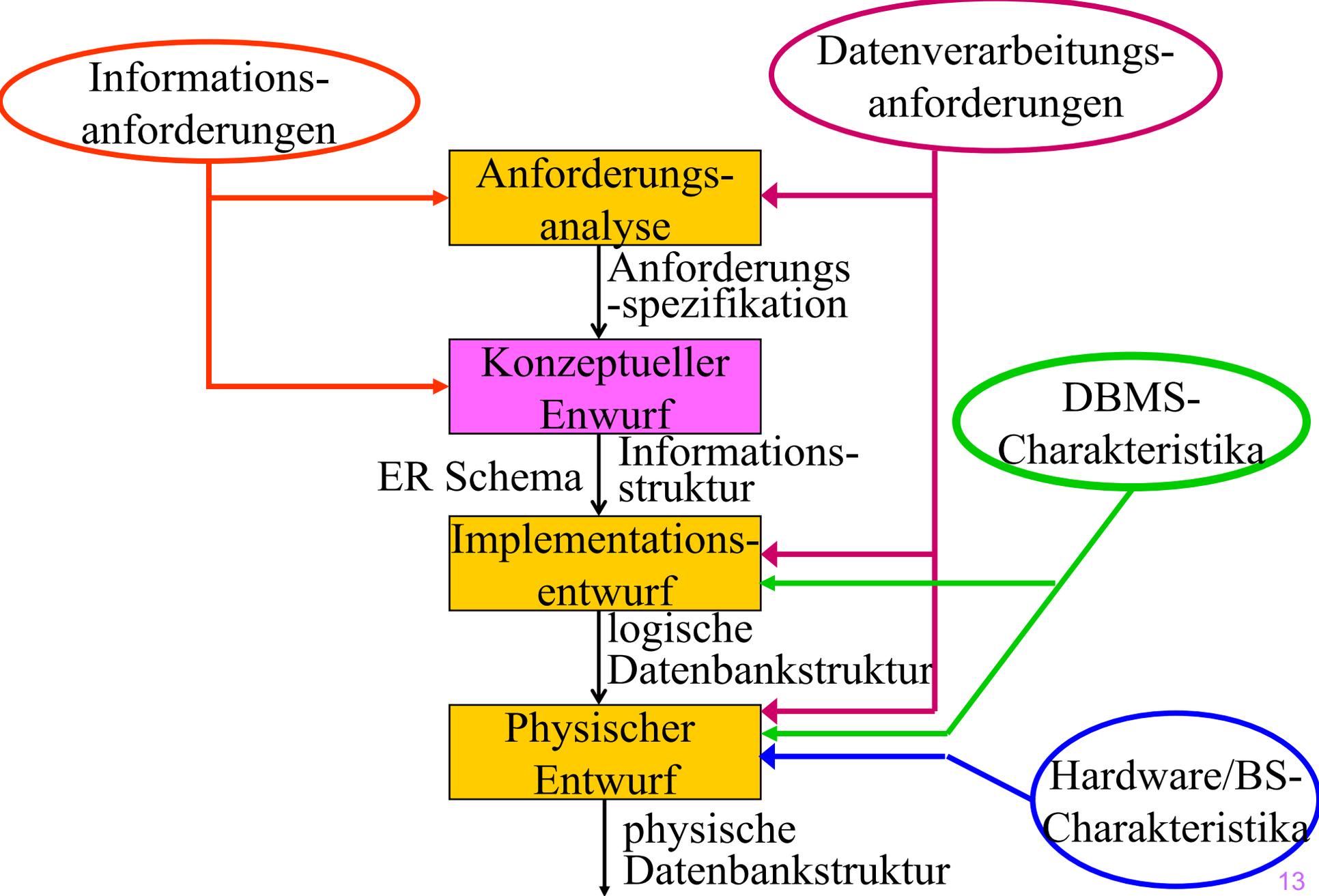
- Beteiligte Objekte:
  - Professor als Prüfer
  - Student als Prüfling
  - Vorlesung als Prüfungsstoff
- Attribute der Beziehung:
  - Datum
  - Uhrzeit
  - Note
- Anzahl: 100 000 pro Jahr

# Prozeßbeschreibungen

## ● Prozeßbeschreibung: *Zeugnisausstellung*

- Häufigkeit: halbjährlich
- benötigte Daten
  - \* Prüfungen
  - \* Studienordnungen
  - \* Studenteninformation
  - \* ...
- Priorität: hoch
- Zu verarbeitende Datenmenge
  - \* 500 Studenten
  - \* 3000 Prüfungen
  - \* 10 Studienordnungen

# Phasen des Datenbankentwurfs



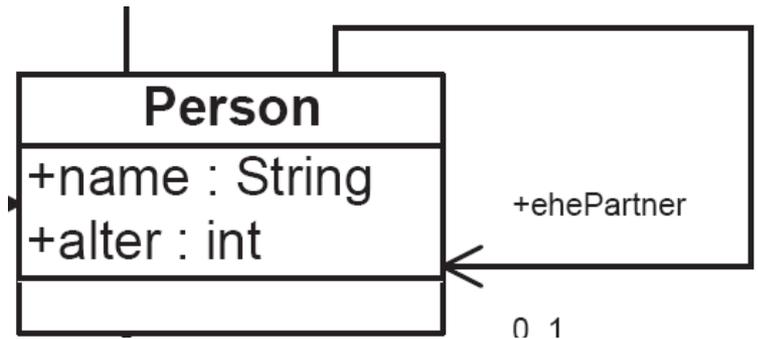
# Datenmodellierung mit UML

- Unified Modelling Language UML
- De-facto Standard für den objekt-orientierten Software-Entwurf
- Zentrales Konstrukt ist die Klasse (class), mit der gleichartige Objekte hinsichtlich
  - Struktur (~Attribute)
  - Verhalten (~Operationen/Methoden)modelliert werden
- Assoziationen zwischen Klassen entsprechen Beziehungstypen
- Generalisierungshierarchien
- Aggregation

# Klassen/Objekttypen in Java

```
class TypName {  
    Typ1 Attr1 ;  
    ...  
    Typn Attrn ;  
    // Operationen folgen hier  
    ...  
} // end class TypName;
```

```
class Person {  
    public String name;  
    public int alter;  
    public Person ehePartner;  
}
```



# Werte versus Objekte

Typen		Instanzen
primitive Typen	→	Werte
Objekttypen (Klassen)	→	Objekte

Sorte	Wertebereich
<b>boolean</b>	{ <i>true</i> , <i>false</i> }
<b>byte</b>	sehr kleine Integer-Zahlen
<b>short</b>	kleine Integer-Zahlen
<b>int</b>	Integer-Zahlen
<b>long</b>	große Integer-Zahlen
<b>float</b>	Fließkomma-Zahlen
<b>double</b>	Fließkomma-Zahlen doppelter Präzision
<b>char</b>	Character

# Java Klassendefinition: Syntax

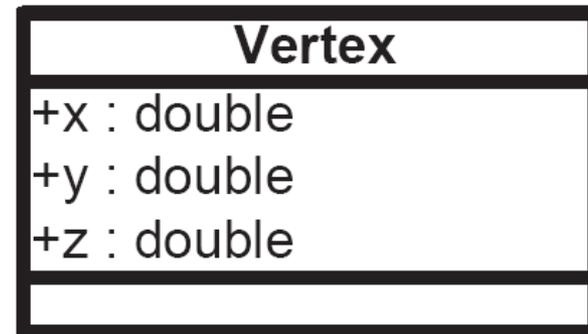
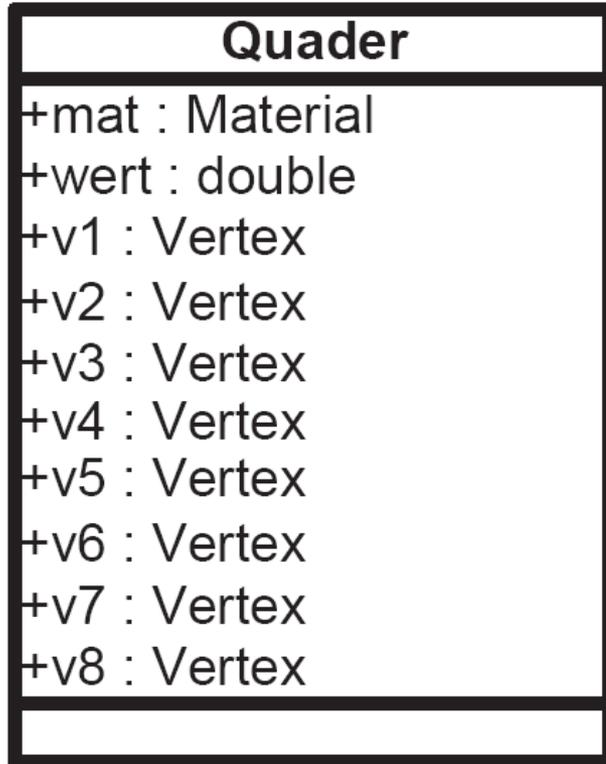
```
[public] [abstract] class A
    [extends B] [implements Schnittstellen] {
        Instanz-Variable;
        . . .
        Instanz-Variable;

        Konstruktor;
        . . .
        Konstruktor;

        Operation/Methode;
        . . .
        Operation/Methode;
    }
```

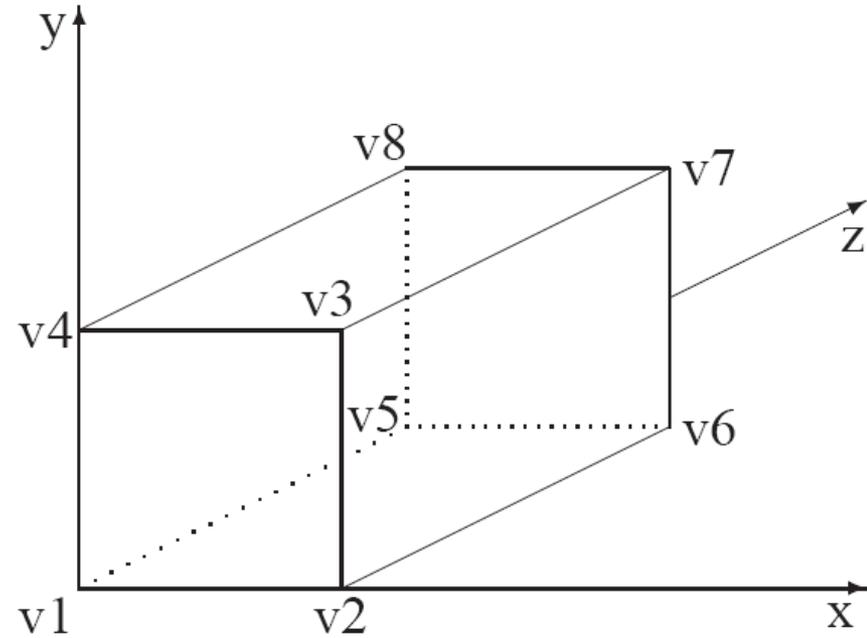
# Klassen/Objekttypen in UML

---



# Klassen in Java

```
class Vertex {  
    public double x;  
    public double y;  
    public double z;  
}  
  
public class Material {  
    public String name;  
    public double spezGewicht;  
}  
  
class Quader {  
    public Vertex v1, v2, v3, v4, v5, v6, v7, v8;  
    public Material mat;  
    public double wert;  
}
```



# Instanziierung

- *int, short, byte, long*-Attribute werden auf den Wert 0 initialisiert.
- *double, float*-Attribute werden initial auf den Wert 0.0 gesetzt.
- *char*-Instanzvariablen werden auf den Wert „\u0000“ initialisiert.
- *boolean*-Attribute werden auf *false* gesetzt.
- Attribute, die auf einen Objekttyp eingeschränkt sind, werden auf *null* gesetzt. Dies ist ein spezieller Wert, der angibt, dass (noch) keine Referenz auf ein Objekt existiert.
- *String*-Attribute werden initial auch auf *null* gesetzt, da Strings in Java als Objekttyp definiert sind.

# Instanziierung eines Quaders

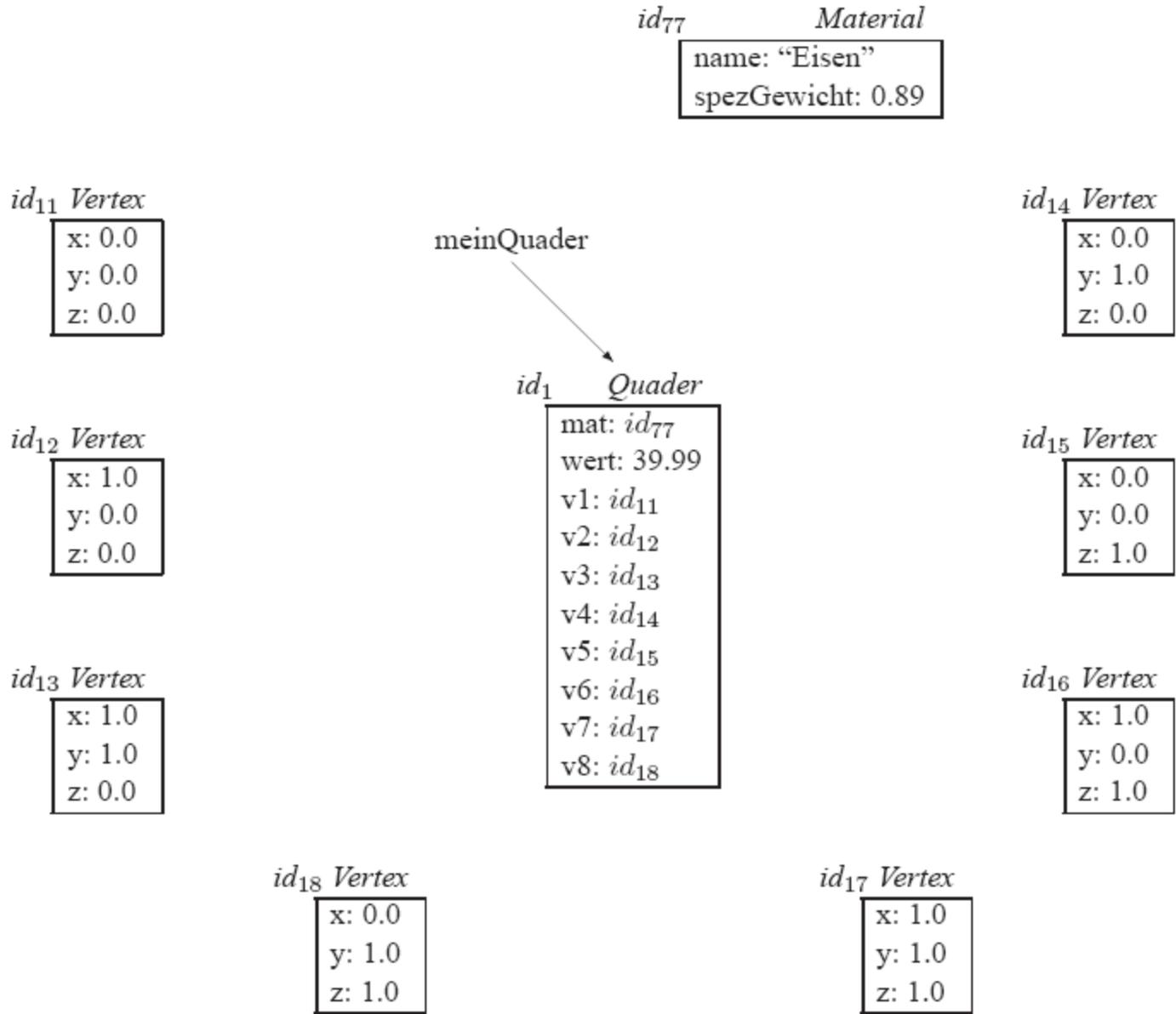
```
Quader meinQuader;  
meinQuader = new Quader();  
  
meinQuader.v1 = new Vertex();  
meinQuader.v1.x = 0.0;  
meinQuader.v1.y = 0.0;  
meinQuader.v1.z = 0.0;  
meinQuader.v2 = new Vertex();  
... // instanziiere und initial  
meinQuader.v8 = new Vertex();  
meinQuader.v8.x = 0.0;  
meinQuader.v8.y = 1.0;  
meinQuader.v8.z = 1.0;  
meinQuader.wert = 39.99;  
  
meinQuader.mat = new Material();  
meinQuader.mat.name = "Eisen";  
meinQuader.mat.spezGewicht = 0.89;
```

*meinQuader*

*id<sub>1</sub> Quader*

<i>mat: null</i>
<i>value: 0.0</i>
<i>v1: null</i>
<i>v2: null</i>
<i>v3: null</i>
<i>v4: null</i>
<i>v5: null</i>
<i>v6: null</i>
<i>v7: null</i>
<i>v8: null</i>

# Resultierendes Objektnetz



# Objekt besteht aus (OID, Typ, Rep)

Jedes Objekt  $o$  kann man demnach als Tripel der folgenden Form auffassen:

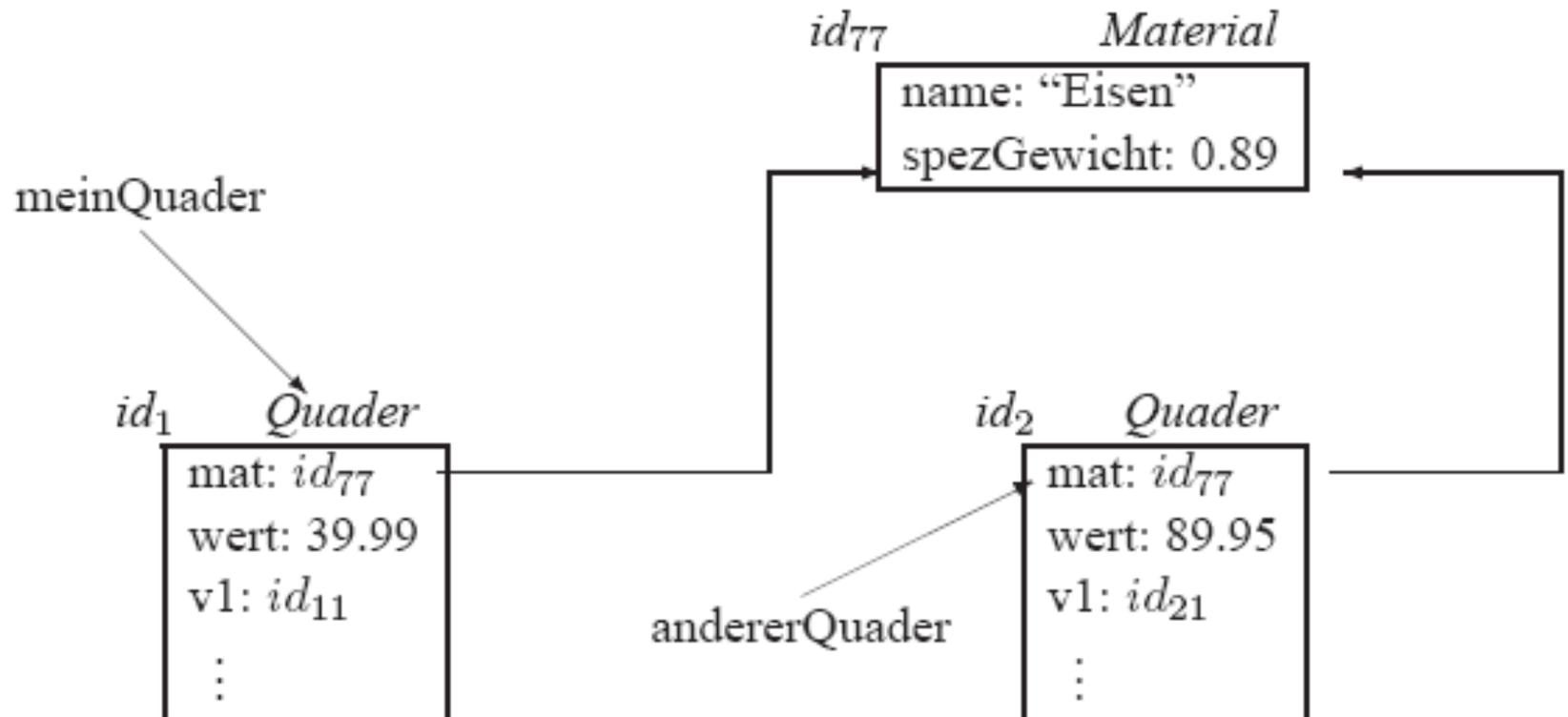
$$o = (id_{\#}, Typ, Rep)$$

Die drei Teile haben die folgende Bedeutung:

- $id_{\#}$  stellt den Objektidentifikator des Objekts  $o$  dar.
  - $Typ$  spezifiziert den Objekttyp, von dem das Objekt instanziiert wurde.
  - $Rep$  entspricht dem internen Zustand (der derzeitigen strukturellen Repräs des Objekts  $o$ ).
- Als OID dient in Java die (virtuelle) Speicheradresse
    - Nennt man physische OID
  - In Datenbanken verwendet man auch logische OIDs
    - Damit Objekte sich „bewegen“ können

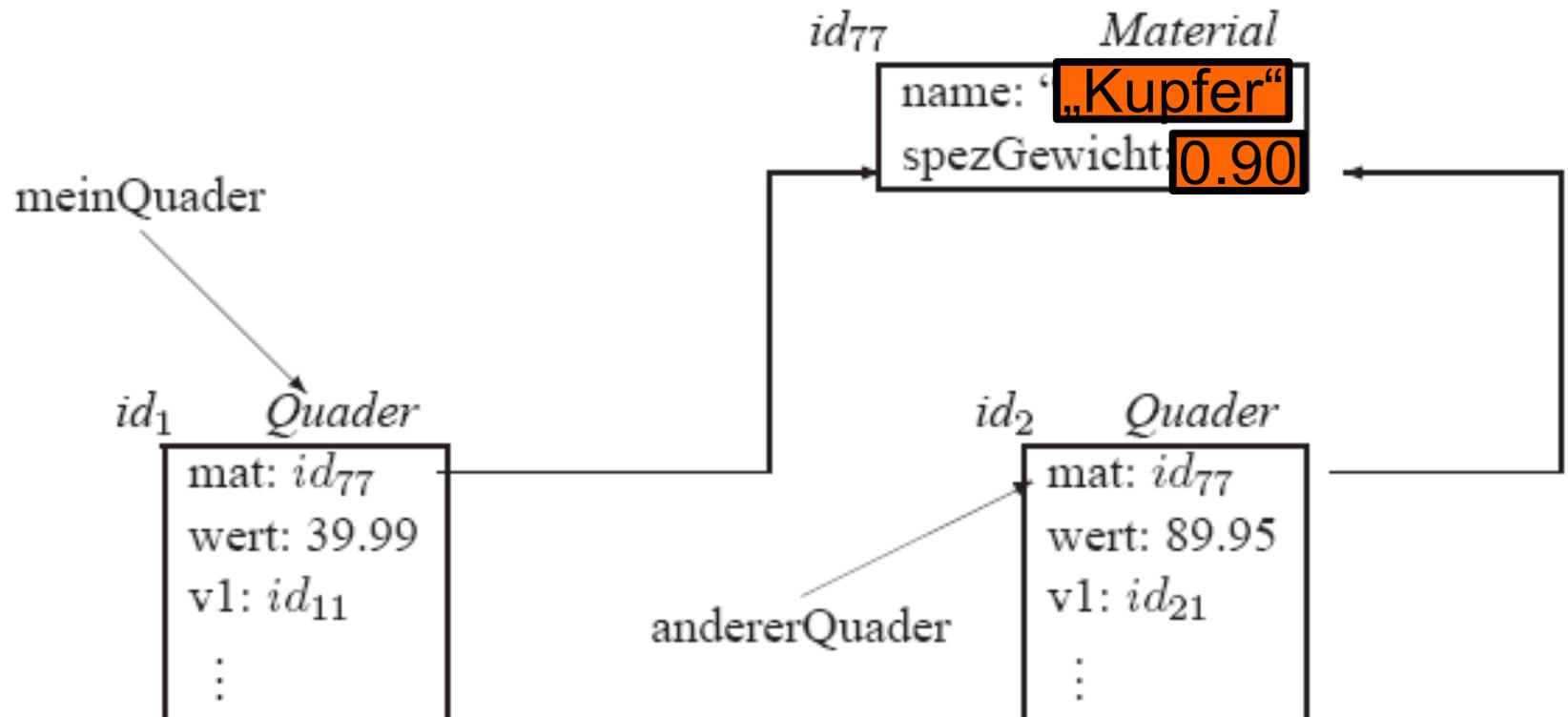
# Shared Subobjects/Gemeinsame Unterobjekte

```
Quader andererQuader;  
...  
andererQuader = new Quader();  
andererQuader.mat = meinQuader.mat;
```



# Shared Subobjects/Gemeinsame Unterobjekte

```
meinQuader.mat.name = "Kupfer";  
meinQuader.mat.spezGewicht = 0.90;
```

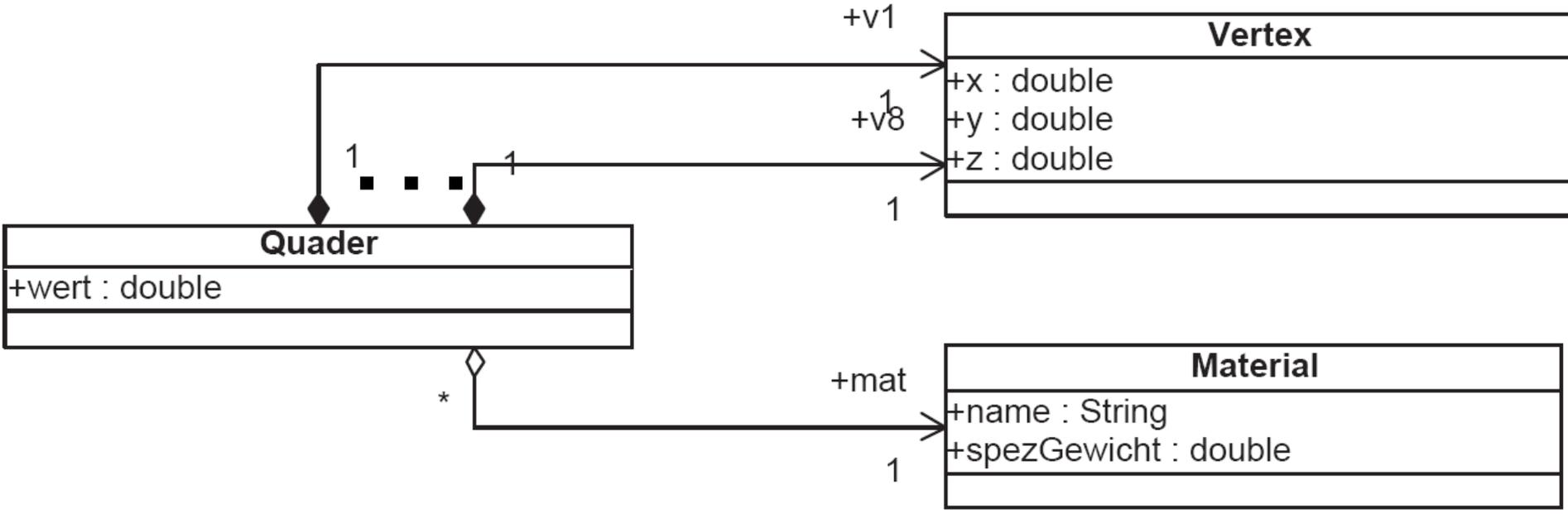


# Wertvergleich versus Objektvergleich

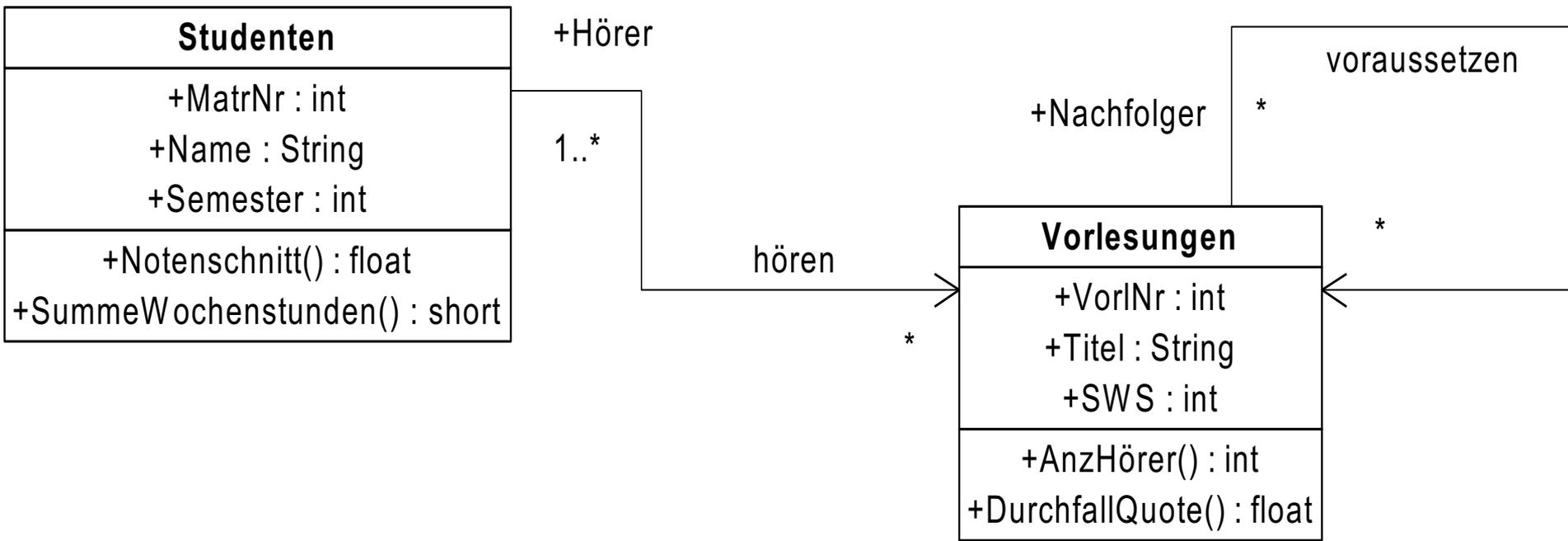
```
andererQuader.mat.name.equals("Kupfer");    // Objekt-Vergleich  
andererQuader.mat.spezGewicht == 0.90;    // Wert-Vergleich
```

- Dasselbe ist nicht dasgleiche!
- Im Restaurant sollte man nie „dasselbe“ sondern „dasgleiche“ wie ein anderer bestellen

# Beziehungen/Assoziationen in UML



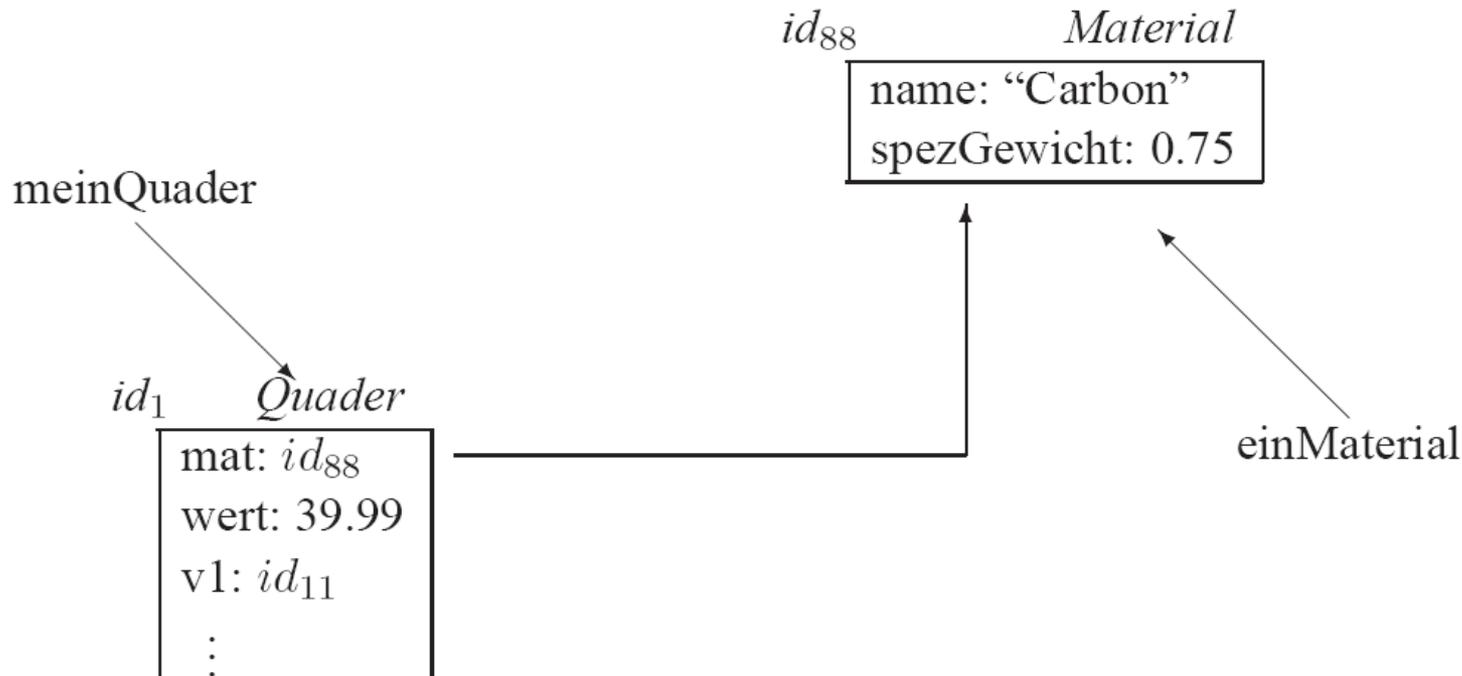
# Klassen und Assoziationen



# Referenzierung/Dereferenzierung

```
Quader meinQuader = new Quader();  
Material einMaterial;  
double w;  
...
```

- (1) `einMaterial = new Material();` // *einMaterial* spei
- (2) `einMaterial.name = "Carbon";`
- (3) `einMaterial.spezGewicht = 0.75;`
- (4) `meinQuader.mat = einMaterial;` // *meinQuader* hat
- (5) `w = meinQuader.mat.spezGewicht;`



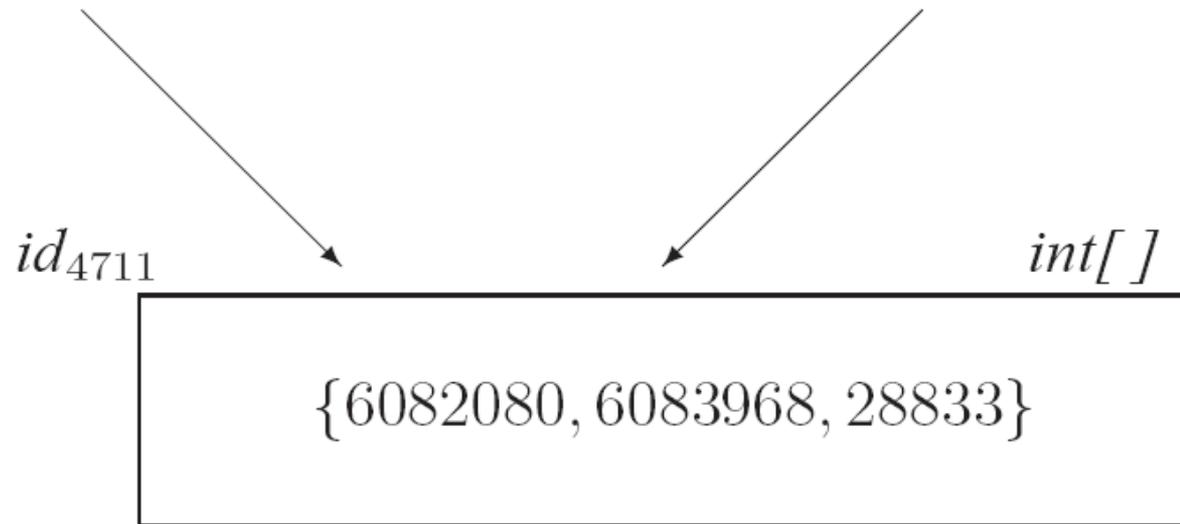


# Kollektion als shared subobject

```
int [] dieJavaSupportHotLine;  
dieJavaSupportHotLine = kempersTelefonNummern;
```

kempersTelefonNummern

dieJavaSupportHotLine

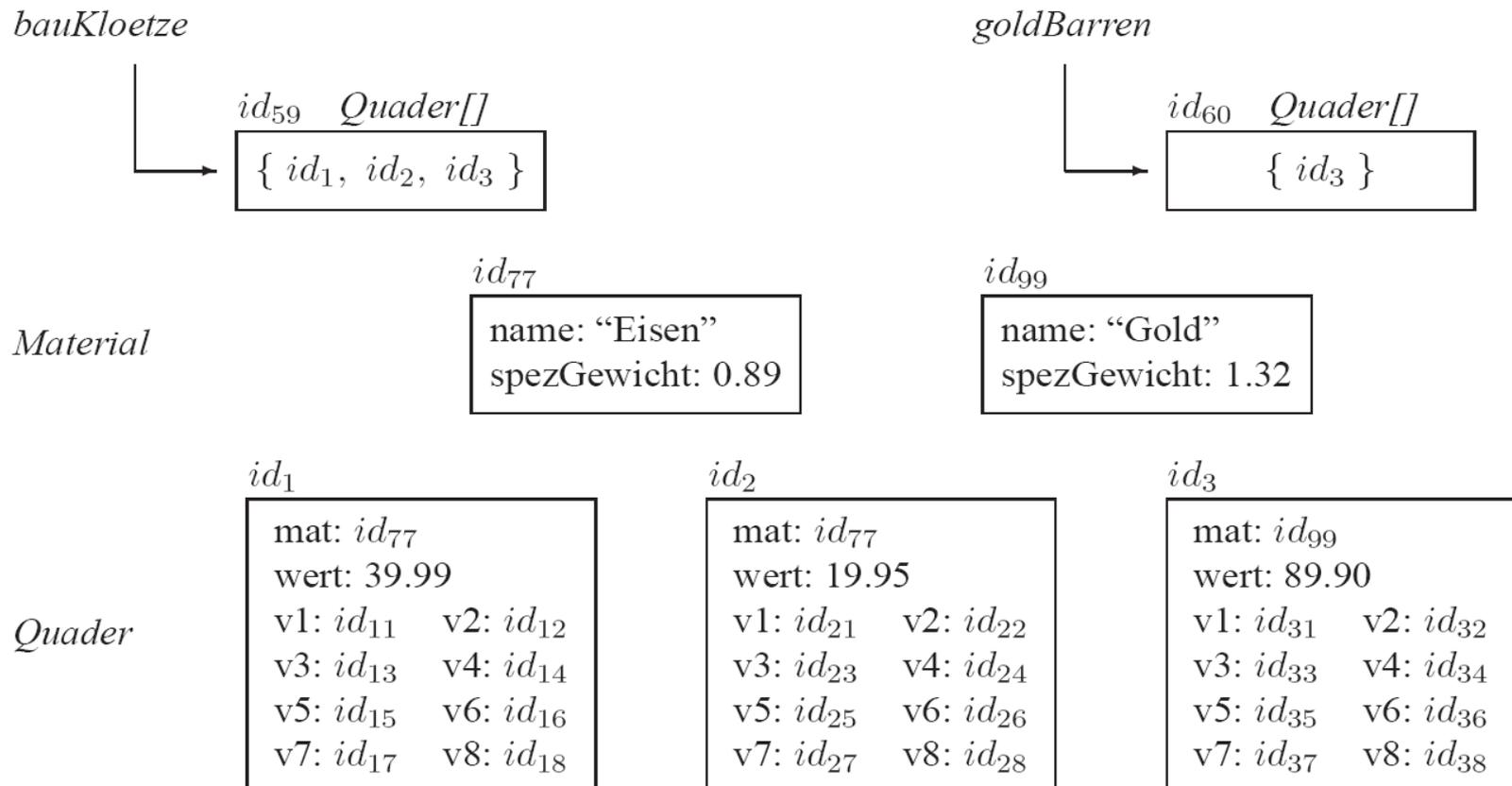


```
dieJavaSupportHotLine[1] = 6083968;
```

# Kollektionen sind „first class citizens“

```
Quader[] bauKloetze = new Quader[3];  
Quader[] goldBarren = new Quader[1];
```

```
Quader meinQuader = new Quader();  
bauKloetze[0] = meinQuader;  
goldBarren[0] = ...;
```



# Kollektion natürlich auch als Typ einer Instanzvariablen möglich ...

```
class Quader2 {  
    public Vertex[] eckPunkte;  
    public Material mat;  
    public double wert;  
}
```

```
einQuader.eckPunkte = new Vertex[8];
```

*id*<sub>299</sub>      *Quader2*

mat: <i>id</i> <sub>77</sub>
wert: 250.00
eckPunkte: <i>id</i> <sub>899</sub>

*id*<sub>899</sub>      *Vertex[]*

{ <i>id</i> <sub>111</sub> , <i>id</i> <sub>112</sub> , <i>id</i> <sub>113</sub> , <i>id</i> <sub>114</sub> , <i>id</i> <sub>115</sub> , <i>id</i> <sub>116</sub> , <i>id</i> <sub>117</sub> , <i>id</i> <sub>118</sub> }
---

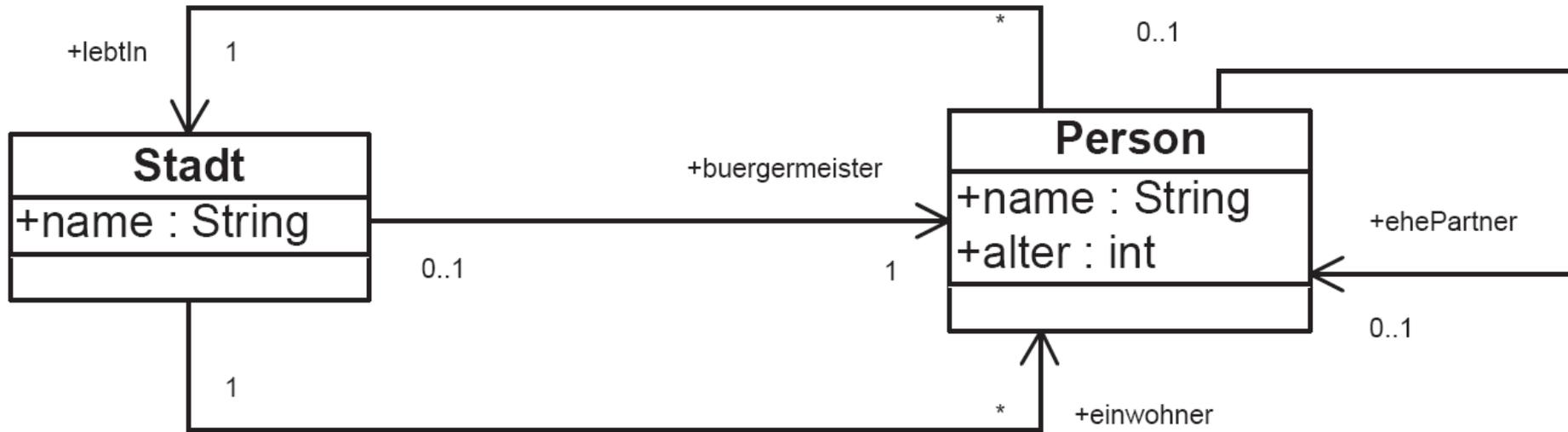


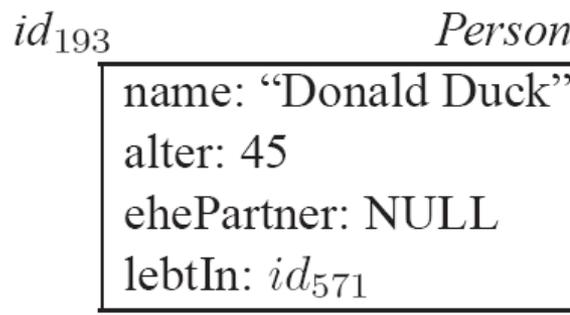
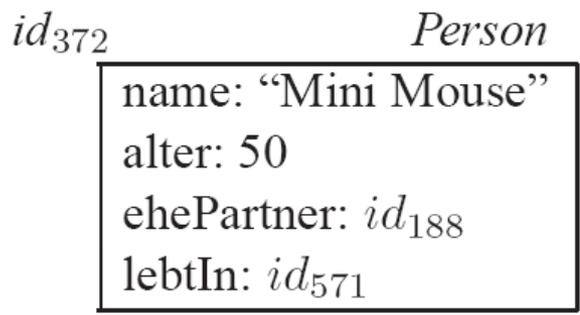
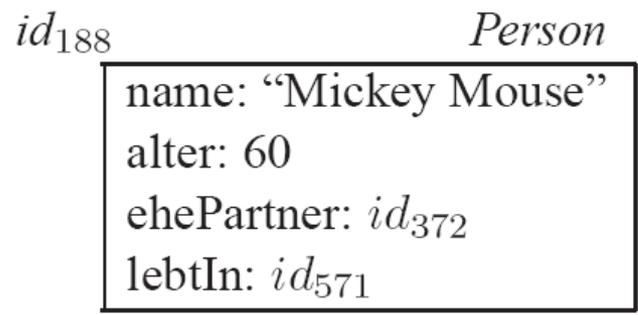
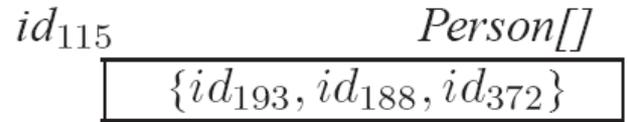
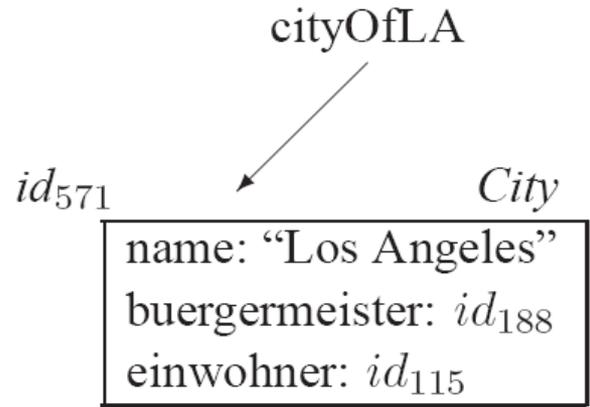
Abbildung 1.12: Die Assoziationen zwischen Person und Stadt

```

class Person {
    public String name;
    public int alter;
    public Person ehePartner;
    public Stadt lebtIn;
}

class Stadt {
    public String name;
    public Person buergermeister;
    public Person[] einwohner;
}
  
```

# Objekt-Netz



# Typisierung von (Pfad-)Ausdrücken

```
int gesamtAlter, alterVonJemand;  
Person jemand;  
String name;  
...
```

(1) `alterVonJemand = cityOfLA.buergermeister.ehePartner.alter;`

(2) `for (int i = 0; i < cityOfLA.einwohner.length; i++) {  
 jemand = cityOfLA.einwohner[i];  
 gesamtAlter = gesamtAlter + jemand.alter;  
}`

$\underbrace{\text{alterVonJemand}}_{\text{int}} = \underbrace{\text{cityOfLA}}_{\text{Stadt}} \cdot \underbrace{\text{.buergermeister.ehePartner}}_{\text{Person}} \cdot \underbrace{\text{.alter}}_{\text{Person}};$

$\underbrace{\hspace{15em}}_{\text{int}}$

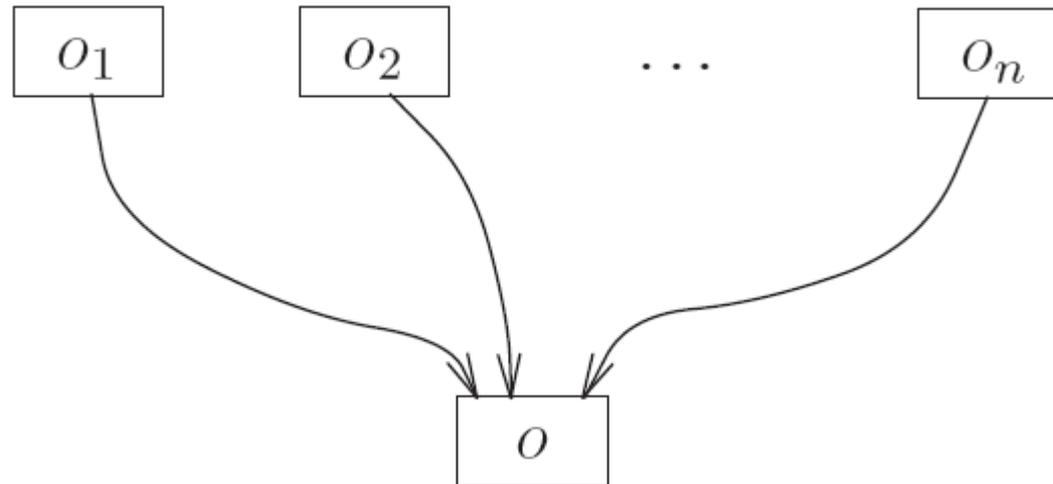


# Typisierung ... cont'd

Person  
Person[]  
Person Stadt  
jemand = cityOfLA.einwohner[i];  
gesamtAlter = gesamtAlter + jemand.alter;  
int int Person  
int

# Speicherbereinigung / Garbage Collection

- Automatisch in Java
- Nur unerreichbare Objekte dürfen gelöscht werden
- Erst wenn die letzte Referenz auf ein Objekt entfernt wurde, darf der garbage collector „zuschlagen“



# Klassen-Attribute

```
class Quader {  
    public static final int anzahlKanten = 12;  
    public static final int anzahlEcken = 8;  
    public static int anzahlQuader = 0;  
    public Vertex v1, v2, v3, v4, v5, v6, v7, v8;  
    public Material mat;  
    public double wert;  
}
```

anzahlKanten: 12

anzahlEcken: 8

anzahlQuader: 3

*Quader*

*id<sub>1</sub>*

mat: *id<sub>77</sub>*

wert: 39.99

v1: *id<sub>11</sub>*    v2: *id<sub>12</sub>*

v3: *id<sub>13</sub>*    v4: *id<sub>14</sub>*

v5: *id<sub>15</sub>*    v6: *id<sub>16</sub>*

v7: *id<sub>17</sub>*    v8: *id<sub>18</sub>*

*id<sub>2</sub>*

mat: *id<sub>77</sub>*

wert: 19.95

v1: *id<sub>21</sub>*    v2: *id<sub>22</sub>*

v3: *id<sub>23</sub>*    v4: *id<sub>24</sub>*

v5: *id<sub>25</sub>*    v6: *id<sub>26</sub>*

v7: *id<sub>27</sub>*    v8: *id<sub>28</sub>*

*id<sub>3</sub>*

mat: *id<sub>99</sub>*

wert: 89.90

v1: *id<sub>31</sub>*    v2: *id<sub>32</sub>*

v3: *id<sub>33</sub>*    v4: *id<sub>34</sub>*

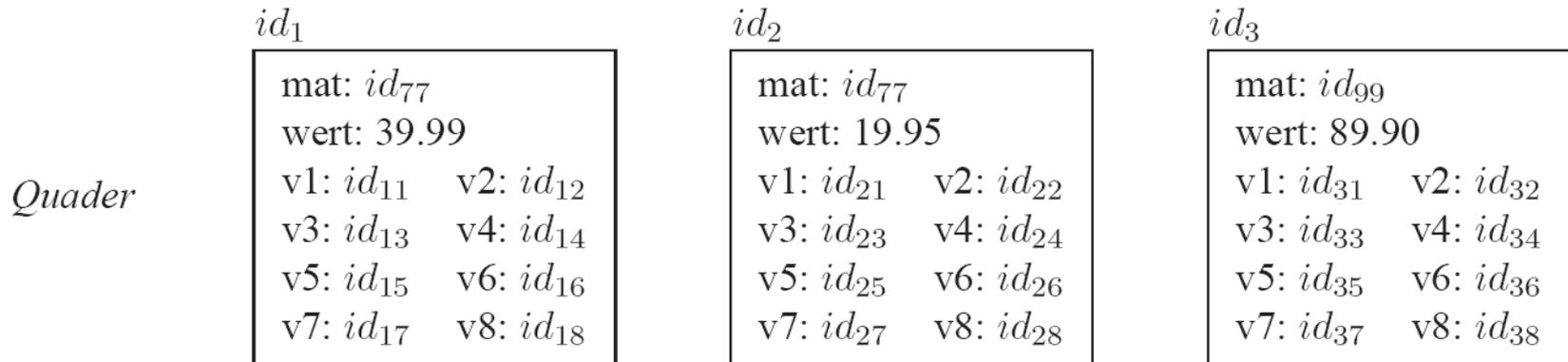
v5: *id<sub>35</sub>*    v6: *id<sub>36</sub>*

v7: *id<sub>37</sub>*    v8: *id<sub>38</sub>*

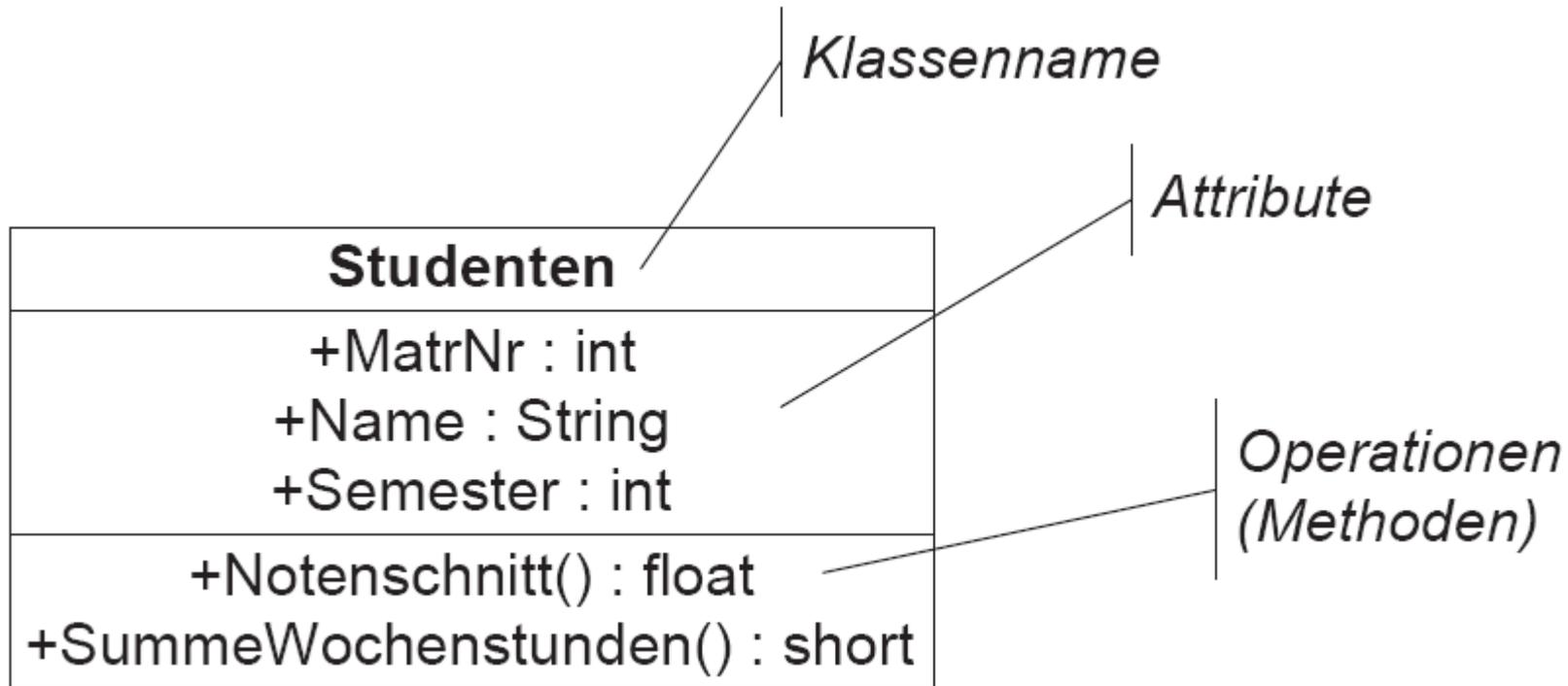
# Klassen-Attribute: Zugriff und Modifikation

```
System.out.println(Quader.anzahlKanten) ;  
Quader.anzahlQuader = Quader.anzahlQuader + 1 ;
```

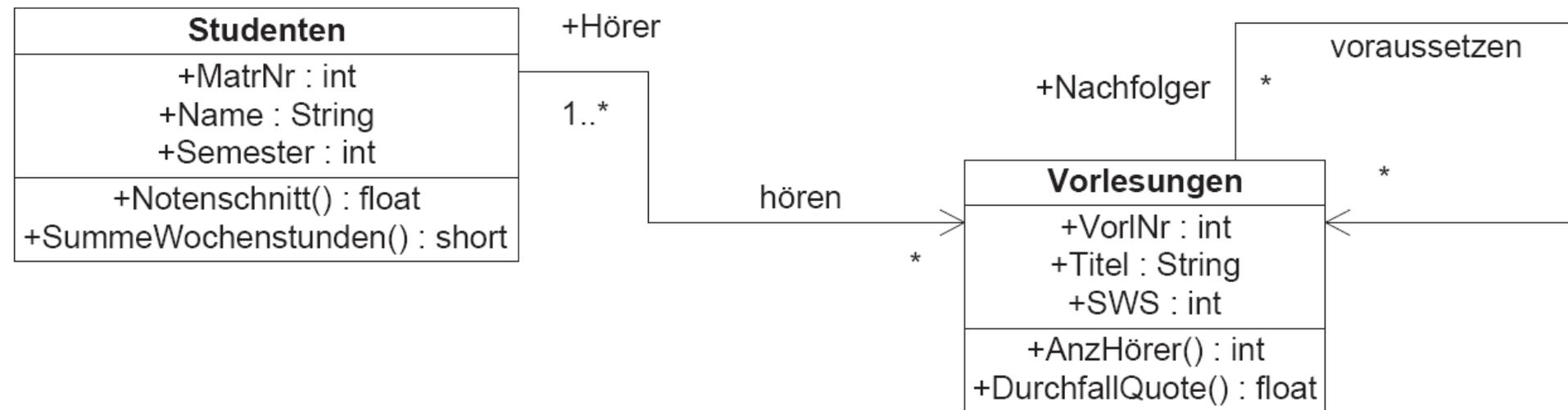
anzahlKanten: 12  
anzahlEcken: 8  
anzahlQuader: 3



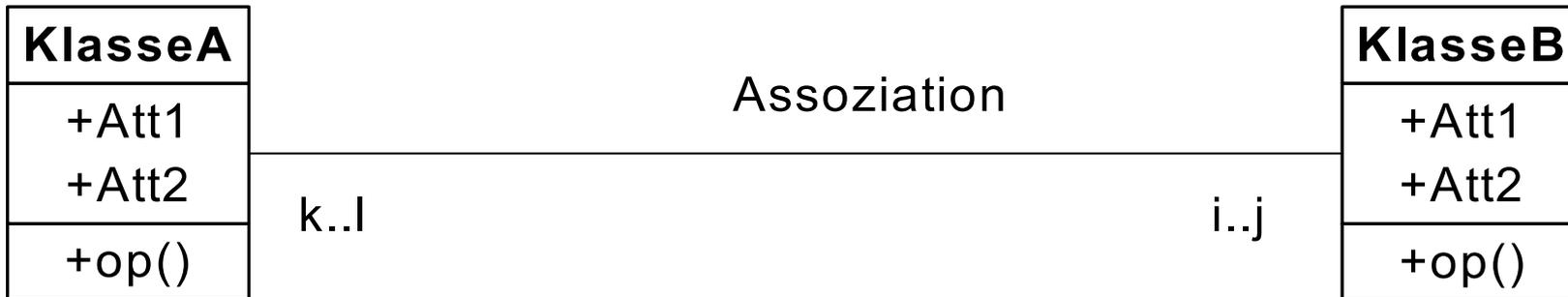
# Systematische Modellierung mit UML und Umsetzung in Java



# Assoziationen



# Multiplizität



- Jedes Element von KlasseA steht mit mindestens i Elementen der KlasseB in Beziehung
- ... und mit maximal j vielen KlasseB-Elementen
- Analoges gilt für das Intervall k..l
- Multiplizitätsangabe ist analog zur Funktionalitätsangabe im ER-Modell
  - **Nicht** zur (min,max)-Angabe: **Vorsicht!**

# Multiplizität/Funktionalität einer Assoziation

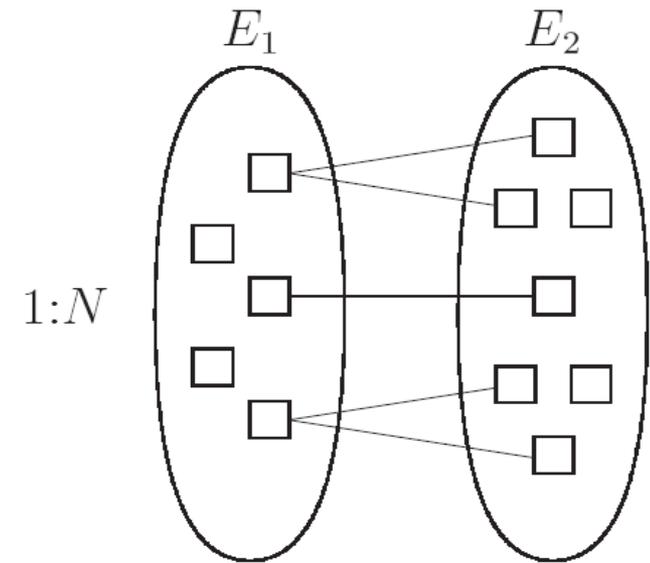
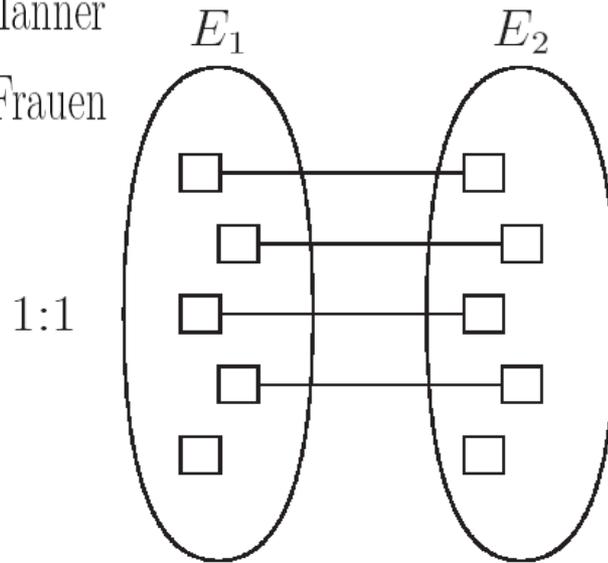


Betrachten wir das abstrakte Beispiel in Abbildung 1.19: Wenn man in UML an einer Seite der binären Assoziation die Multiplizitätsangabe  $i..j$  macht, so bedeutet dies, dass jedes Objekt der Klasse auf der anderen Seite mit mindestens  $i$  und höchstens  $j$  Objekten der Klasse auf dieser Seite in Beziehung stehen muss. Bezogen auf unser Beispiel bedeutet dies, dass jedes Objekt der Klasse  $E_1$  mit mindestens  $i$  und mit maximal  $j$  Objekten der Klasse  $E_2$  in Beziehung stehen muss/darf. Analog muss jedes Objekt der Klasse  $E_2$  mit mindestens  $k$  Objekten der Klasse  $E_1$  in Beziehung stehen und es darf maximal mit  $l$  Objekten der Klasse  $E_1$  in dieser Beziehung stehen. Wenn die minimale und die maximale Anzahl übereinstimmt, also  $m..m$  gilt, so vereinfacht man dies in UML zu einem einzigen Wert  $m$ . Dies gilt auch für  $*..*$  was immer als  $*$  angegeben wird.

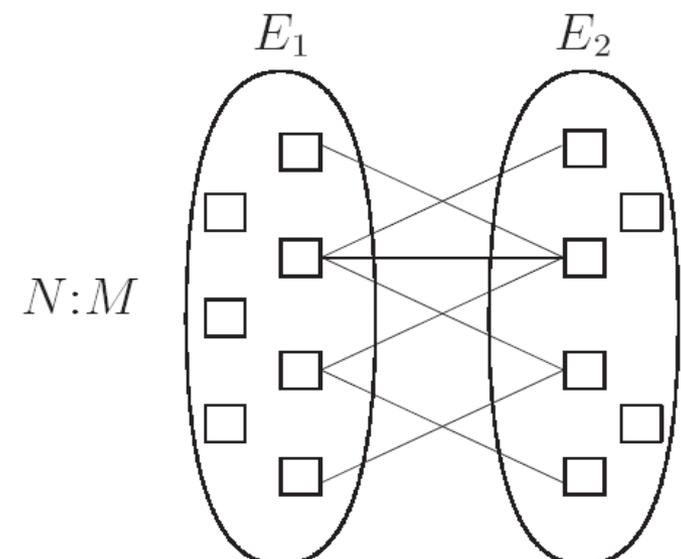
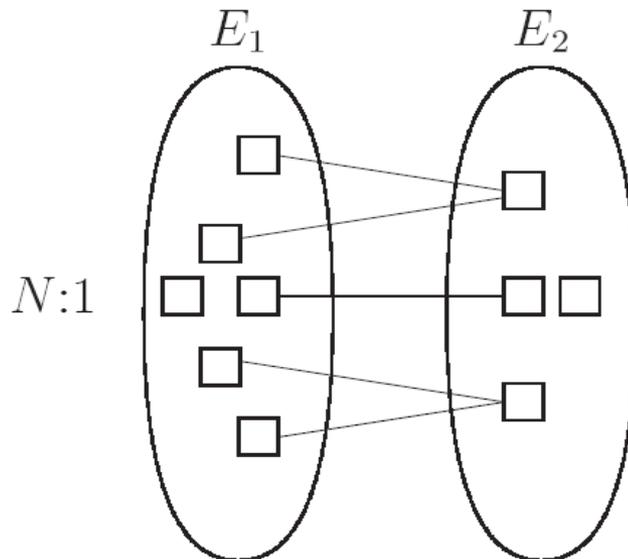
# Funktionalitäten

Ehemann : Frauen  $\rightarrow$  Männer

Ehefrau : Männer  $\rightarrow$  Frauen



angestelltBei : Personen  $\rightarrow$  Firmen



- *Eins-zu-eins* bzw. *1:1-Beziehung*, falls jedem Objekt  $e_1$  aus  $E_1$  höchstens ein Objekt  $e_2$  aus  $E_2$  zugeordnet ist und umgekehrt jedem Objekt  $e_2$  aus  $E_2$  ebenfalls maximal ein Objekt  $e_1$  aus  $E_1$ . Man beachte, dass es auch Objekte aus  $E_1$  (bzw.  $E_2$ ) geben kann, denen kein „Partner“ aus  $E_2$  (bzw.  $E_1$ ) zugeordnet ist.

In der UML-Notation aus Abbildung 1.19 ist eine Eins-zu-eins-Assoziation dadurch gekennzeichnet, dass  $l = j = 1$  gilt.

Ein Beispiel einer „realen“ 1:1-Beziehung ist *verheiratet* zwischen den Objekttypen *Männer* und *Frauen* – zumindest nach europäischem Recht.

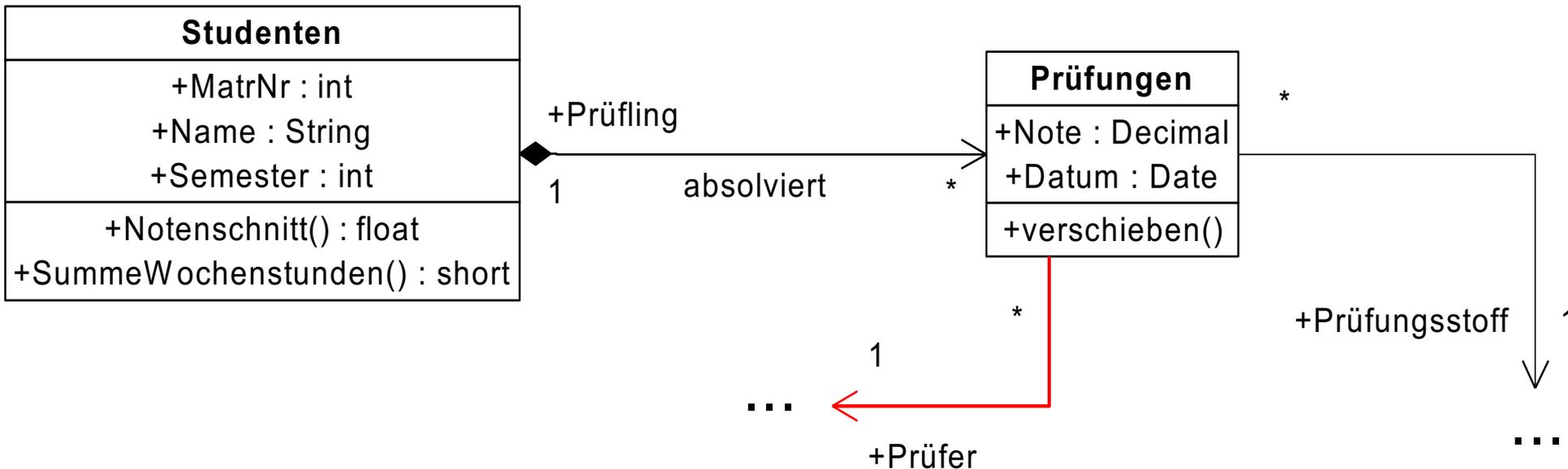
- *Eins-zu-viele* bzw. *1:N-Beziehung*, falls jedem Objekt  $e_1$  aus  $E_1$  beliebig viele (also mehrere oder auch gar keine) Objekte aus  $E_2$  zugeordnet sein können, aber jedes Objekt  $e_2$  aus der Menge  $E_2$  mit maximal einem Objekt aus  $E_1$  in Beziehung steht.

In der UML-Notation aus Abbildung 1.19 ist eine Eins-zu-viele-Assoziation dadurch gekennzeichnet, dass  $l = 1$  und  $j > 1$  gilt. Insbesondere kann  $j = *$  gelten.

Ein anschauliches Beispiel für eine 1:N-Beziehung ist *angestelltBei* zwischen *Personen* und *Firmen*, wenn wir davon ausgehen, dass eine Firma i.a. mehrere Personen beschäftigt, aber eine Person nur bei einer (oder gar keiner) Firma angestellt ist.

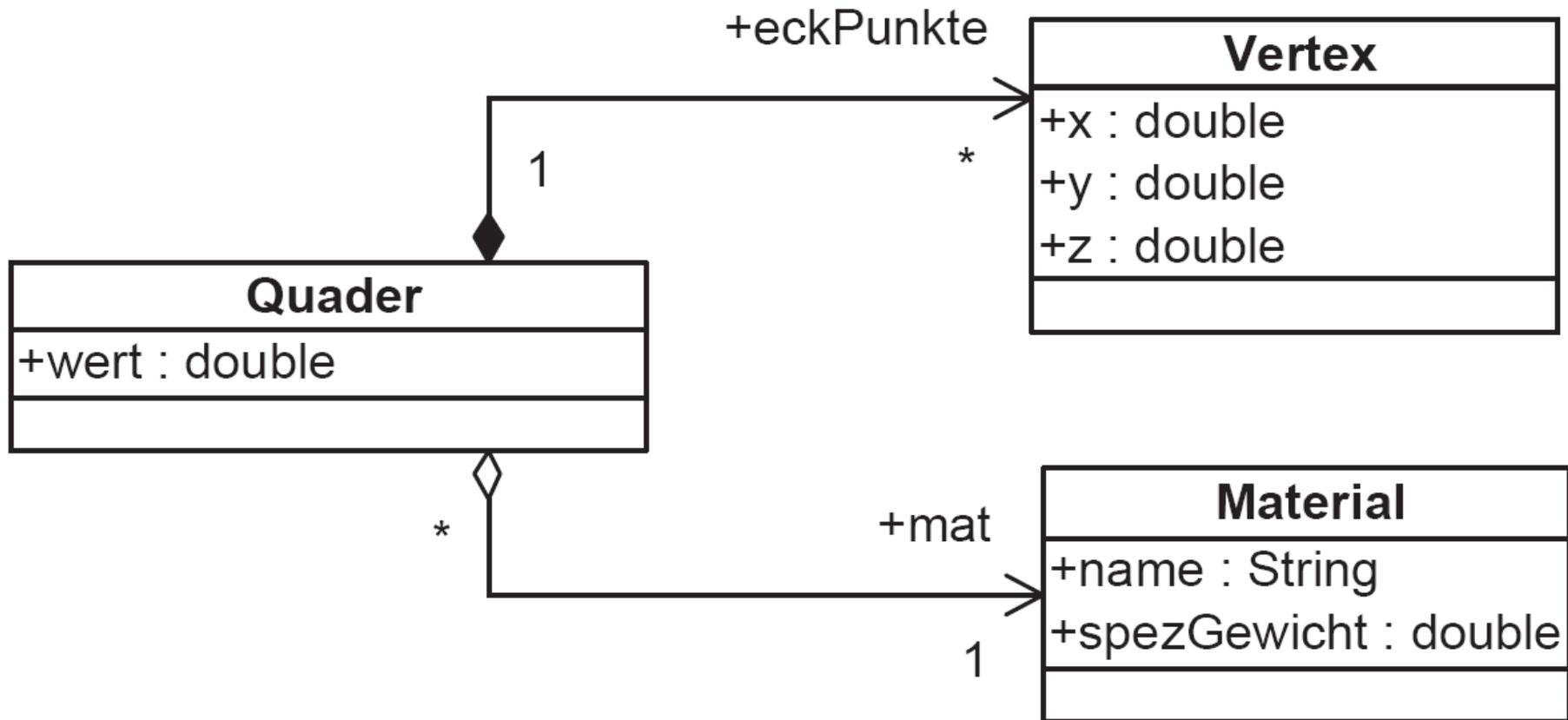
- *N:1-Beziehung*, falls analoges zu obigem gilt.
- *N:M-Beziehung*, wenn keinerlei Restriktionen gelten müssen, d.h. jedes Objekt aus  $E_1$  mit mehreren Objekten aus  $E_2$  in Beziehung stehen kann und umgekehrt jedes Objekt aus  $E_2$  mit mehreren Objekten aus  $E_1$  assoziiert werden darf.

# Aggregation/Komposition

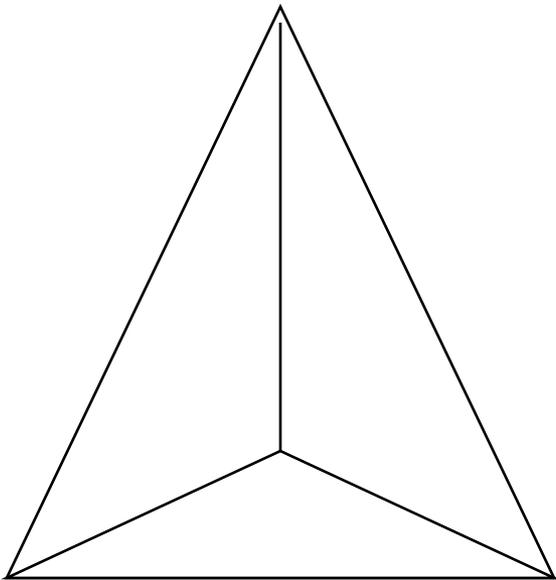
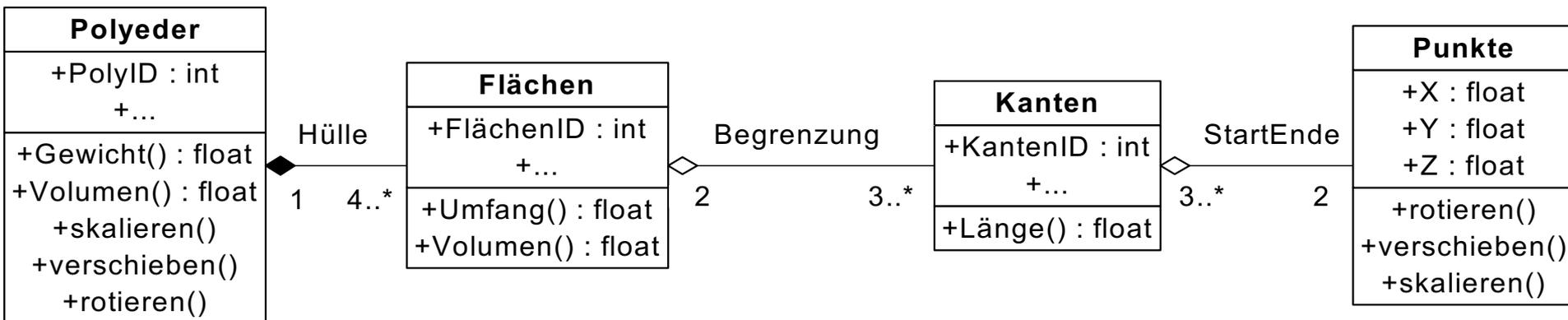


- Komposition (ausgefüllter Diamant)
  - Exklusive Zuordnung
  - Existenzabhängig
- Aggregation („leerer“ Diamant)
  - Nicht-exklusive
  - Nicht-existenzabhängige Teil/Ganzes-Beziehung

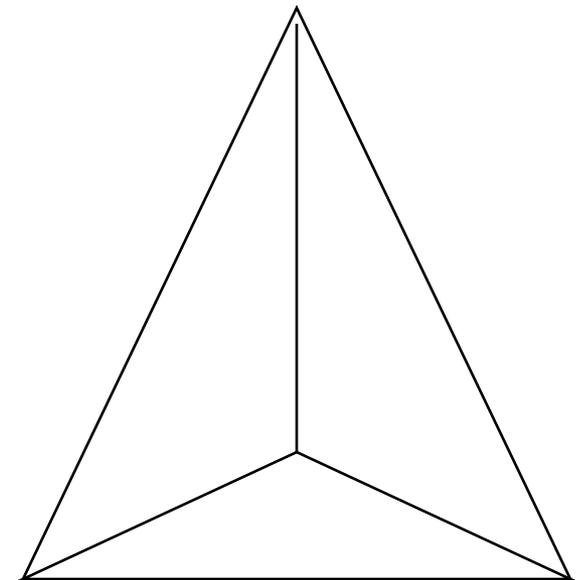
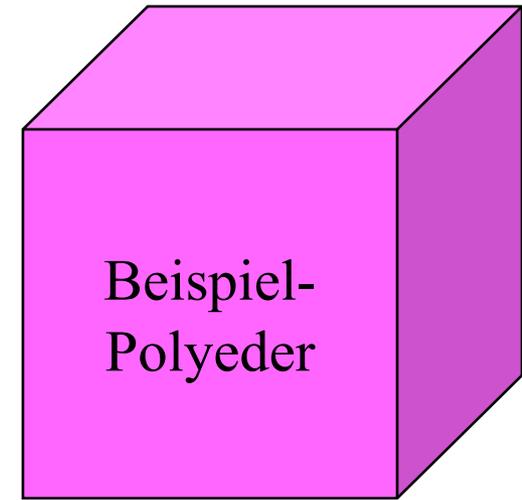
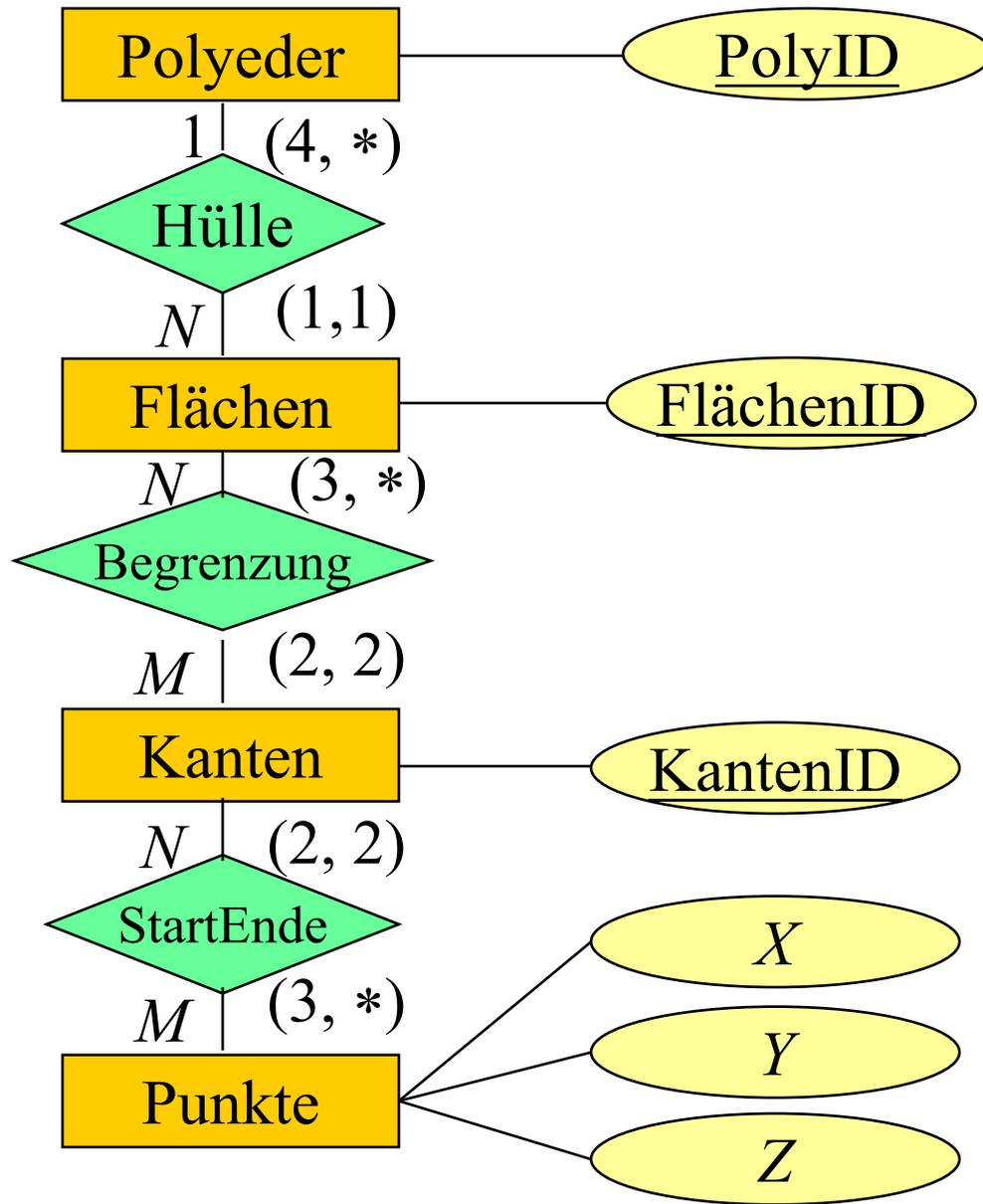
# Aggregation/Komposition



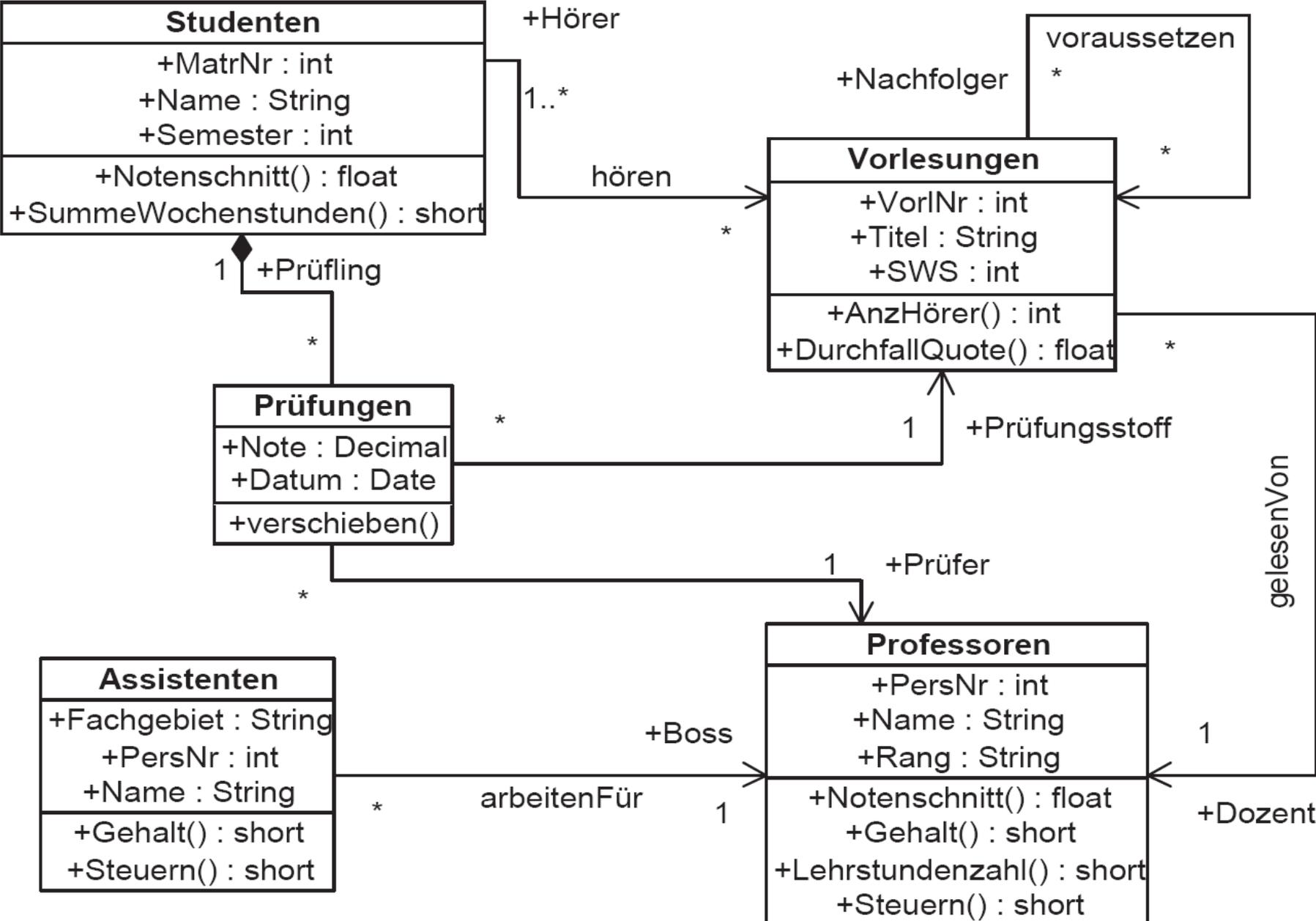
# Begrenzungsflächenmodellierung von Polyedern in UML

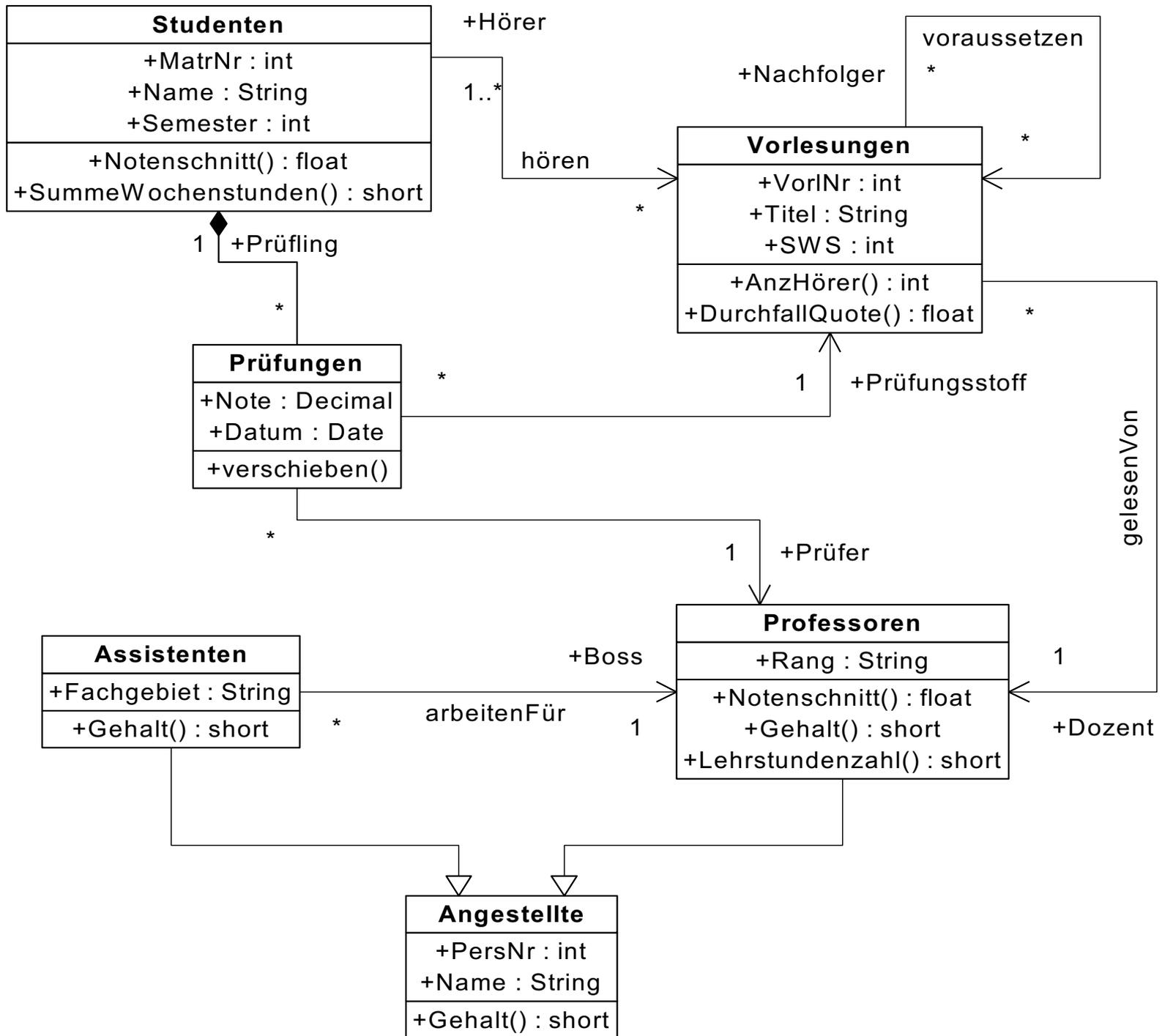


# Begrenzungsflächendarstellung



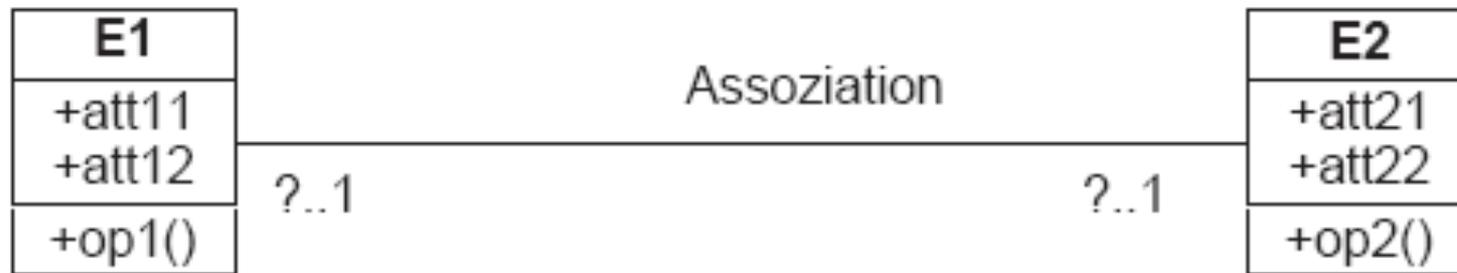
# Universitäts-Modell





# Umsetzung in Java

## 1:1-Assoziation

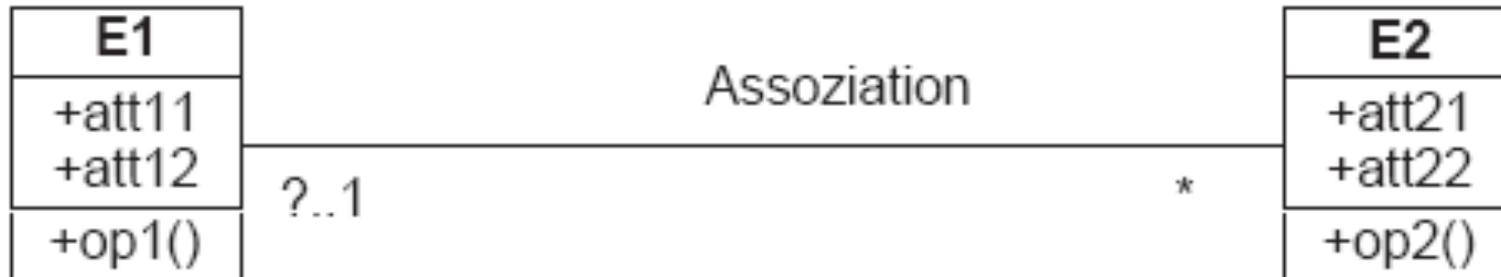


```
class E1 {
    public ... att11;
    public ... att12;
    public E2 zugeordnetesE2;
    public ... op1() {}
}
```

```
class E2 {
    public ... att21;
    public ... att22;
    public E1 zugeordnetesE1;
    public ... op2() {}
}
```

# Umsetzung in Java

## 1:N-Assoziation



```
class E1 {  
    public ... att11;  
    public ... att12;  
    public E2[] zugeordneteE2;  
    public ... op1() {}  
}
```

```
class E2 {  
    public ... att21;  
    public ... att22;  
    public E1 zugeordnetesE1;  
    public ... op2() {}  
}
```

```
class Quader2 {  
    public Vertex[] eckPunkte;  
    public Material mat;  
    public double wert;  
}
```

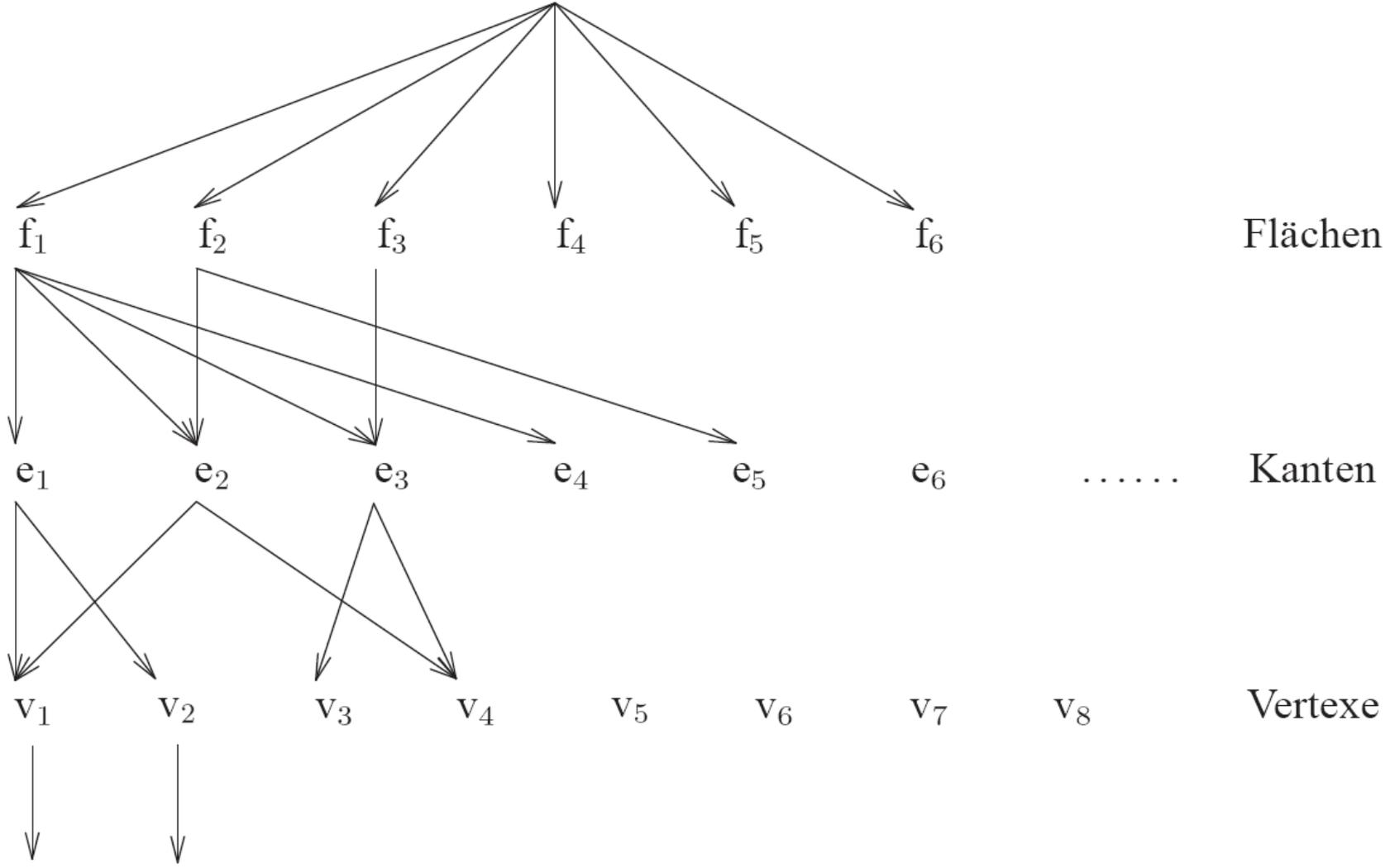
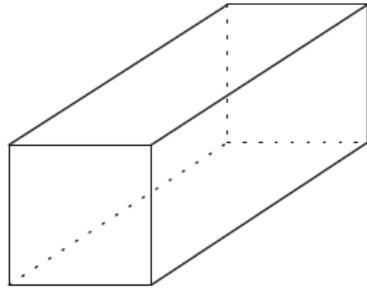
# Umsetzung einer Assoziation in Java viele-viele (N:M)



```
class E1 {
    public ... att11;
    public ... att12;
    public E2[] zugeordneteE2;
    public ... op1() {}
}

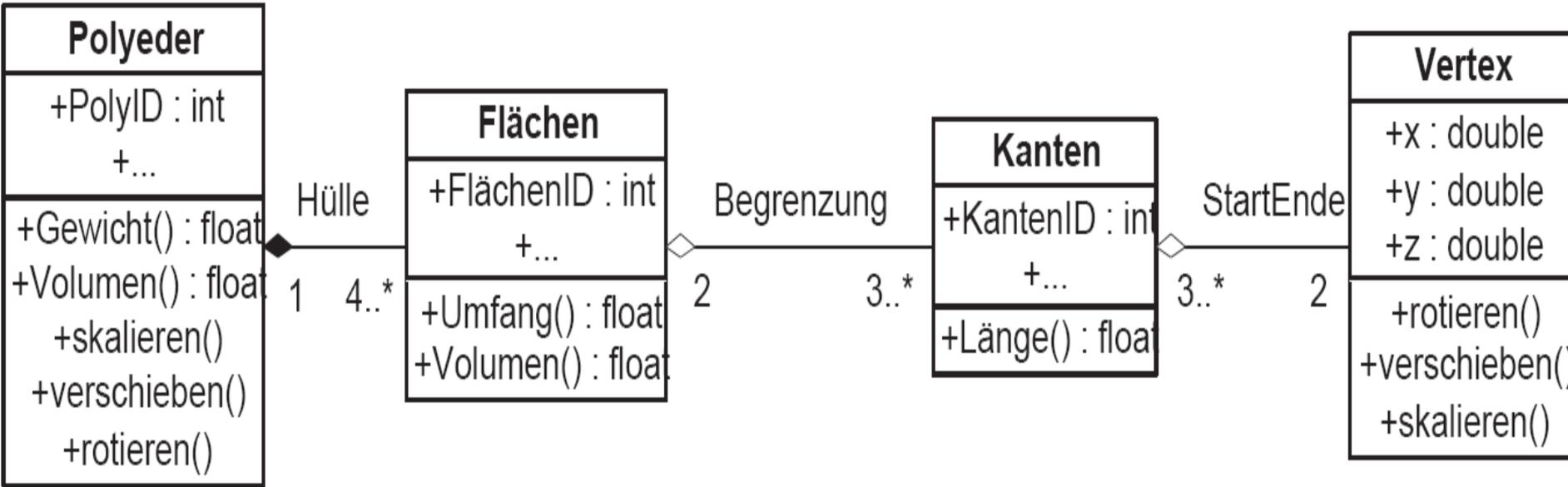
class E2 {
    public ... att21;
    public ... att22;
    public E1[] zugeordneteE1;
    public ... op2() {}
}
```

# Begrenzungs- Flächen-Modell

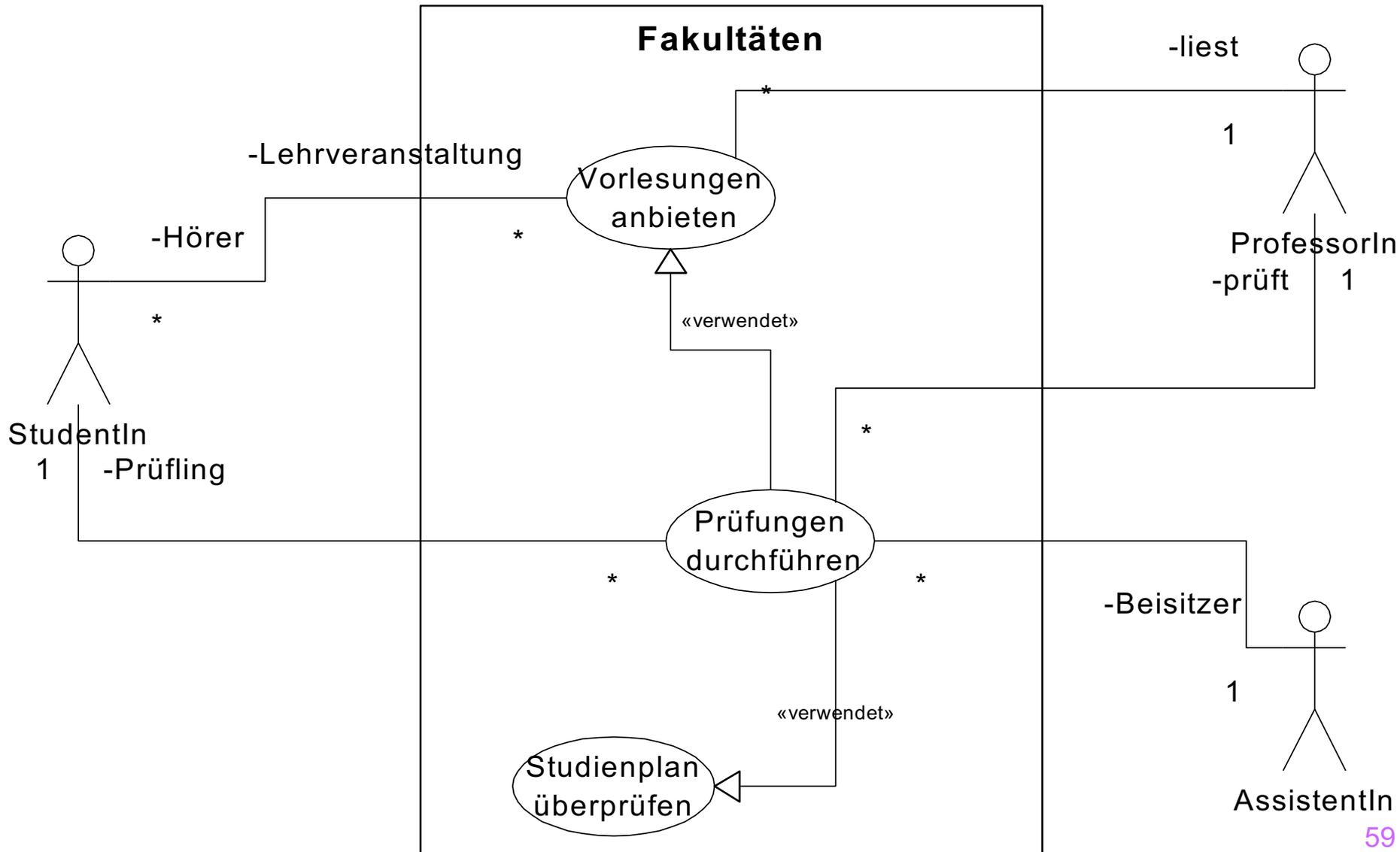


Koordinaten

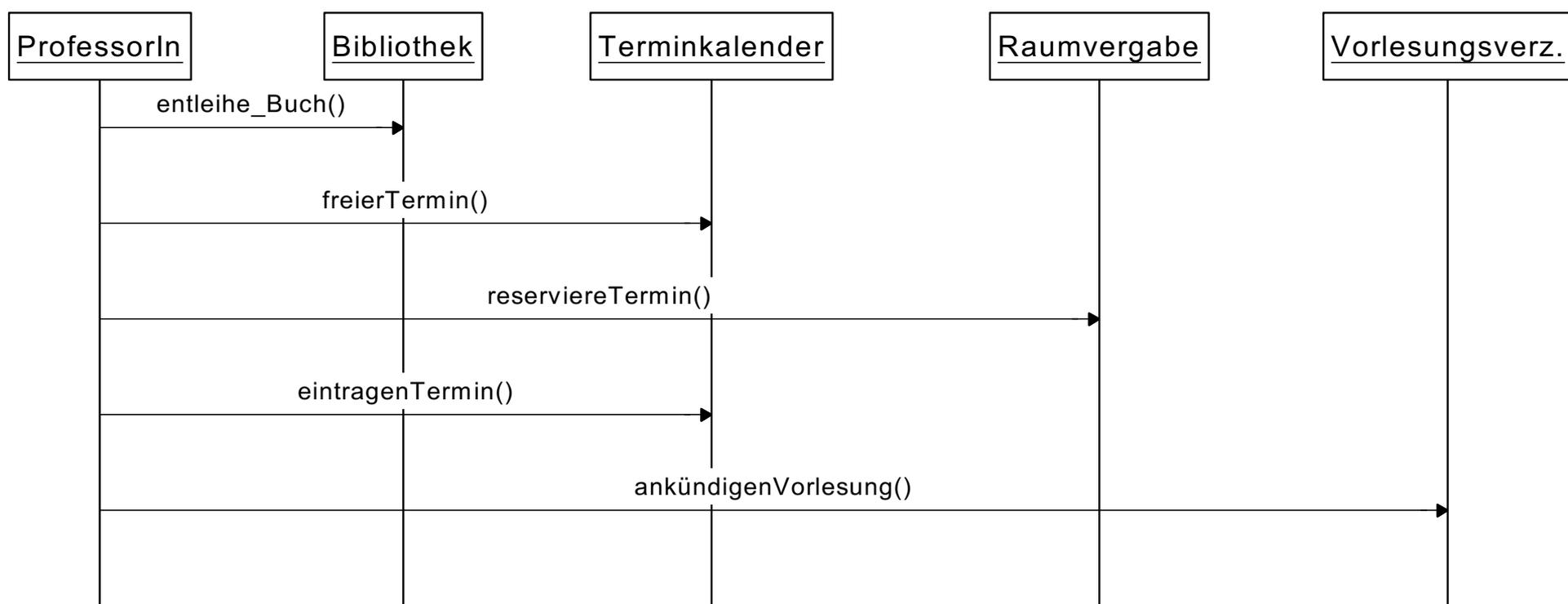
# Polyeder in UML



# Anwendungsfälle (use cases)



# Interaktions-Diagramm: Modellierung komplexer Anwendungen



# Interaktions-Diagramm: *Prüfungsdurchführung*

