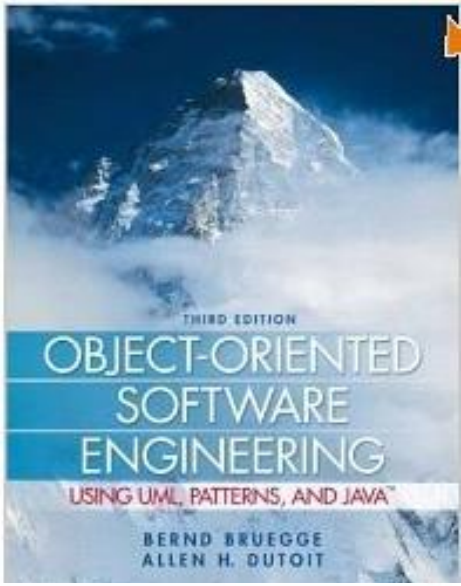
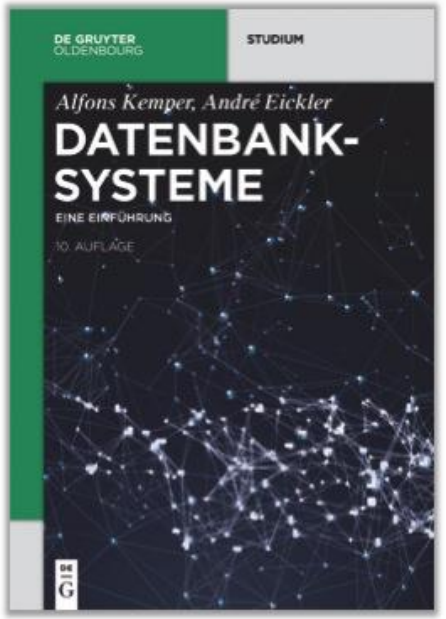
Einführung in die Informatik II für Ingenieurwissenschaften (MSE)

- Prof. Alfons Kemper, Ph.D.
- Simon Ellmann
- Christoph Anneser

- 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:
  - Simon Ellmann (ellmann@in.tum.de)
  - Christoph Anneser (anneser@in.tum.de)

- Tutor:
  - Sebastian Reichbauer

Organisatorisches I

- Vorlesungs-Website: https://db.in.tum.de/teaching/ss24/ei2
- Moodle-Kurs: https://www.moodle.tum.de/course/view.php?id=96067

- Vorlesung:
  - Montags 10:00 – 11:15
  - Raum: BC2 0.01.17 (Garching-Hochbrück)
  - Aufzeichnungen des SS20 auf der Vorlesungs-Website

- Zentralübung:
  - Montags 11:30 – 12:30
  - Raum: BC2 0.01.17 (Garching-Hochbrück)
  - Aufzeichnungen des SS21 auf Panopto (need to sign in first!)

Organisatorisches II

- Fragestunde:
  - Donnerstags 15:30 – 16:30
  - Raum: BC2 0.01.17 (Garching-Hochbrück)

- Alle Materialen sind auf der Vorlesungs-Website zu finden

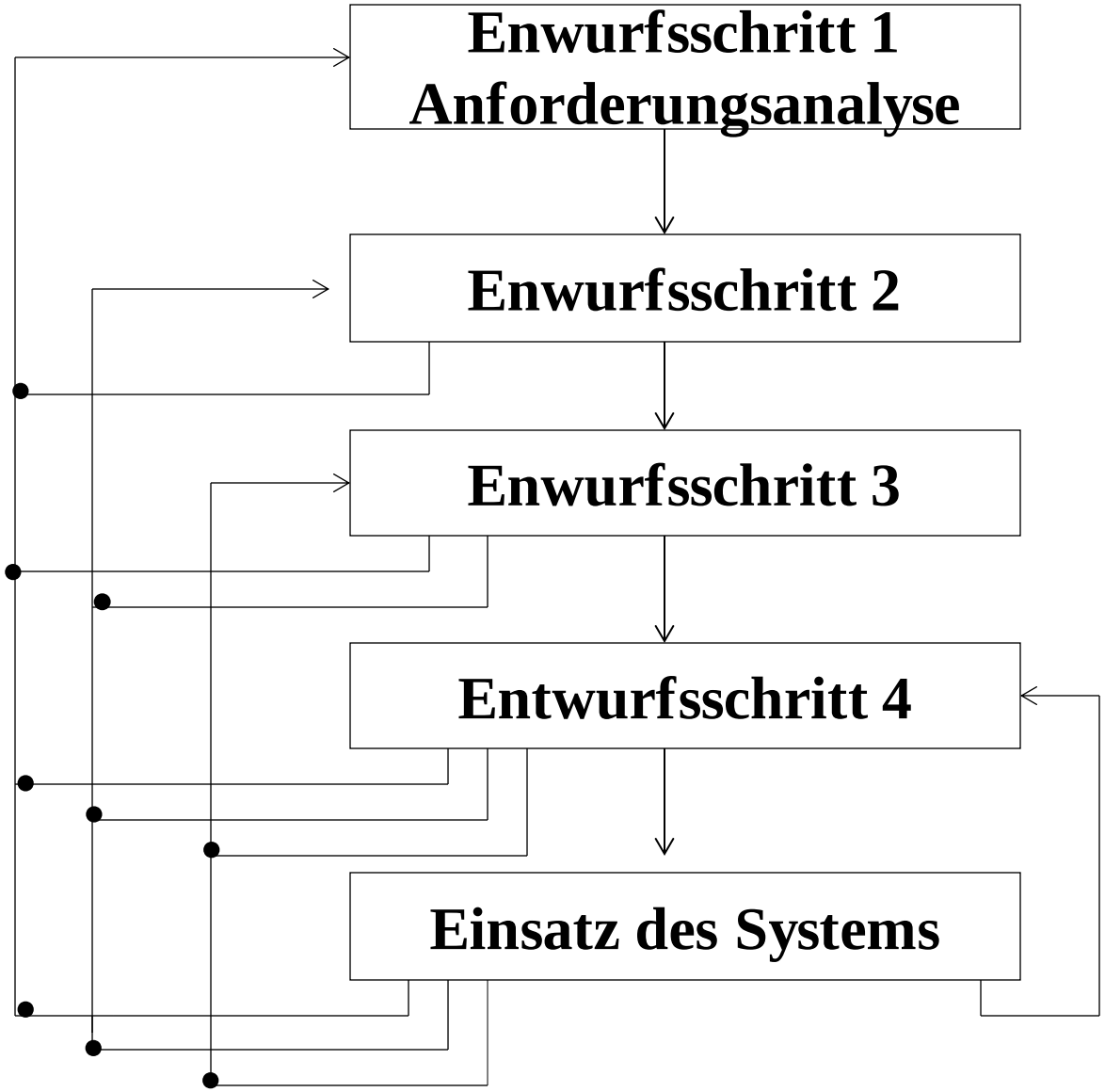
- Prüfungstermin:
  - Voraussichtlich Anfang September (ohne Gewähr)

Datenbankentwurf

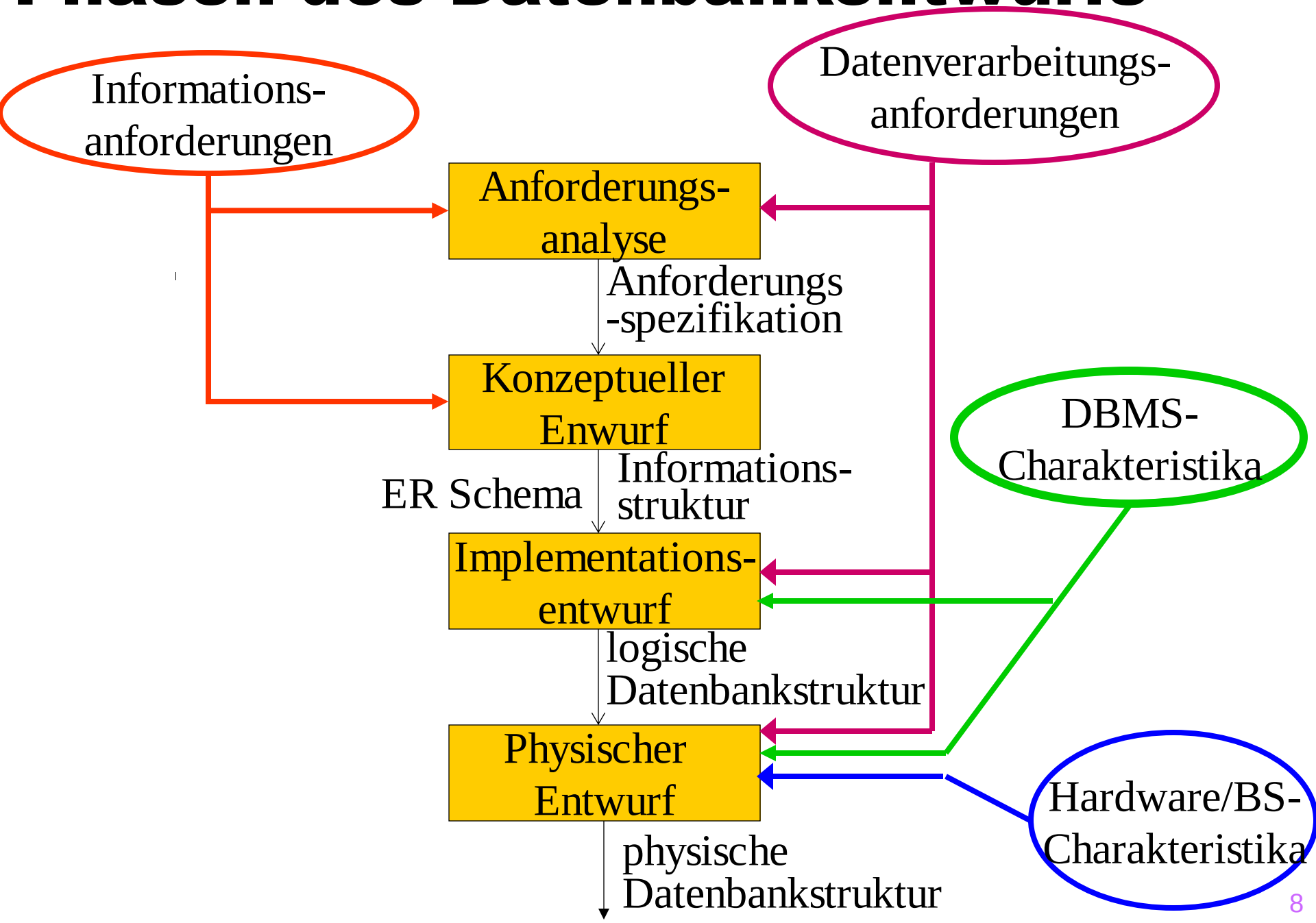
Abstraktionsebenen des Datenbankentwurfs

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

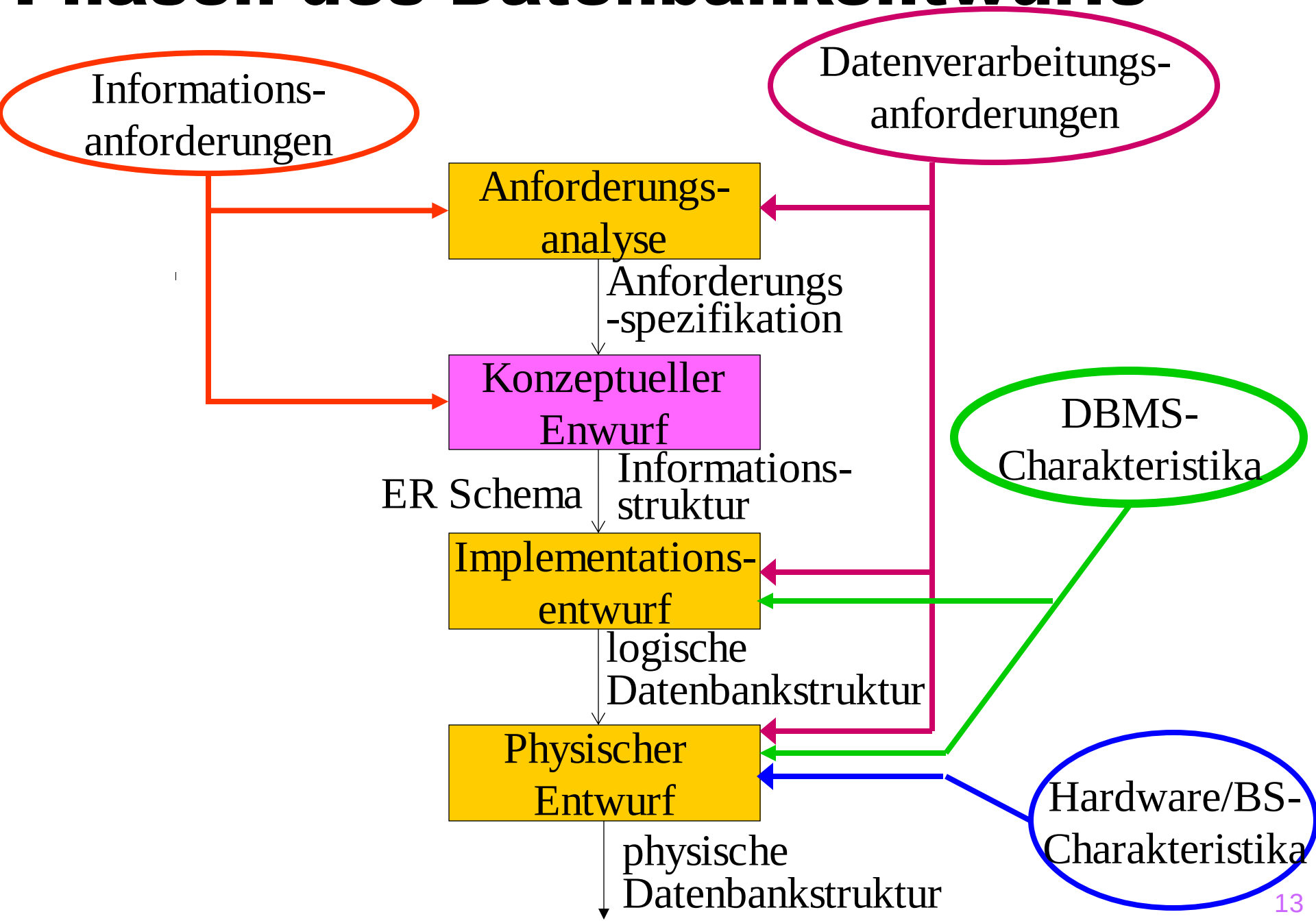
Beziehungsbeschreibung: *prüfen*

- 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



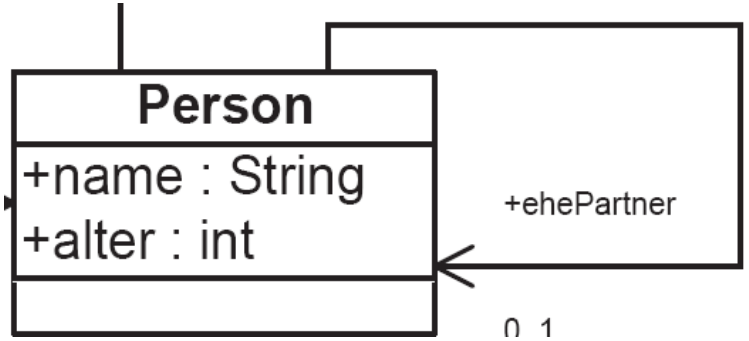
Datenmodellierung mit UML

- 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 {
        Typ_1 Attr_1 ;
        ...;
        Typ_n Attr_n ;
    // Operationen folgen hier
        ...
} // end class TypName;
```




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

Werte versus Objekte

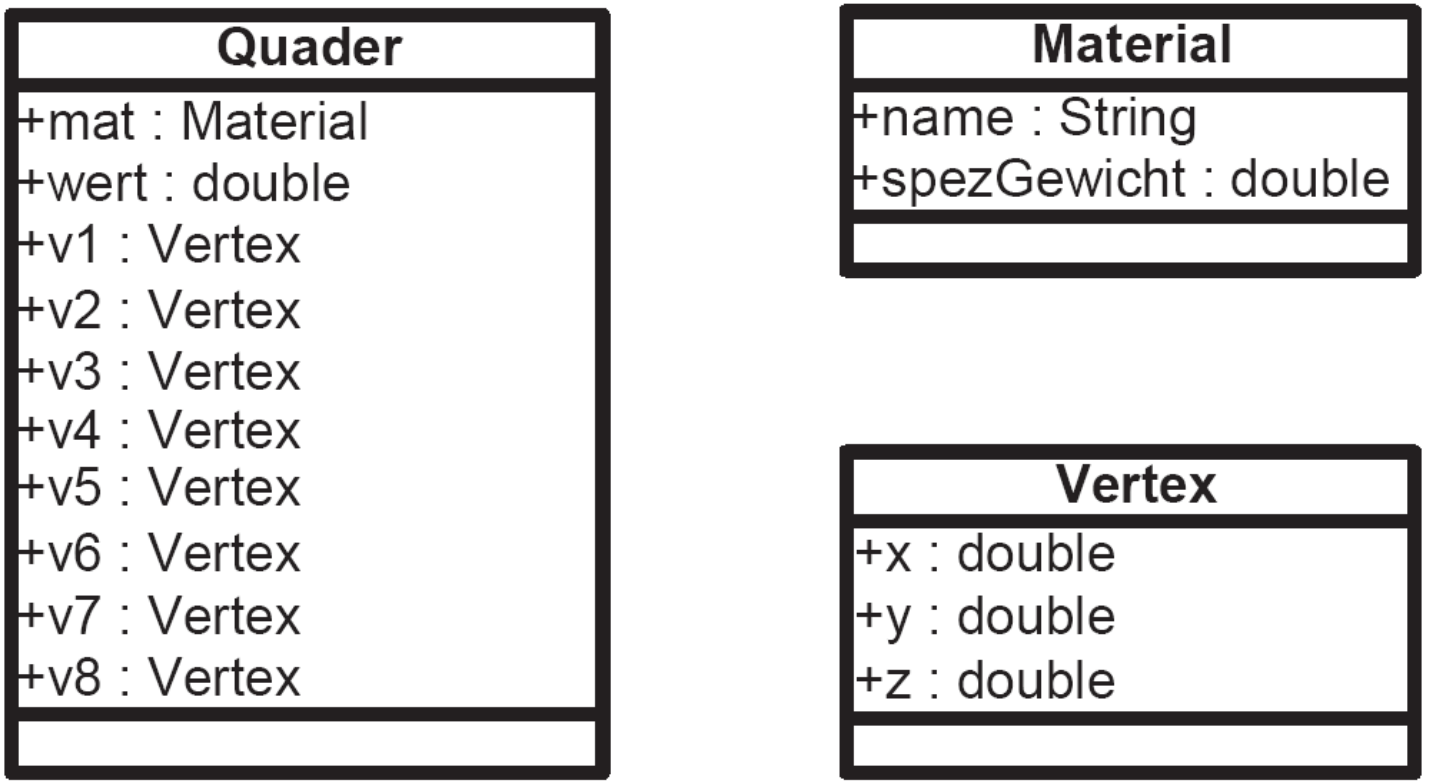
Typen		Instanzen
primitive Typen	$\longrightarrow$	Werte
Objekttypen (Klassen)	$\longrightarrow$	Objekte

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

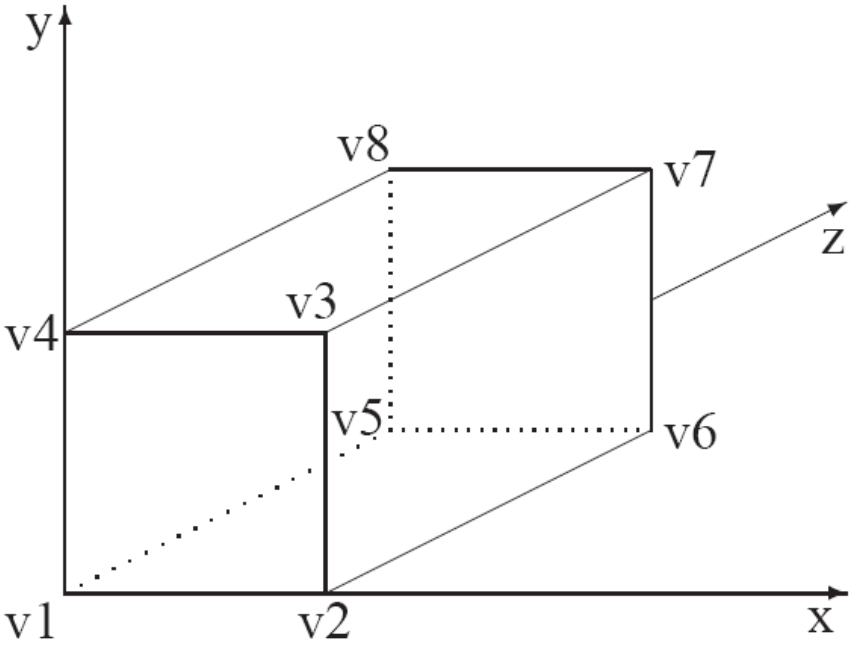
Java Klassendefinition: Syntax

```
[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



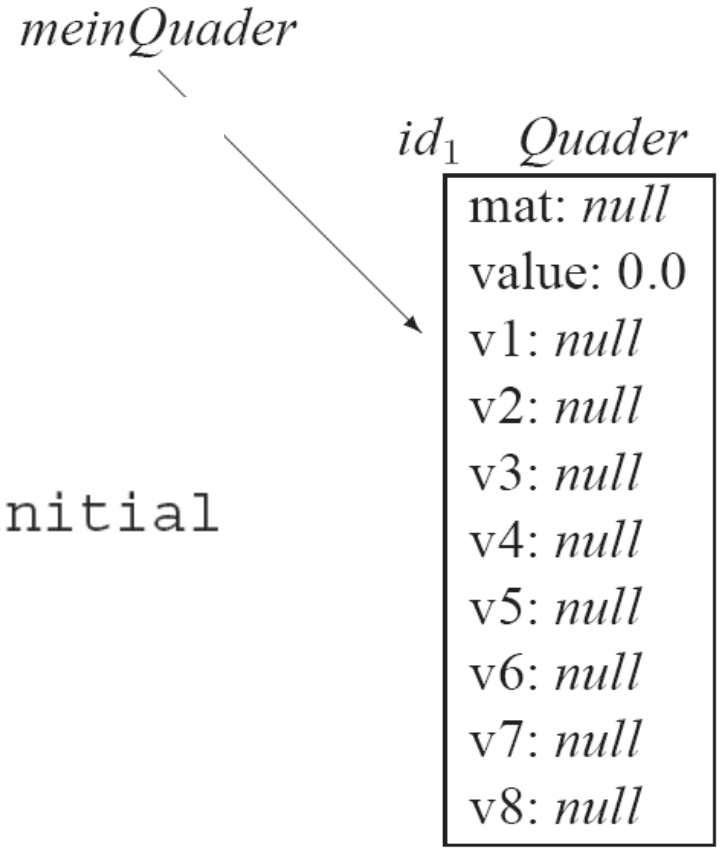

```
class Vertex {
    public double x;
    public double y;
    public double z;
}

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

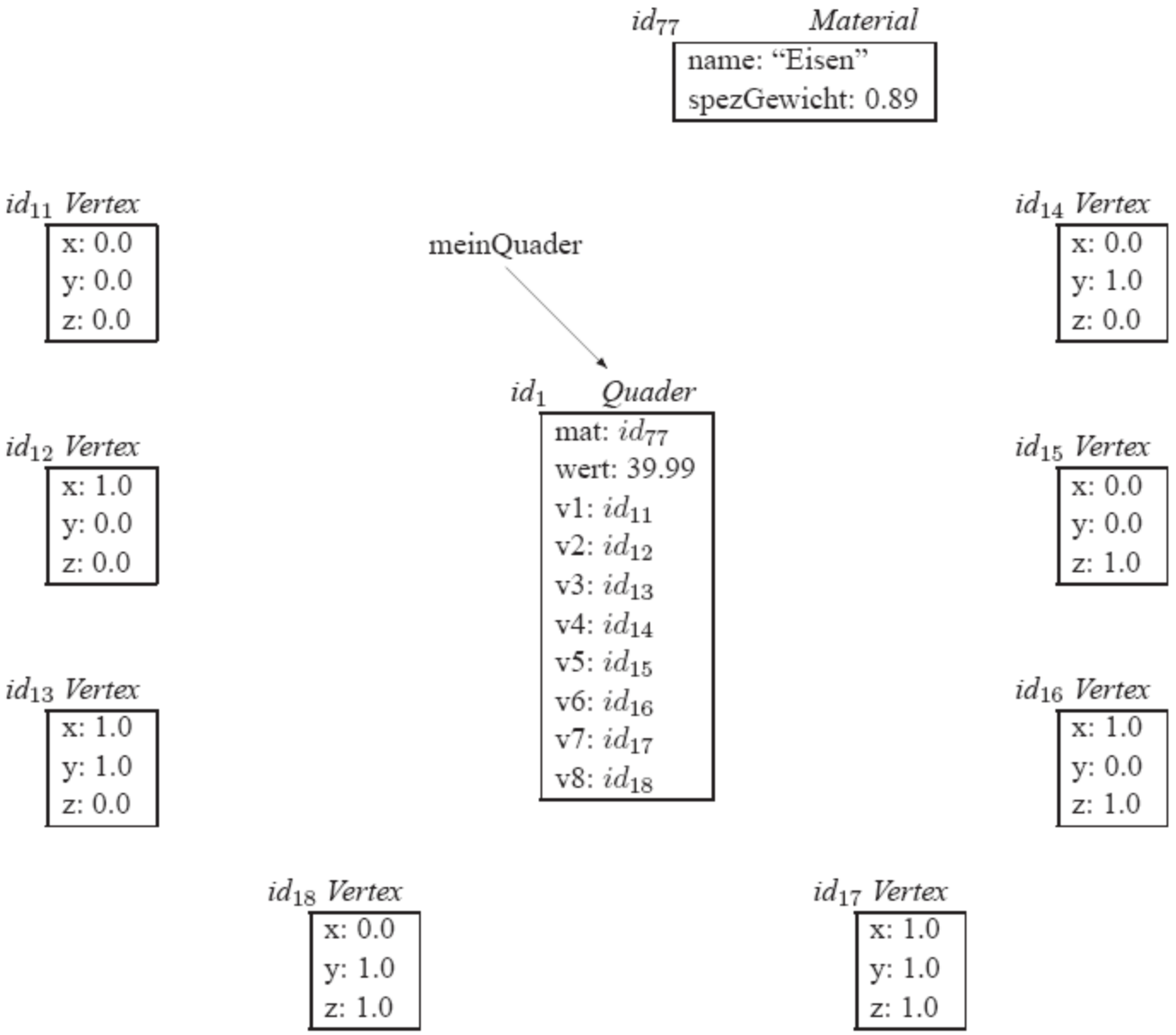
- *int, short, byte, long*-Attribute werden auf den Wert 0 initialisiert.
- *double, float*-Attribute werden initial auf den Wert 0.0 gesetzt.
- *char*-Instanzvariablen werden auf den Wert „$\backslash u0000$“ initialisiert.
- *boolean*-Attribute werden auf *false* gesetzt.
- Attribute, die auf einen 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;
```



Resultierendes Objektnetz



Objekt besteht aus (OID, Typ, Rep)

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

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

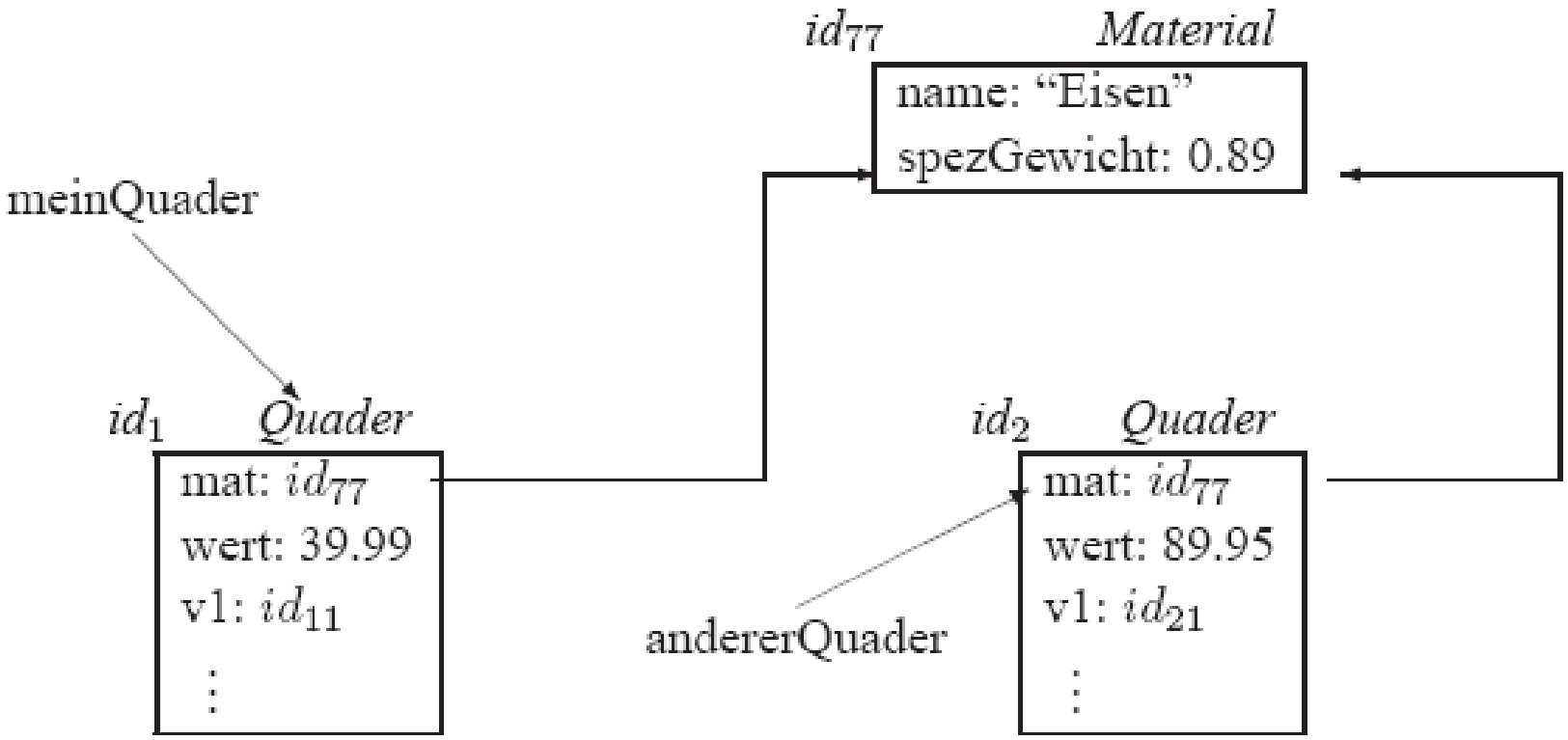
Die drei Teile haben die folgende Bedeutung:

- $id_{\#}$ 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 Repräsentation) 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

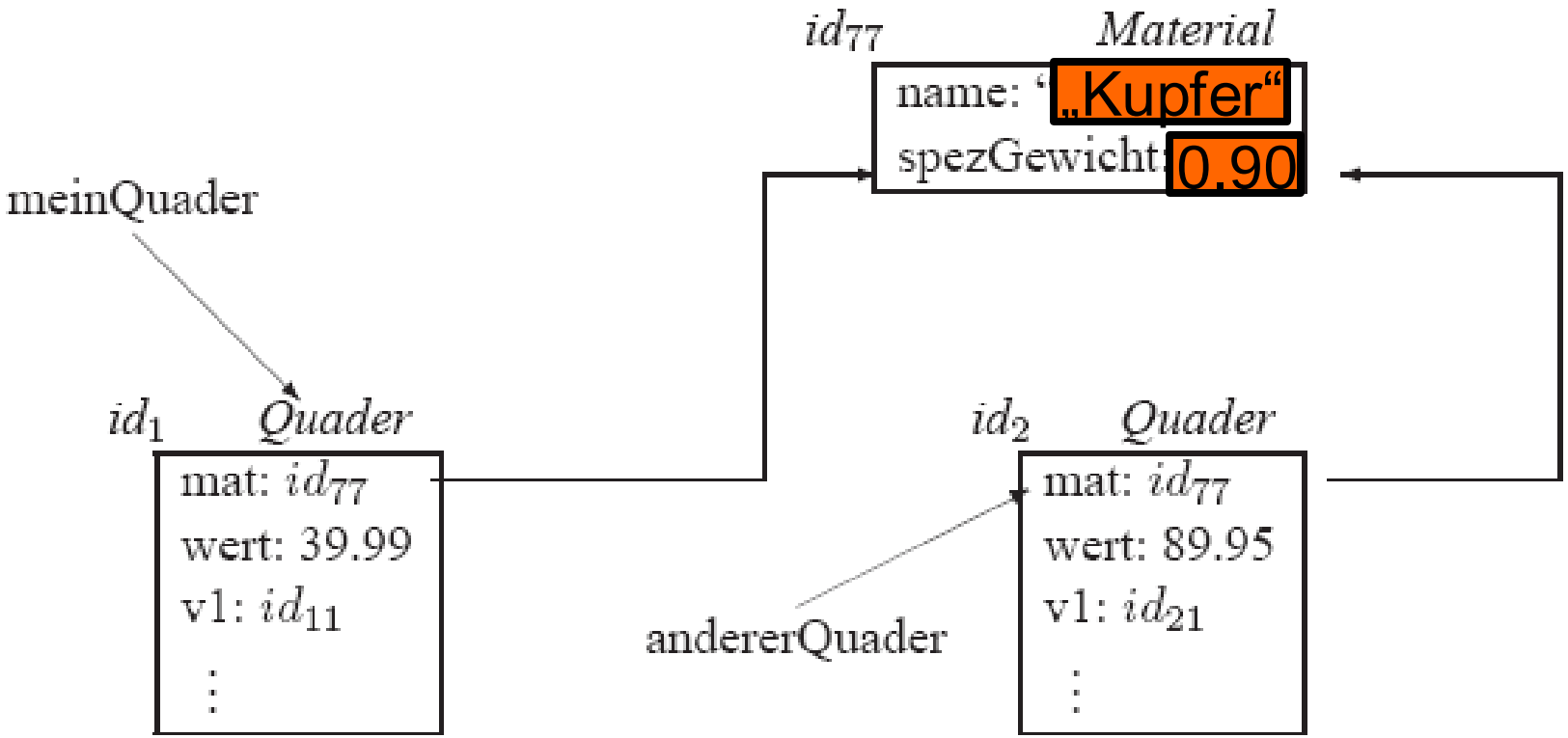
Shared Subobjects/Gemeinsame Unterobiekte

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



Shared Subobjects/Gemeinsame Unterobjekte

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

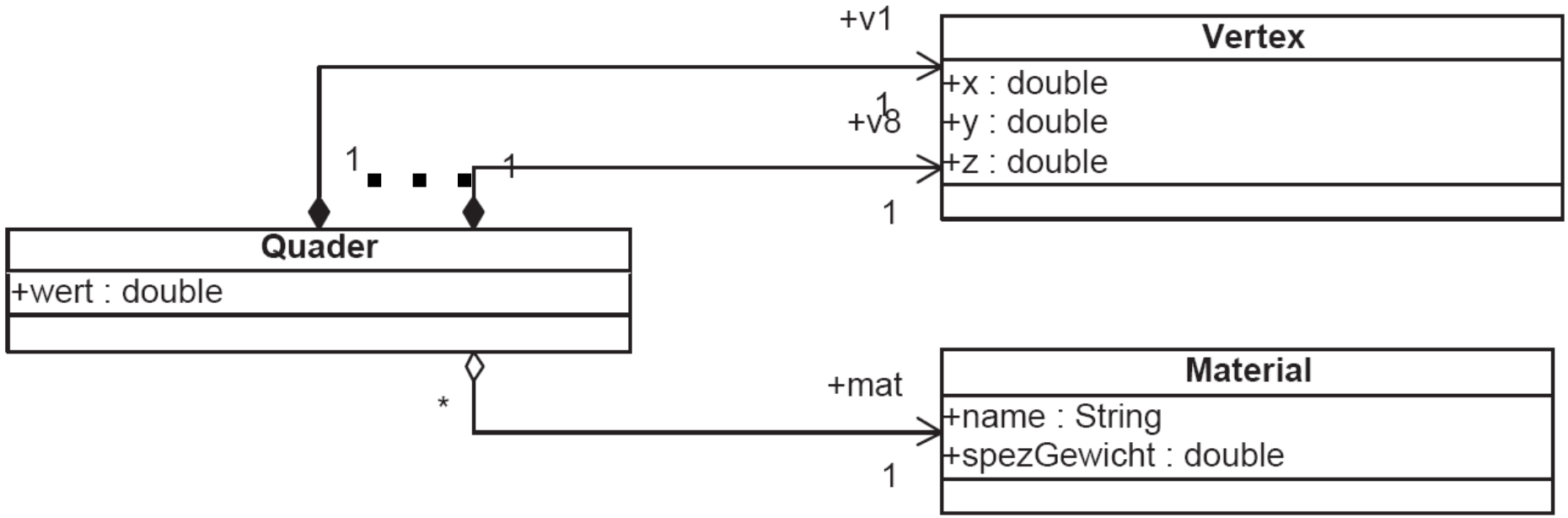


Wertvergleich versus Objektvergleich

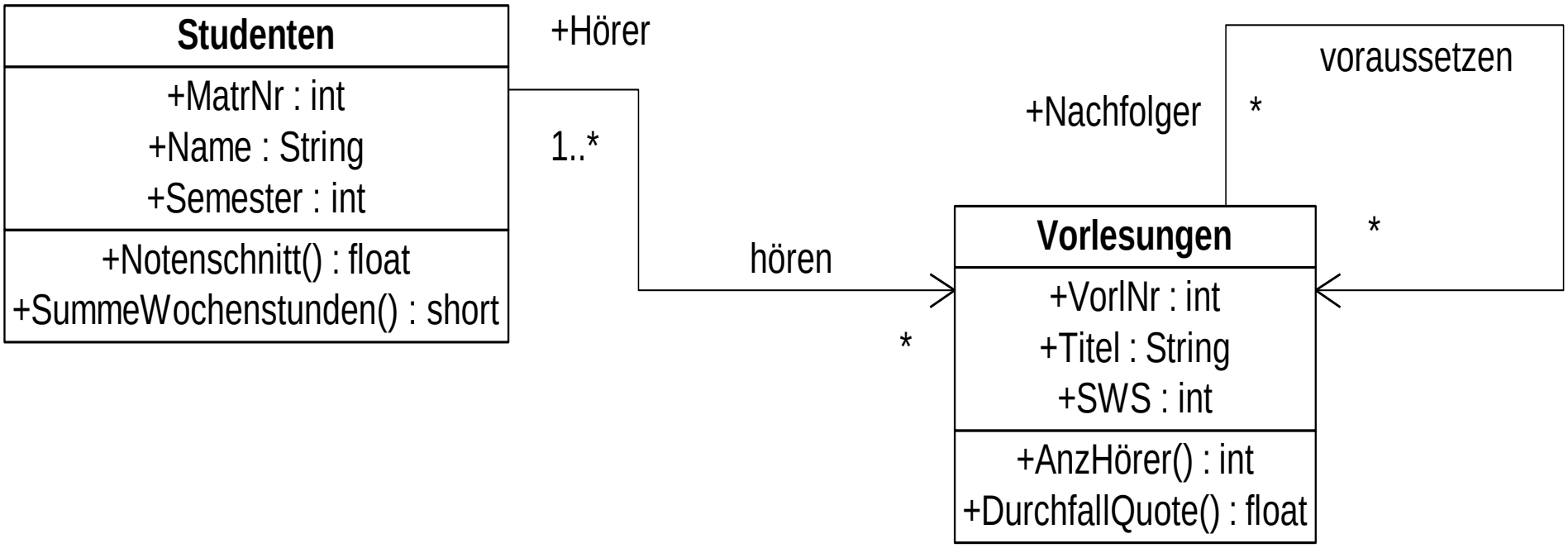
```
andererQuader.mat.name.equals("Kupfer");    // Objekt-Vergleich
andererQuader.mat.spezGewicht == 0.90;      // Wert-Vergleich
```

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

Beziehungen/Assoziationen in UML

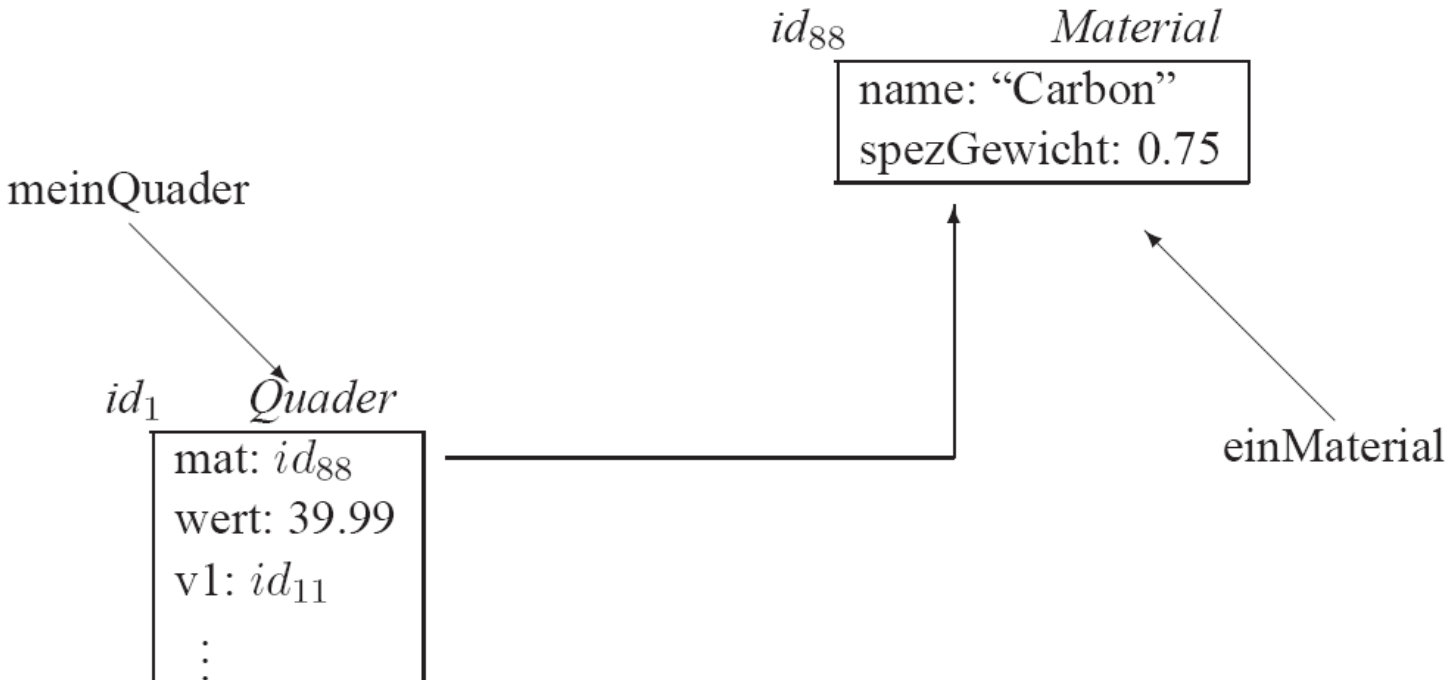


Klassen und Assoziationen



Referenzierung/Dereferenzierung

```
   Quader meinQuader = new Quader();
   Material einMaterial;
   double w;
   ...
(1)einMaterial = new Material();     // einMaterial speichert die OID id88
(2)einMaterial.name = "Carbon";
(3)einMaterial.spezGewicht = 0.75;
(4)meinQuader.mat = einMaterial;     // meinQuader hat die OID id1
(5)w = meinQuader.mat.spezGewicht;
```



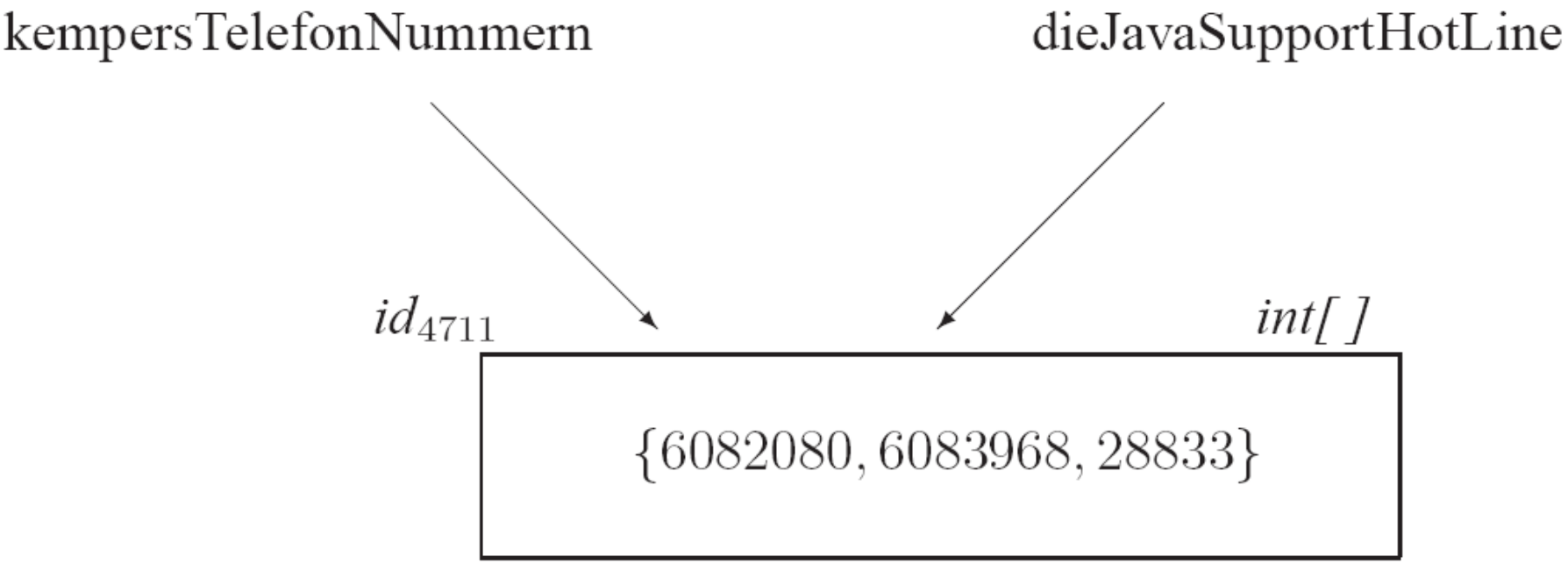
Kollektionen mit Arrays

```
ElementTyp[] arrayName = new ElementTyp[25];
ElementTyp einElem;
arrayName[0] = einElem;
```

```
int[] kempersTelefonNummern = {6082080,0,28833};
   // äquivalent zu:
   // kempersTelefonNummern = new int[3];
   // kempersTelefonNummern[0] = 6082080;
   // kempersTelefonNummern[1] = 0;
   // kempersTelefonNummern[2] = 28833;

int privNummer = kempersTelefonNummern[0];  // pr
                                  // privNummer == 6082080
```

Kollektion als shared subobject

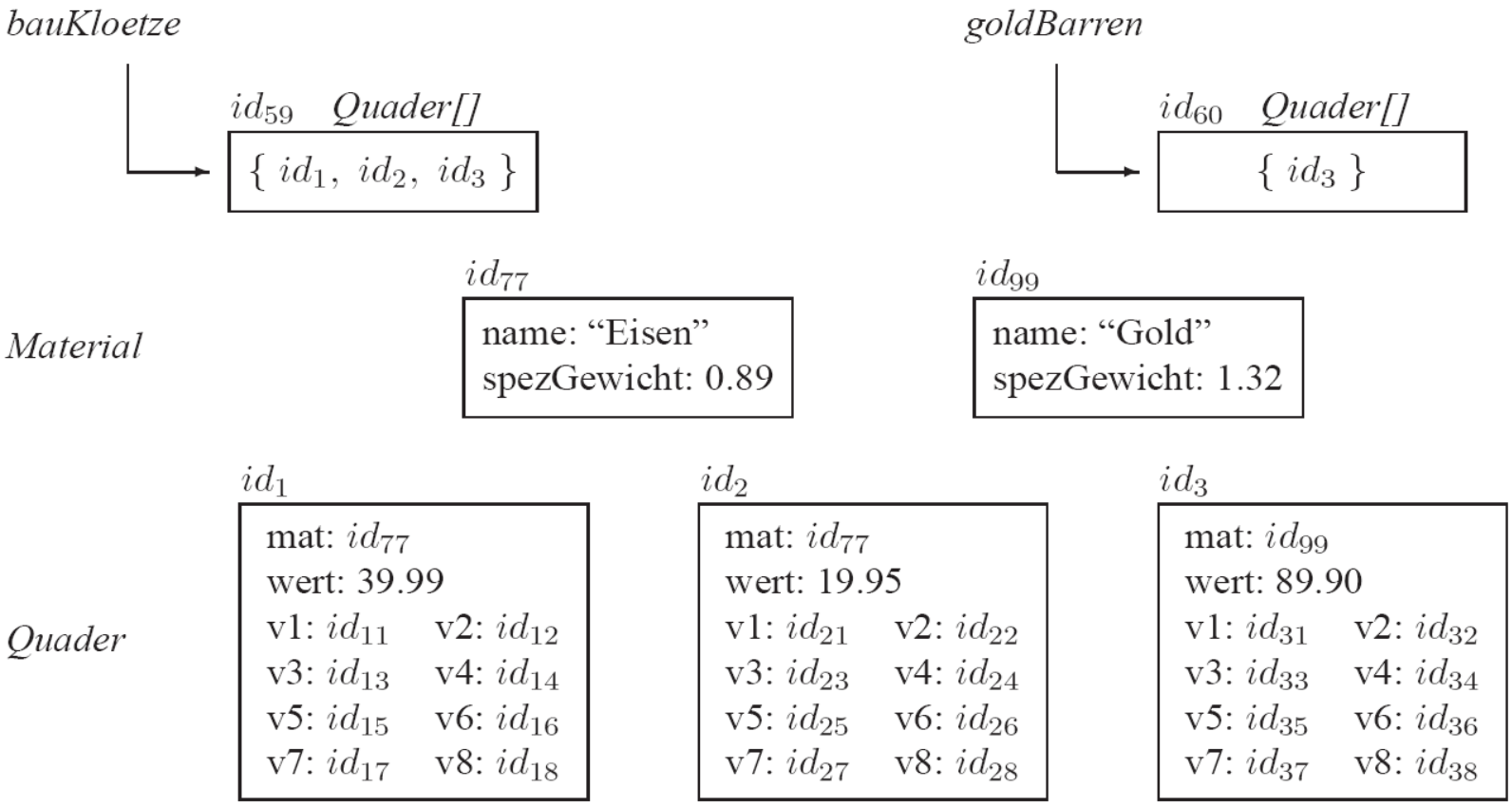
```
int[] dieJavaSupportHotLine;
dieJavaSupportHotLine = kempersTelefonNummern;
```




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

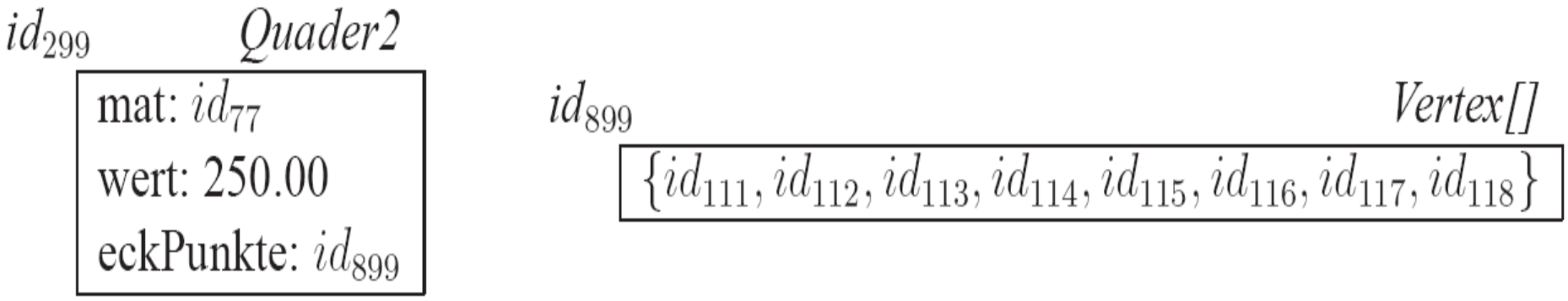
Kollektionen sind „first class citizens“

```
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];
```



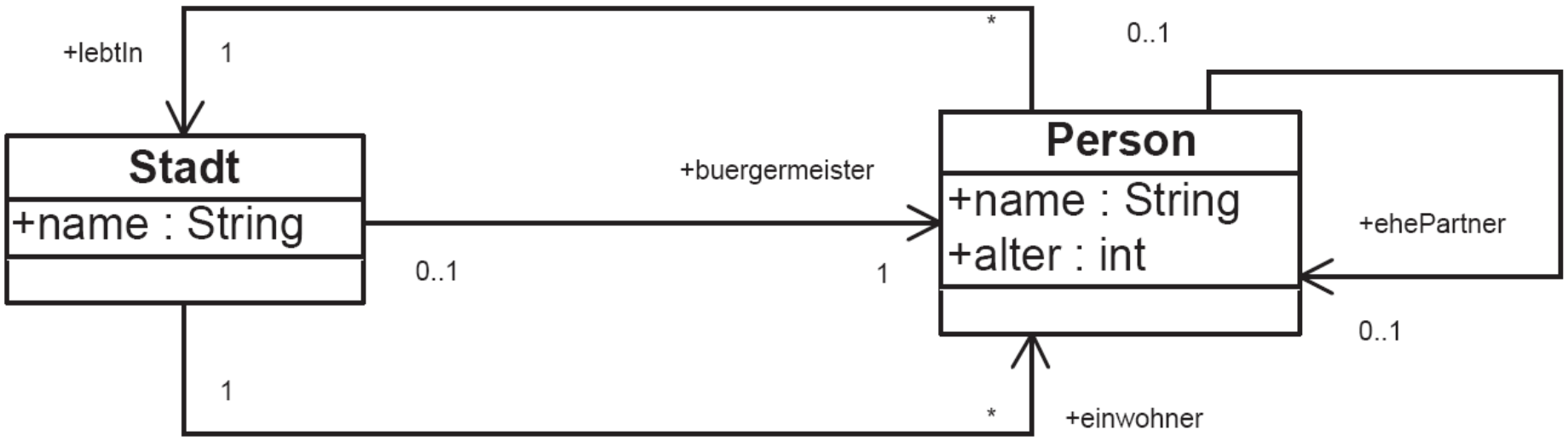
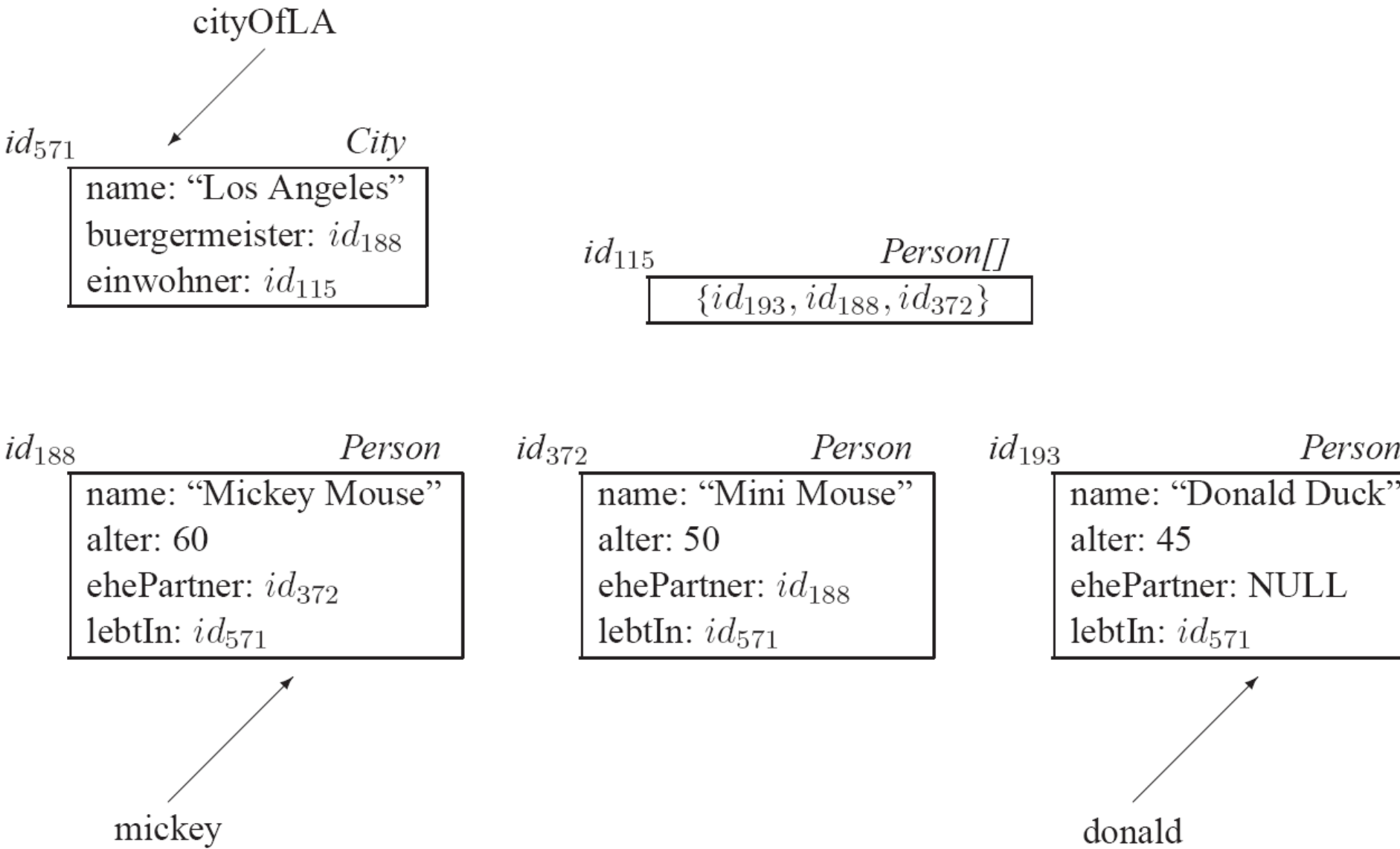


Abbildung 1.12: Die Assoziationen zwischen Person und Stadt

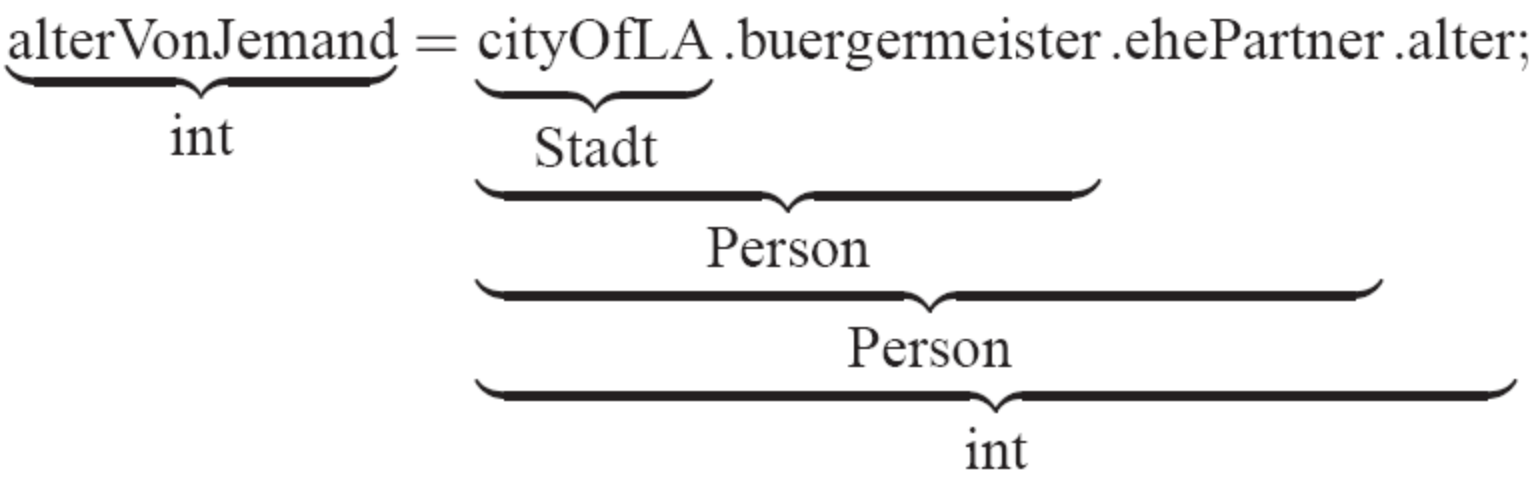
```
class Person {
    public String name;
    public int alter;
    public Person ehePartner;
    public Stadt lebtIn;
}

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

Objekt-Netz


cityOfLA
id_{571}
City
name: "Los Angeles"
buergermeister: id_{188}
einwohner: id_{115}
id_{115}
Person[]
$\{id_{193}, id_{188}, id_{372}\}$
id_{188}
Person
name: "Mickey Mouse"
alter: 60
ehePartner: id_{372}
lebtIn: id_{571}
id_{372}
Person
name: "Mini Mouse"
alter: 50
ehePartner: id_{188}
lebtIn: id_{571}
id_{193}
Person
name: "Donald Duck"
alter: 45
ehePartner: NULL
lebtIn: id_{571}
mickey
donald

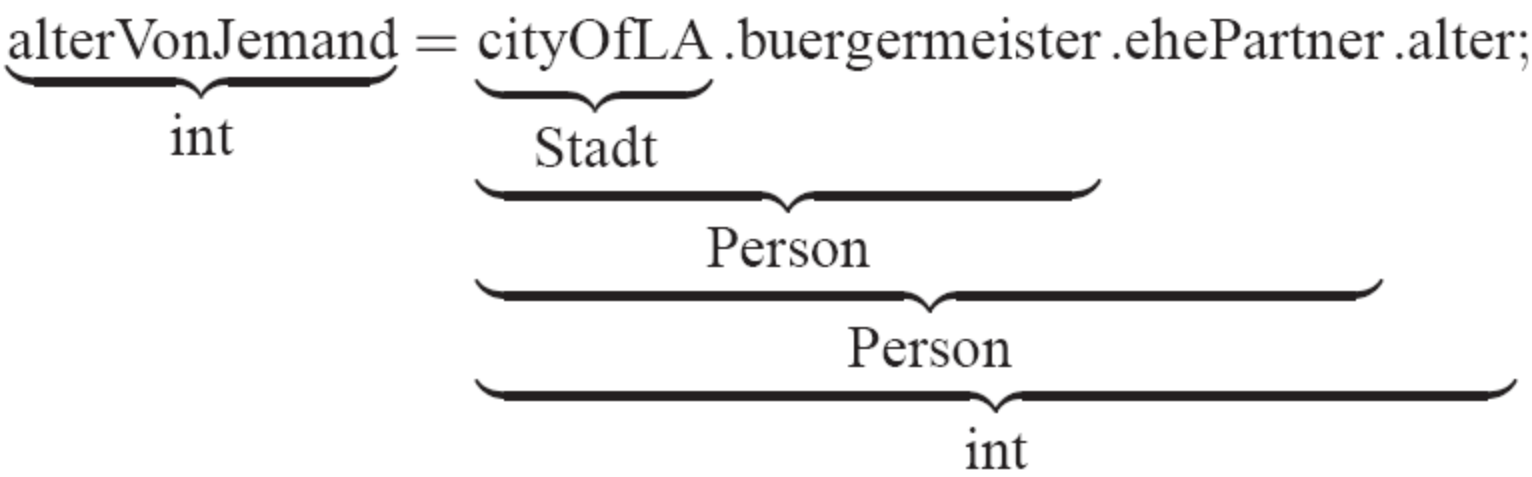
Typisierung von (Pfad-)Ausdrücken

```
   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;
   }
```

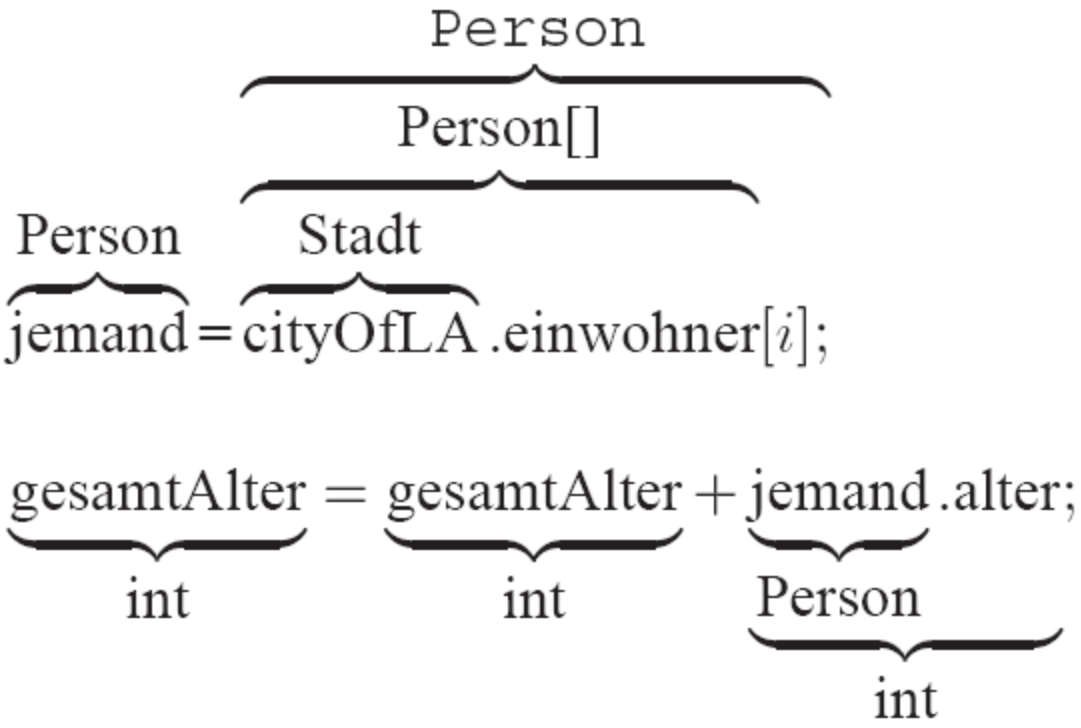


Typisierung von (Pfad-)Ausdrücken

```
   int gesamtAlter, alterVonJemand;
   Person jemand;
   String name;
   ...
(1)alterVonJemand = cityOfLA.buergermeister.ehePartner.alter;
(2)
   for (Person jemand : cityOfLA.einwohner) {
      gesamtAlter = gesamtAlter + jemand.alter;
   }
```

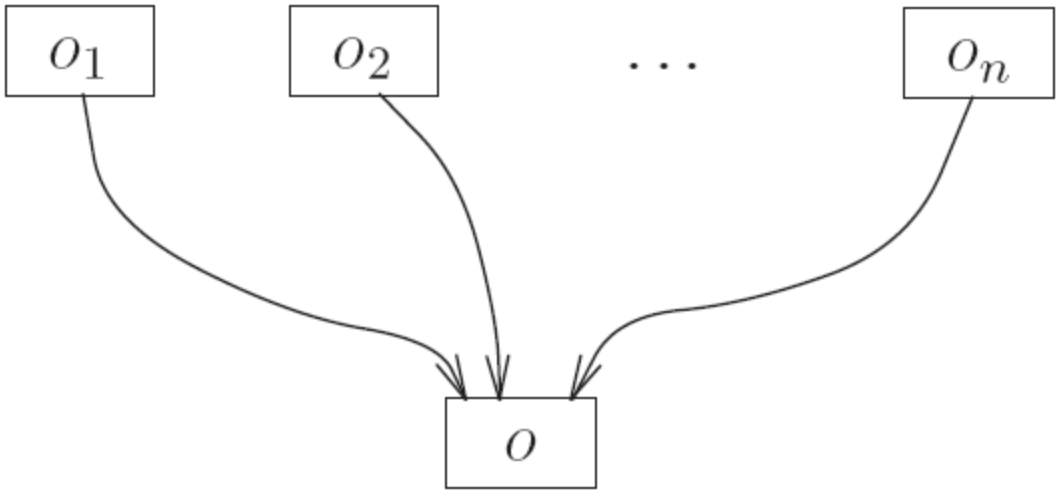


Typisierung ... cont'd



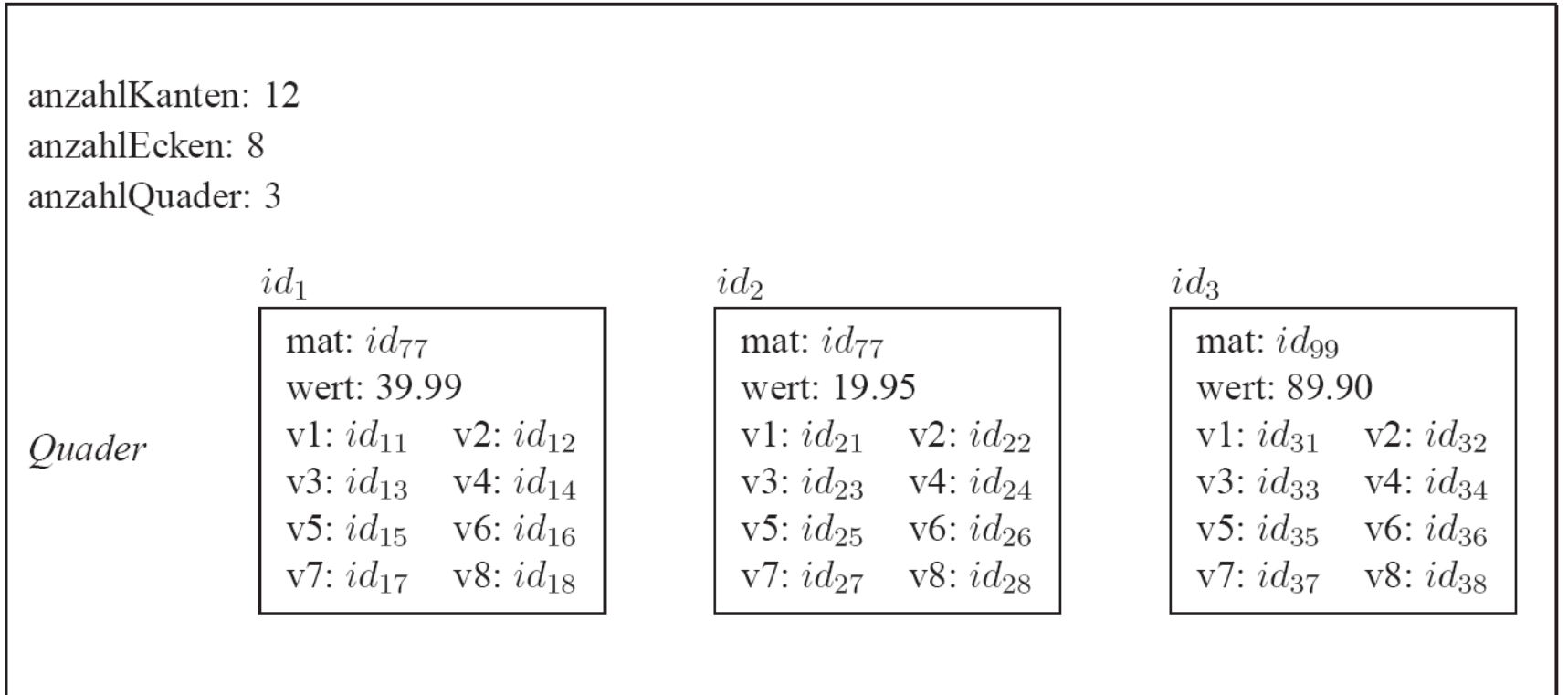
Speicherbereinigung / Garbage Collection

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



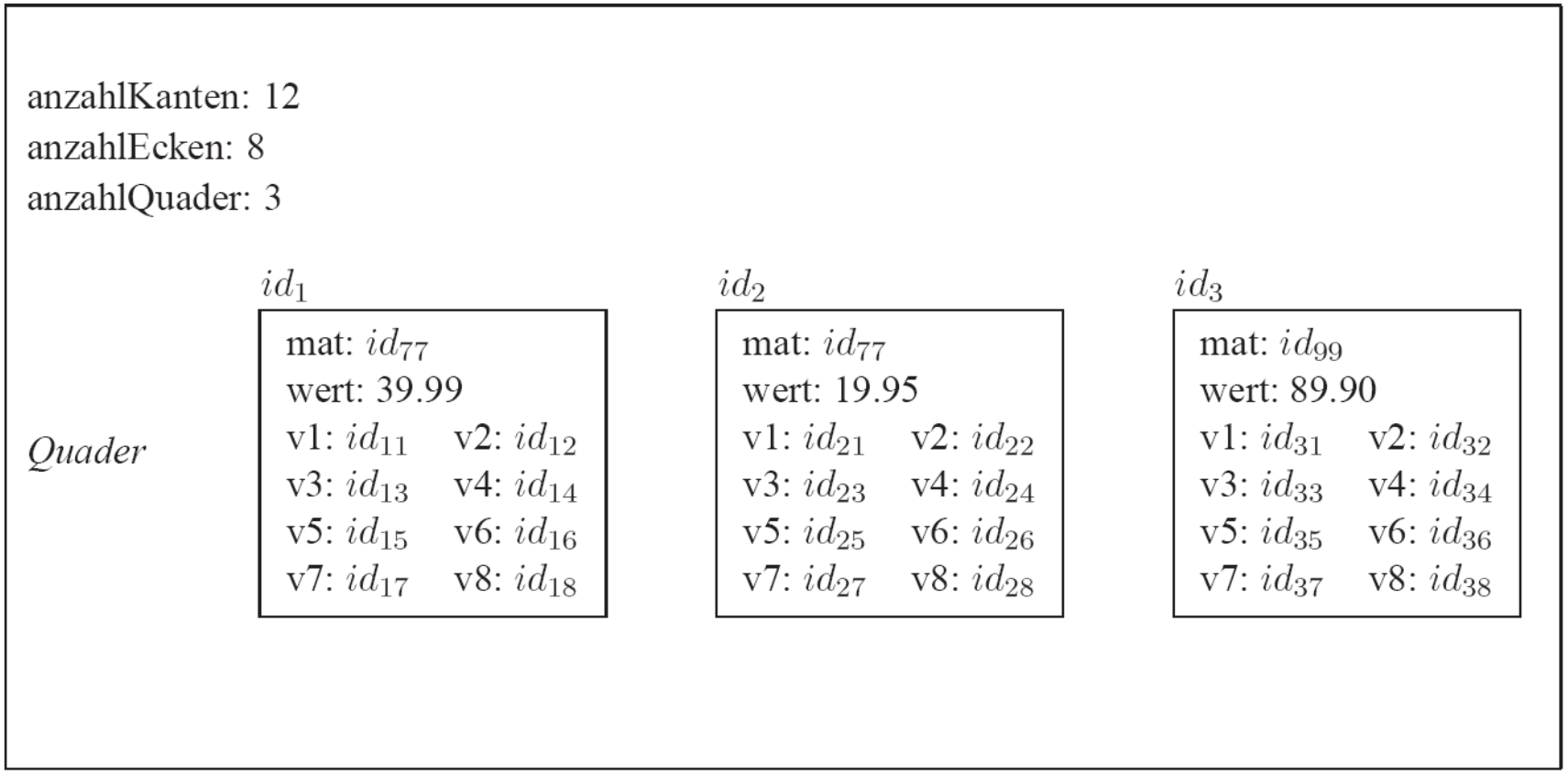
Klassen-Attribute

```
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;
}
```

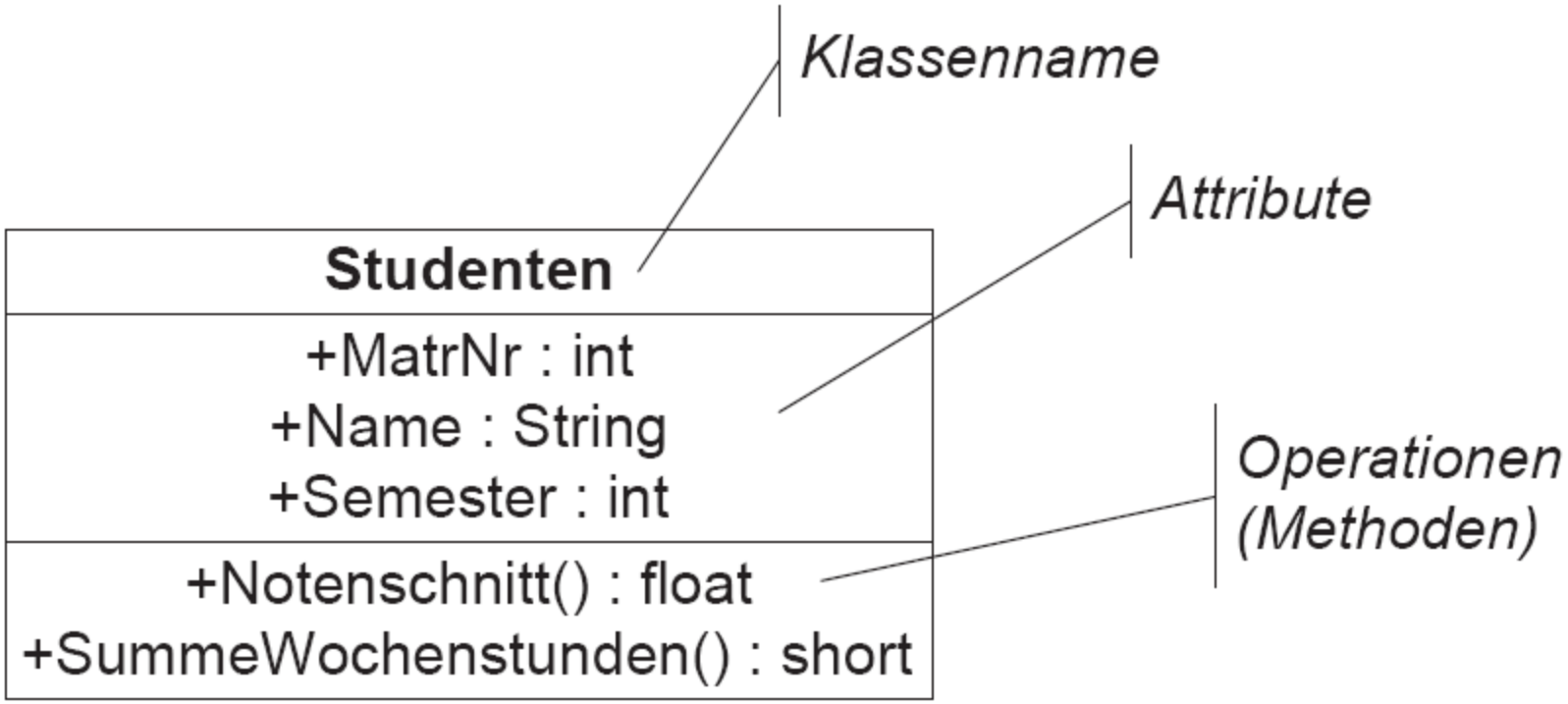


Klassen-Attribute: Zugriff und Modifikation

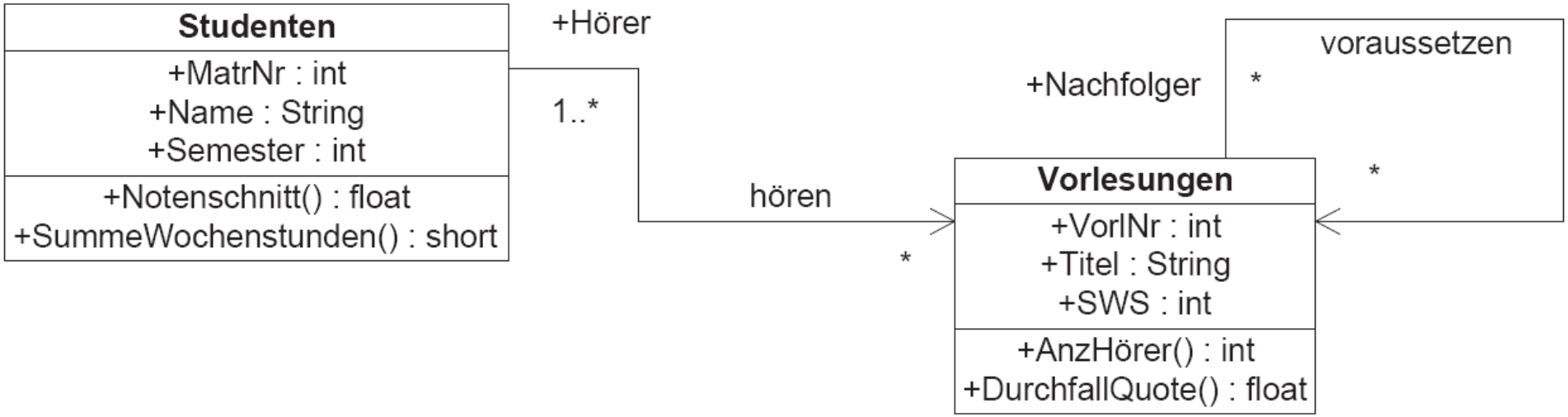
```
System.out.println(Quader.anzahlKanten);
Quader.anzahlQuader = Quader.anzahlQuader + 1;
```



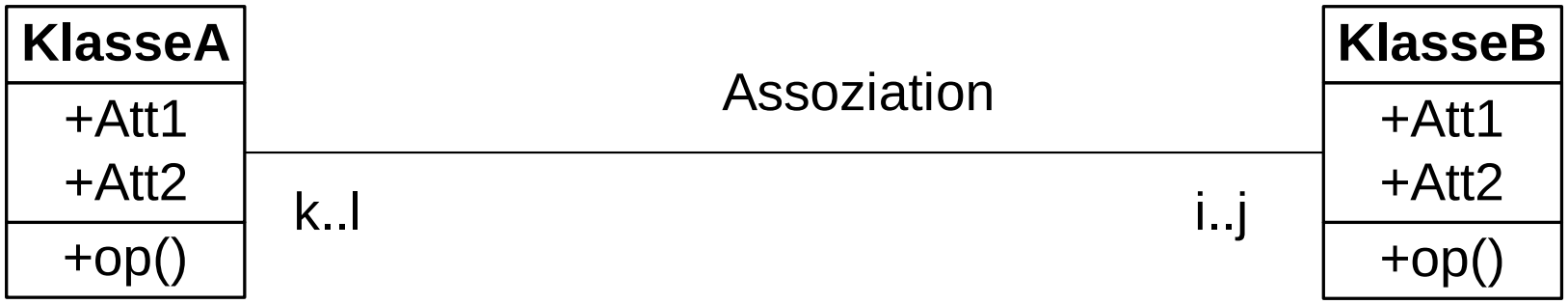
Systematische Modellierung mit UML und Umsetzung in Java



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

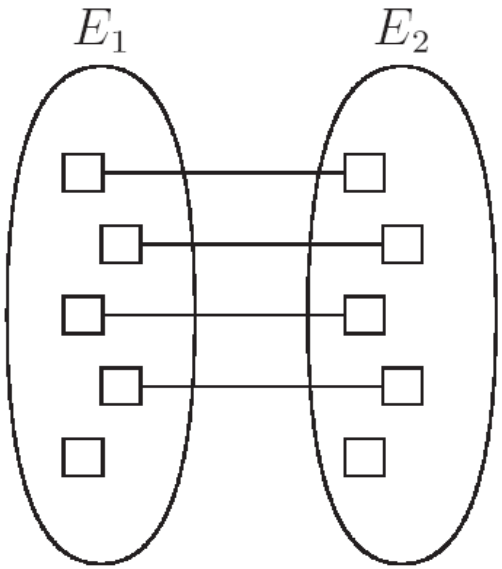
Betrachten wir das abstrakte Beispiel in Abbildung 1.19: Wenn man in UML an einer Seite der binären Assoziation die Multiplizitätsangabe $i..j$ macht, so bedeutet dies, dass jedes Objekt der Klasse auf der anderen Seite mit mindestens i und höchstens j Objekten der Klasse auf dieser Seite in Beziehung 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

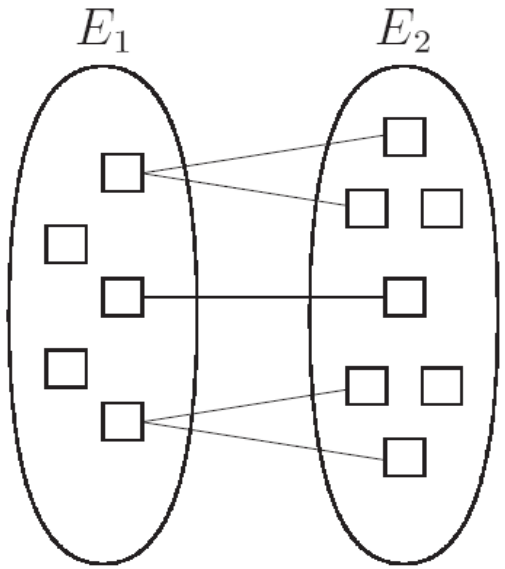
Ehemann : Frauen → Männer

Ehefrau : Männer → Frauen

1:1

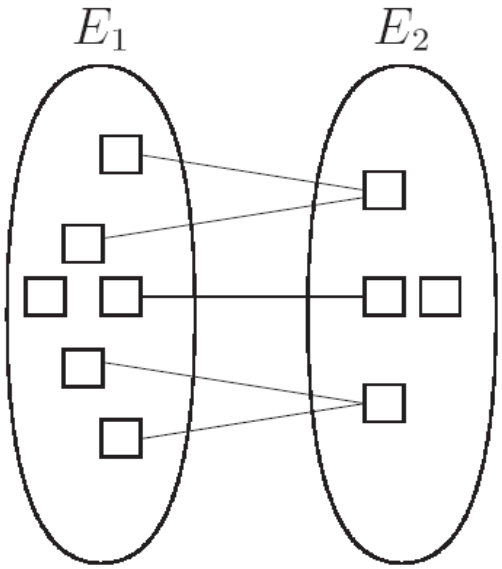


1:N

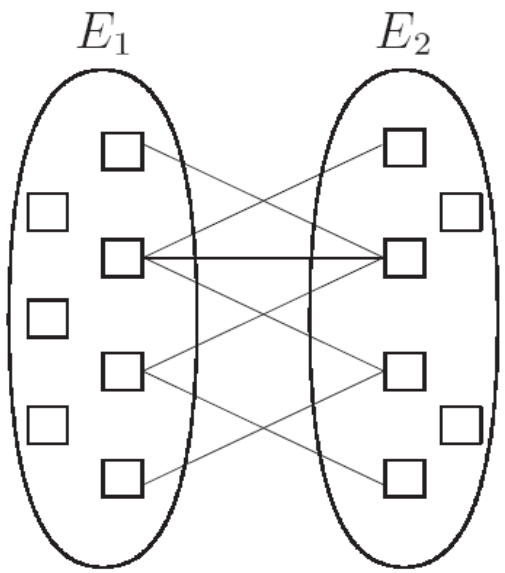


angestelltBei : Personen → Firmen

N:1



N:M



- *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.

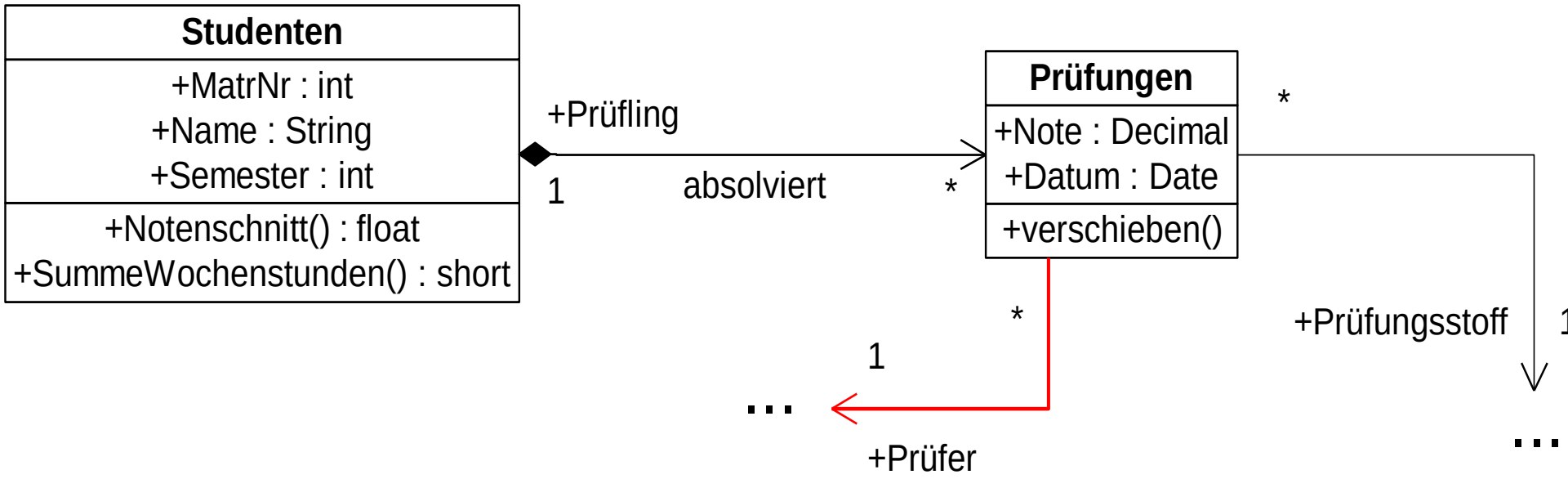
  In der UML-Notation aus Abbildung 1.19 ist eine Eins-zu-viele-Assoziation dadurch gekennzeichnet, dass $l = 1$ und $j > 1$ gilt. Insbesondere kann $j = *$ gelten.

  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.

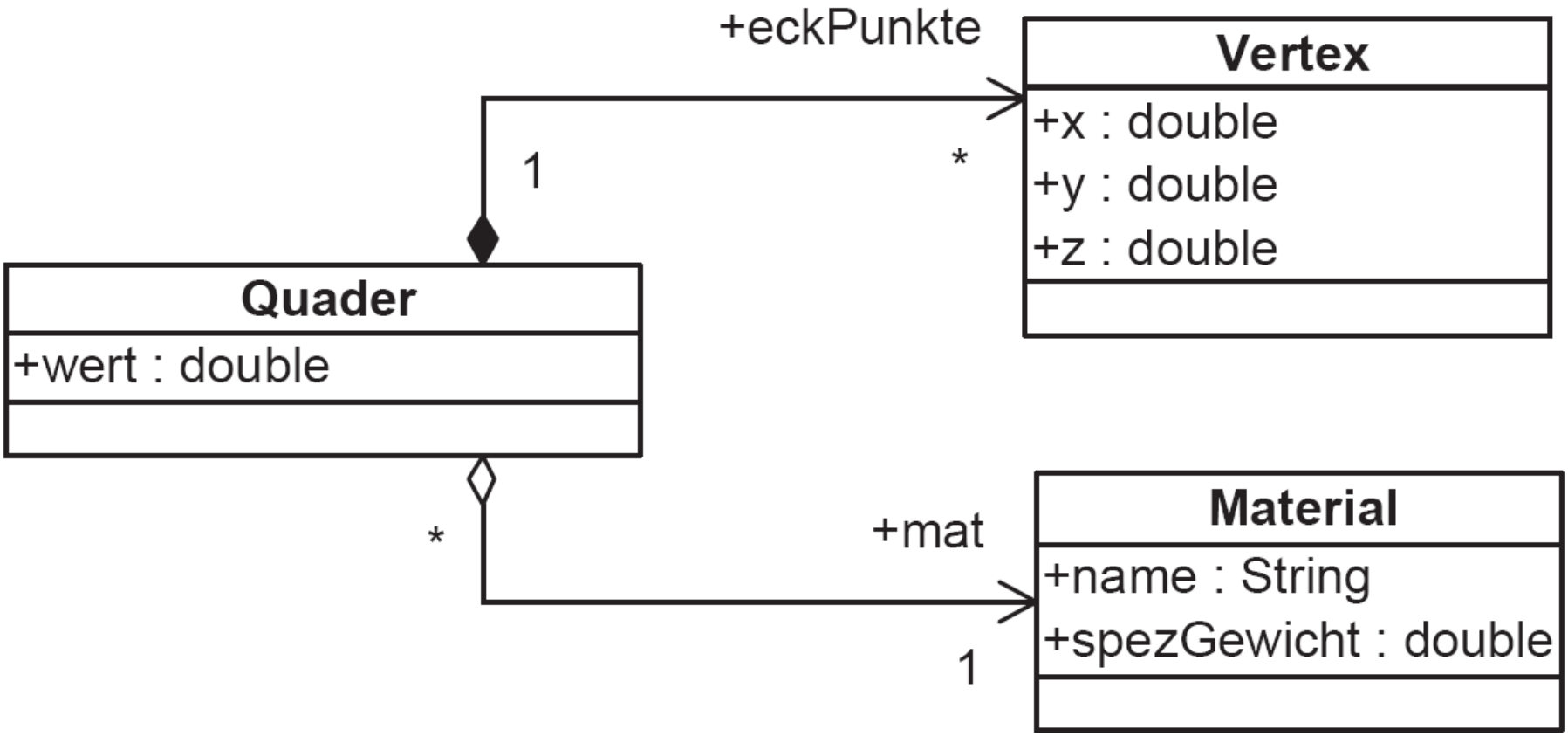
- $N{:}M$-*Beziehung*, wenn keinerlei Restriktionen gelten müssen, d.h. jedes Objekt aus E_1 mit mehreren Objekten aus E_2 in Beziehung stehen kann und umgekehrt jedes Objekt aus E_2 mit mehreren Objekten aus E_1 assoziiert werden darf.

Aggregation/Komposition

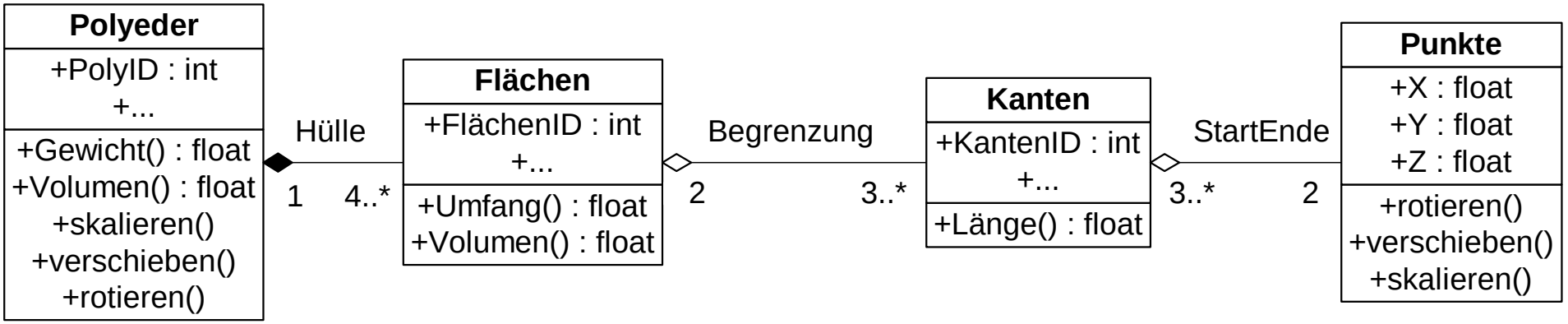


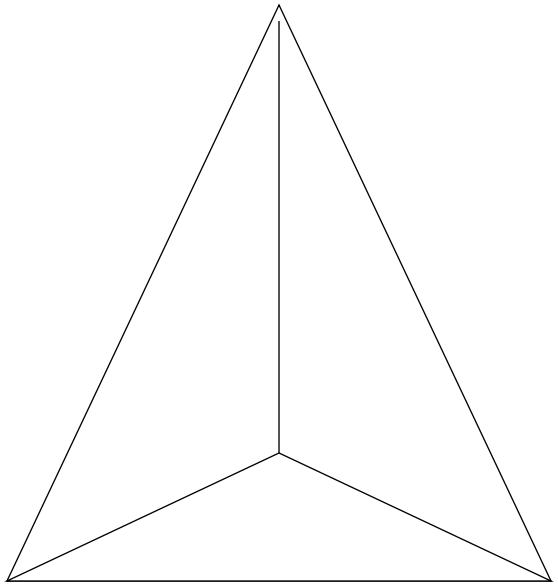
- Komposition (ausgefüllter Diamant)
  - Exklusive Zuordnung
  - Existenzabhängig
- Aggregation („leerer" Diamant)
  - Nicht-exklusive
  - Nicht-existenzabhängige Teil/Ganzes-Beziehung

Aggregation/Komposition

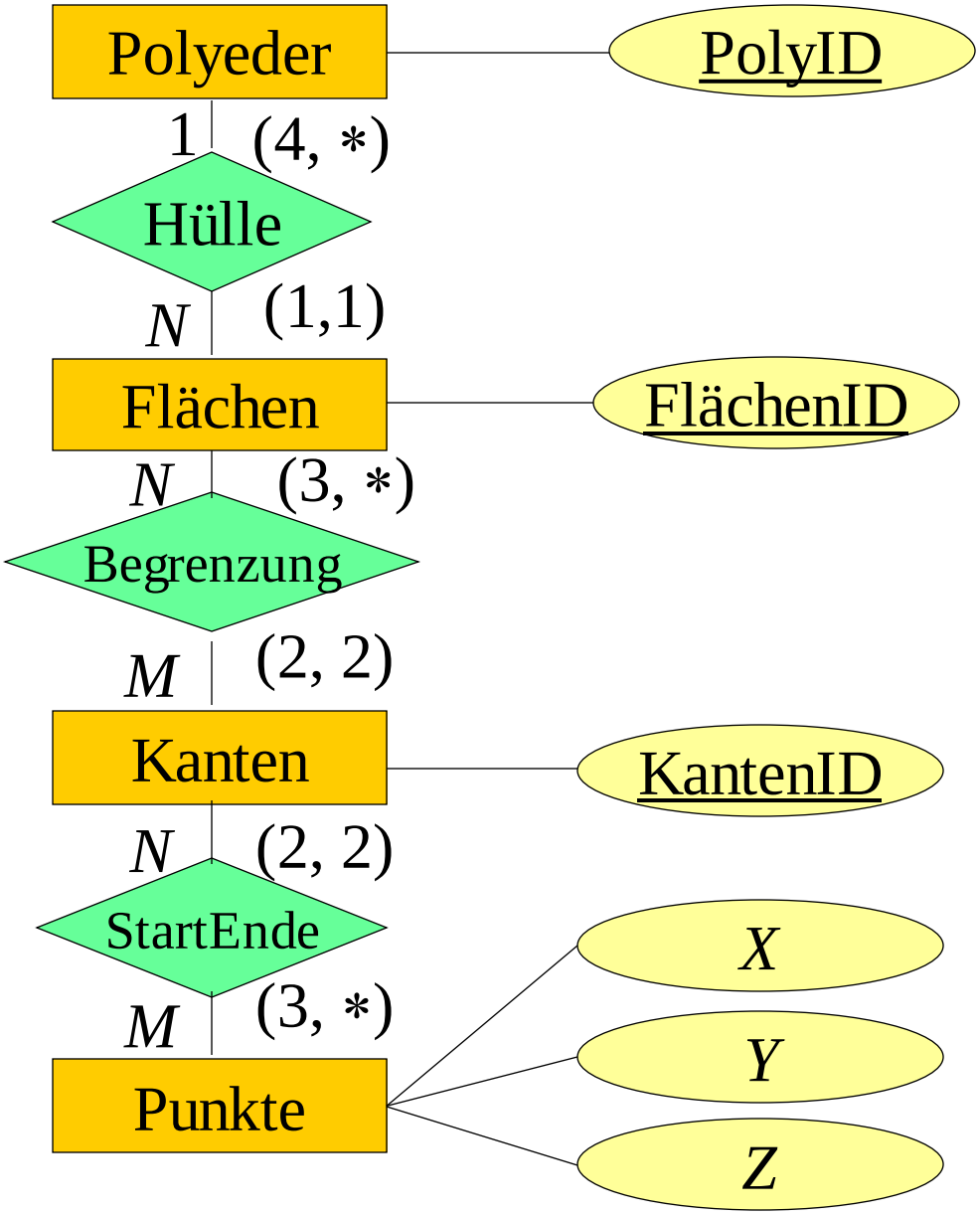


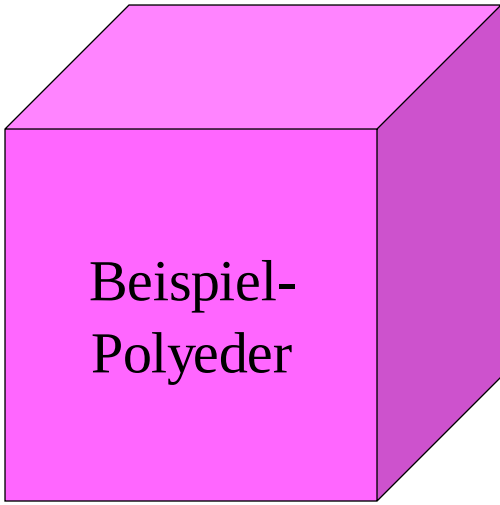
Begrenzungsflächenmodellierung von Polyedern in UML

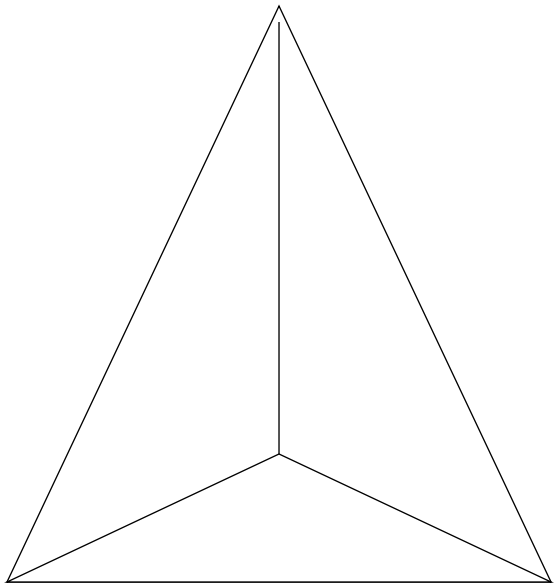




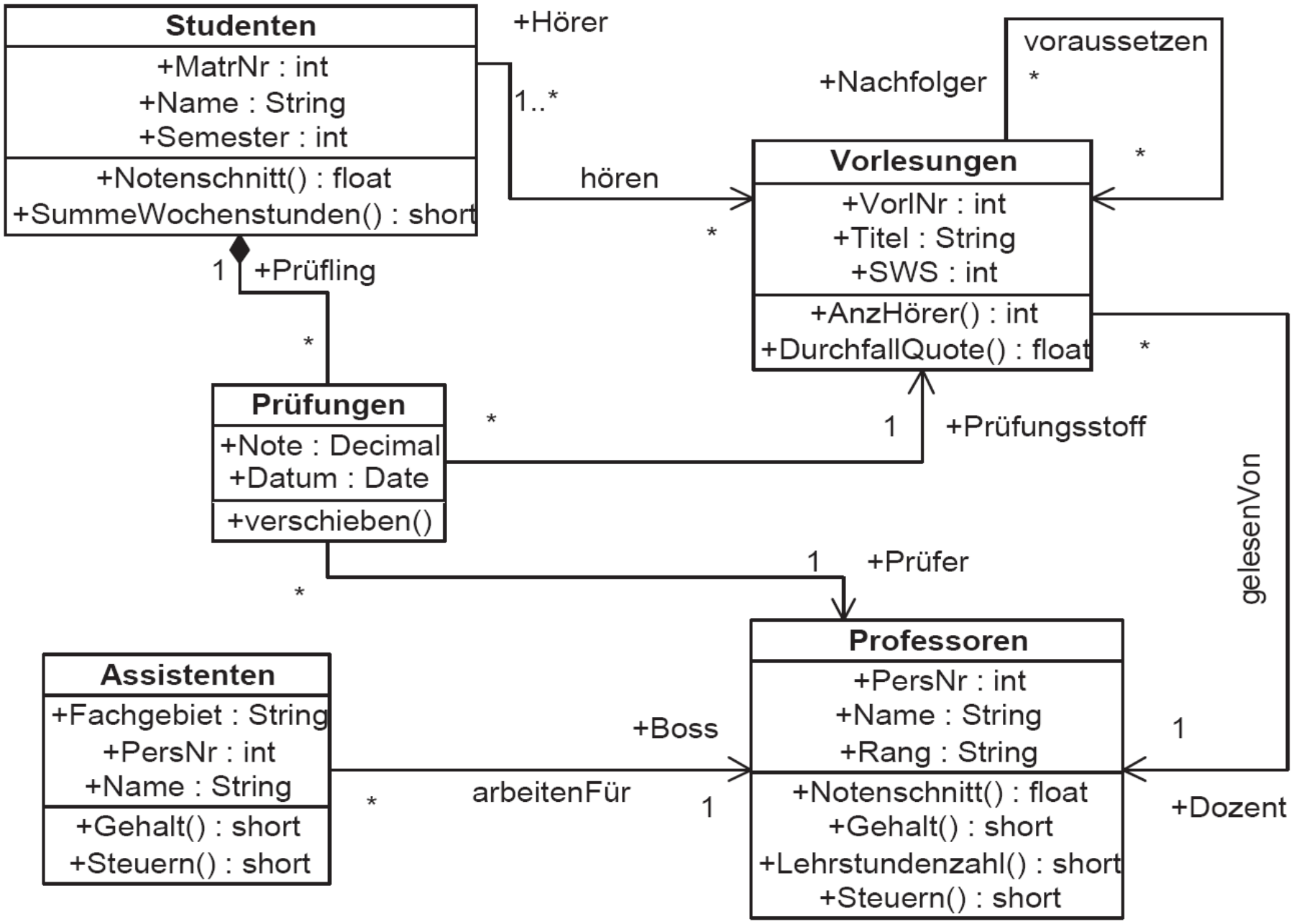
Begrenzungsflächendarstellung

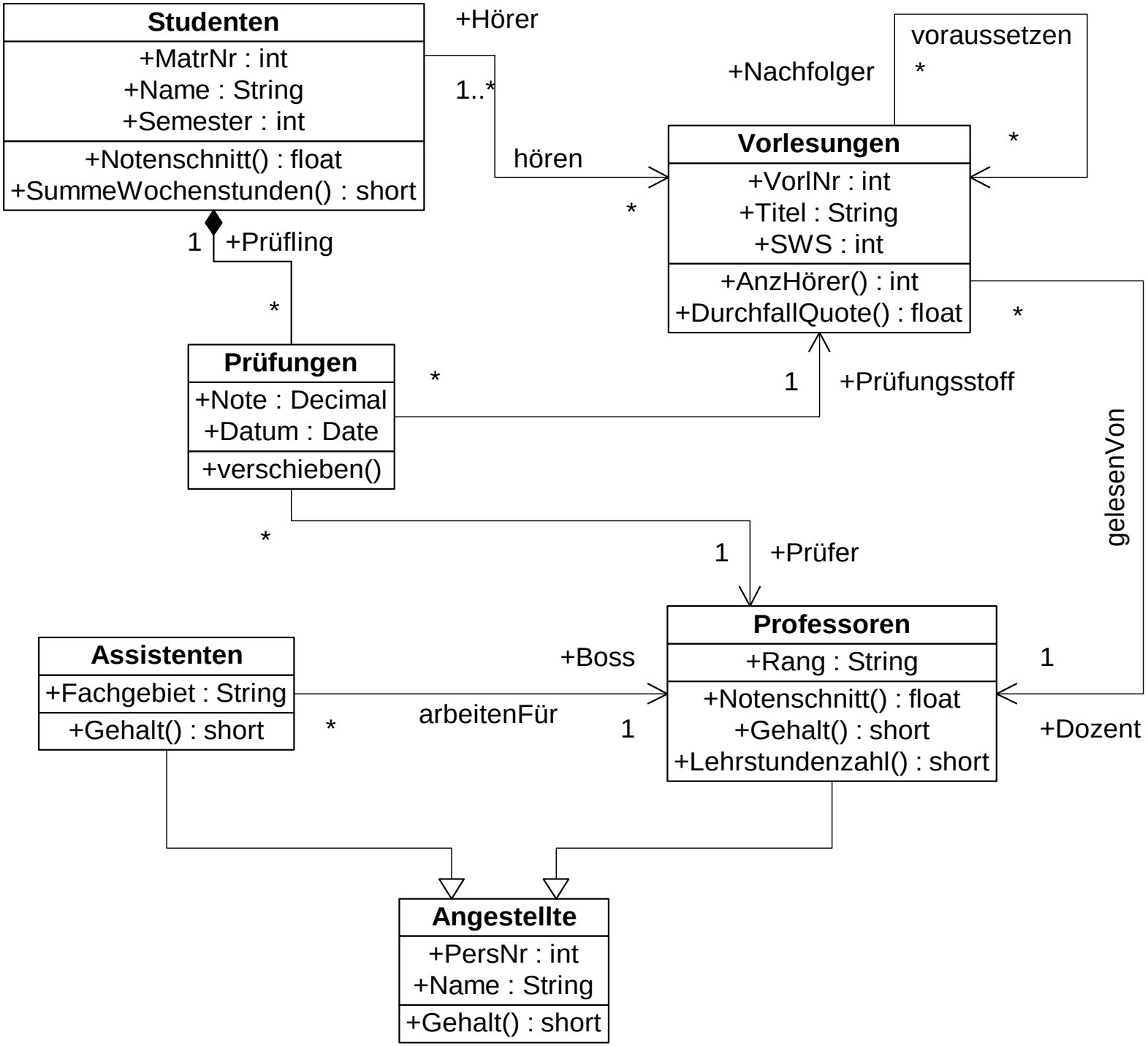






Universitäts-Modell




Studenten
+MatrNr : int
+Name : String
+Semester : int
+Notenschnitt() : float
+SummeWochenstunden() : short
+Hörer
1..*
hören
*
Vorlesungen
+VorlNr : int
+Titel : String
+SWS : int
+AnzHörer() : int
+DurchfallQuote() : float
voraussetzen
*
+Nachfolger
*
1
+Prüfling
*
Prüfungen
+Note : Decimal
+Datum : Date
+verschieben()
*
1
+Prüfungsstoff
*
gelesenVon
*
1
+Prüfer
Assistenten
+Fachgebiet : String
+Gehalt() : short
*
arbeitenFür
+Boss
1
Professoren
+Rang : String
+Notenschnitt() : float
+Gehalt() : short
+Lehrstundenzahl() : short
1
+Dozent
Angestellte
+PersNr : int
+Name : String
+Gehalt() : short

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() {}
}
```

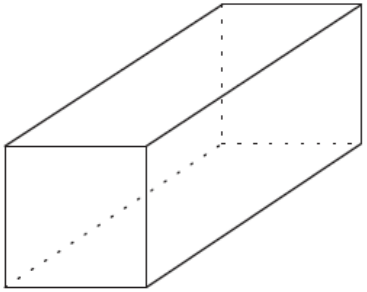
```
class Quader2 {
    public Vertex[] eckPunkte;
    public Material mat;
    public double wert;
}
```

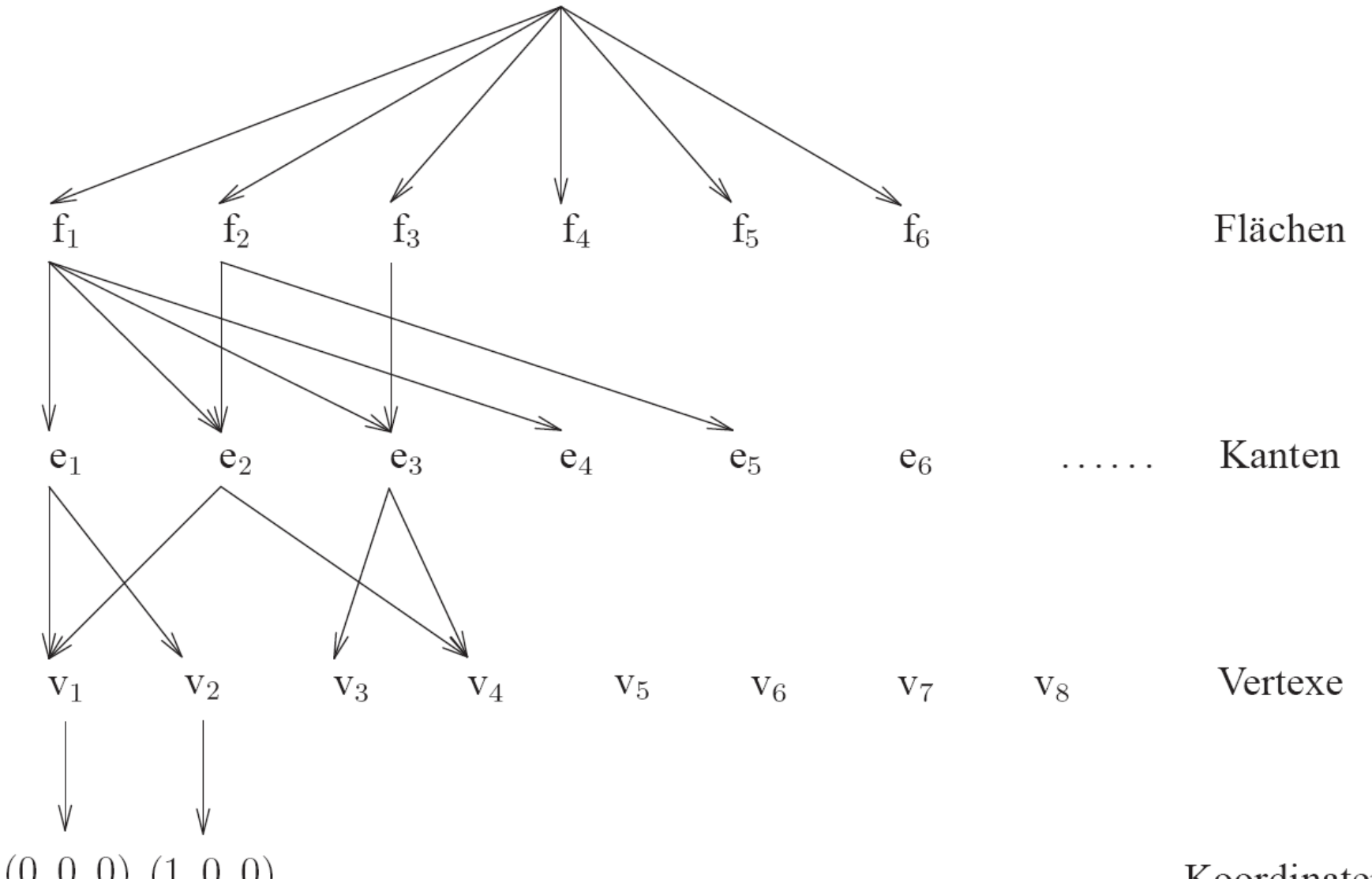
Umsetzung einer Assoziation in Java viele-viele (N:M)


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

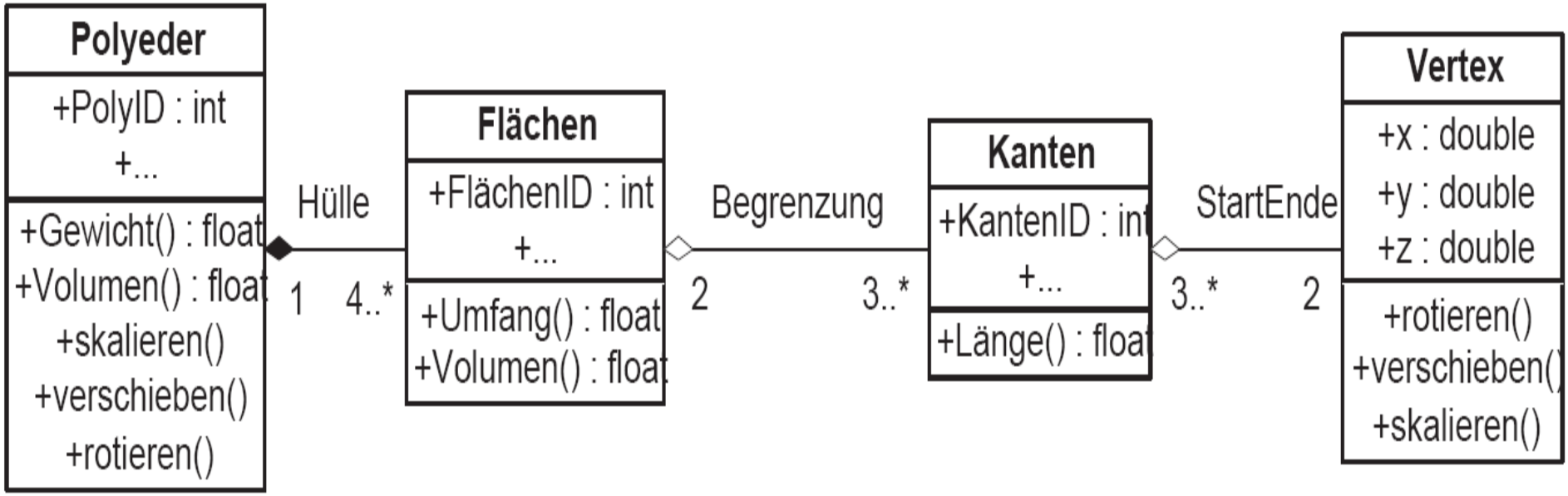
```
class E2 {
    public ... att21;
    public ... att22;
    public E1[] zugeordneteE1;
    public ... op2() {}
}
```

Begrenzungs-Flächen-Modell

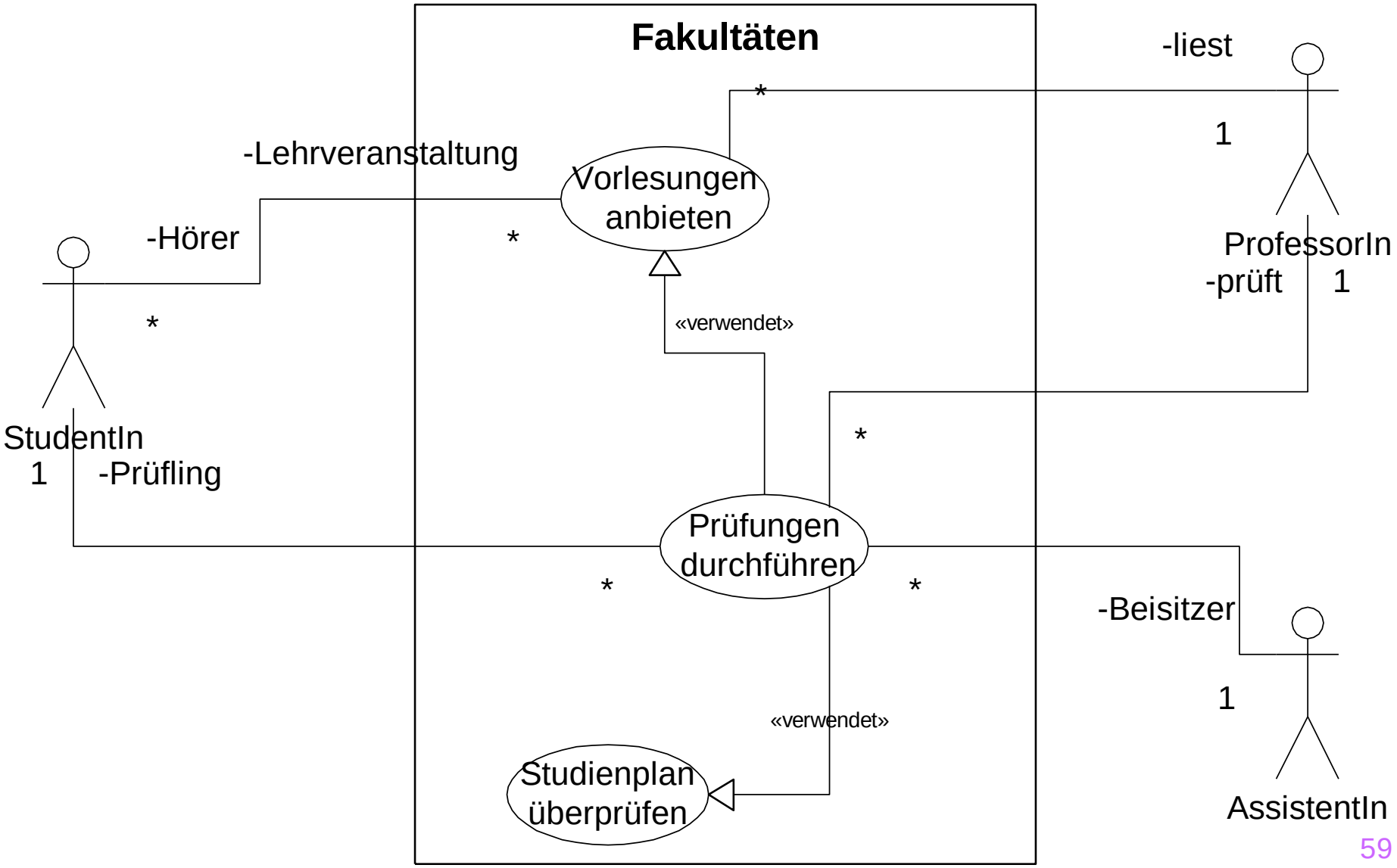




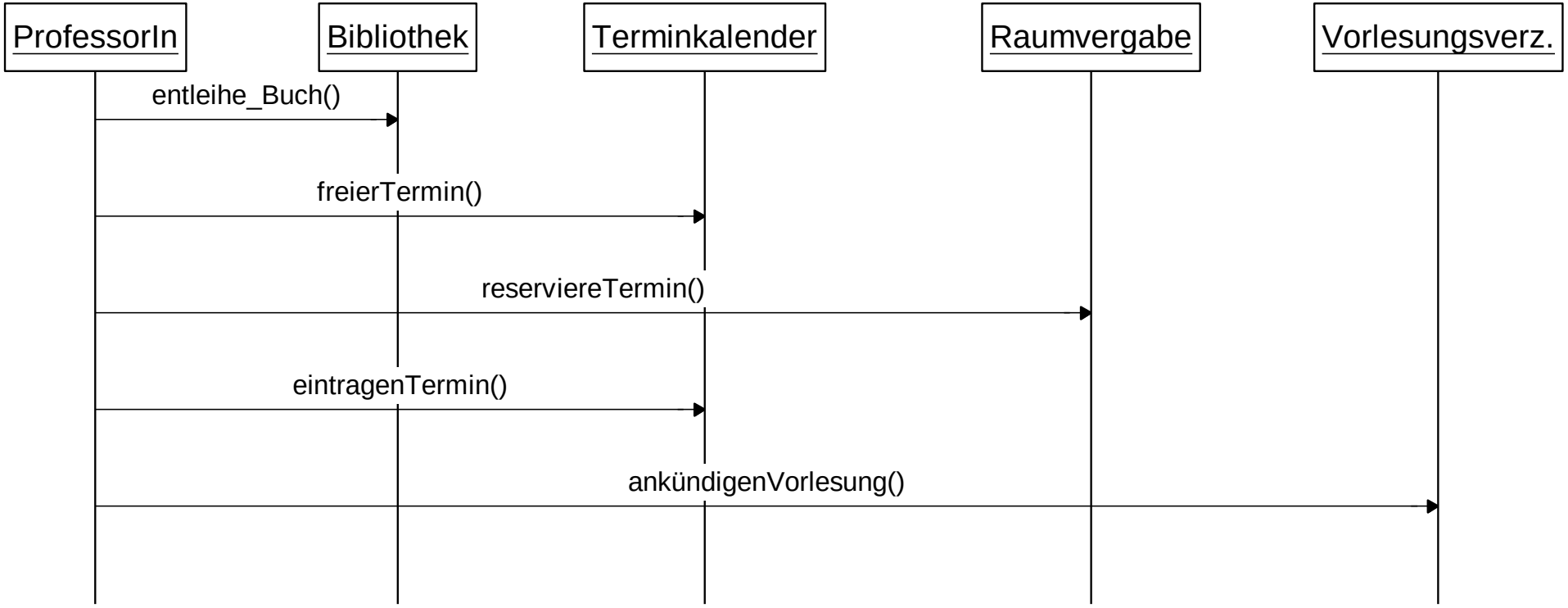
Polyeder in UML



Anwendungsfälle (use cases)



Interaktions-Diagramm: Modellierung komplexer Anwendungen



Interaktions-Diagramm:
Prüfungsdurchführung