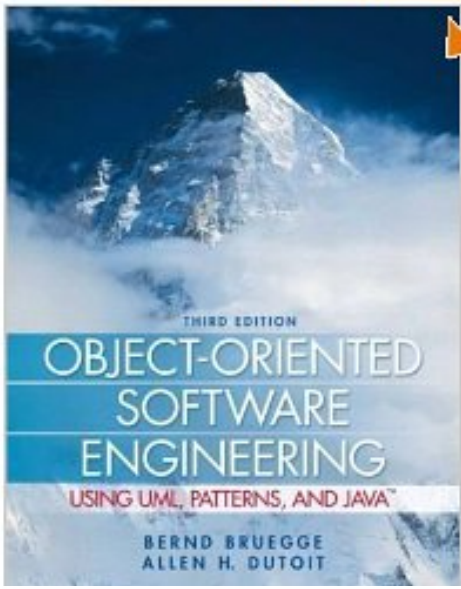
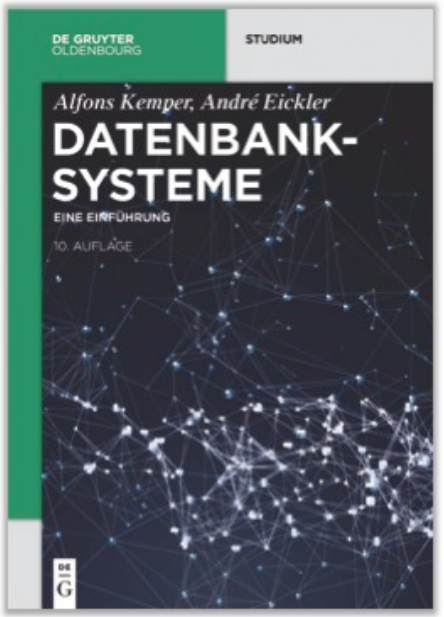
Einführung in die Informatik II für Ingenieurwissenschaften (MSE)

- 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



Generalisierung/Spezialisierung
Subtypisierung/Vererbung

- Bringt Struktur in die Klassen-Diagramme
- Erhöht die Wiederverwendbarkeit
- Erlaubt die schrittweise Verfeinerung

Motivation: Problem der Wiederverwendung

```
public class Person {
    public String name;
    public int alter;
    public Person ehePartner;
    // ...
    public Person(String n, int a) {
        this.name = n; this.alter = a;
    }
    public void heiraten(Person partner) {
        this.ehePartner = partner;
    }
    // ...
} // public class Person
```

Motivation; cont'd

```
public class Angestellter {
    public String name;
    public int alter;
    public Person ehePartner;
    public int steuerNr;
    public double gehalt;
    public Angestellter boss;

    public Angestellter(String n, int a, int s, double g) {
        this.name = n;
        this.alter = a;
        this.steuerNr = s;
        this.gehalt = g;
    }

    public void heiraten(Person partner) {
        this.ehePartner = partner;
    }

    public boolean istPensioniert() {
        return (this.alter > 64);
    }
} // public class Angestellter
```

Keine Wiederverwendung

Angestellte können nur Personen heiraten???

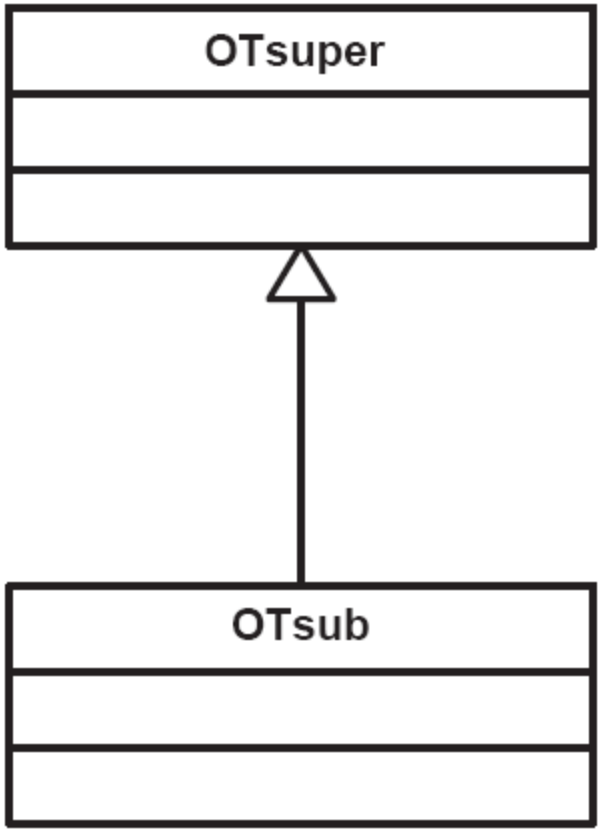
Zwei schwerwiegende Problem gibt es mit diesen beiden Klassendefinitionen:

1. *Mangelnde Wiederverwendbarkeit*
   Die Klasse *Person* enthält viele Komponenten, die in der Klasse *Angestellter* in gleicher Form nochmals repliziert wurden. Es wäre vorteilhafter gewesen, die Definition der Klasse *Angestellter* auf der Definition der Klasse *Person* aufzubauen.

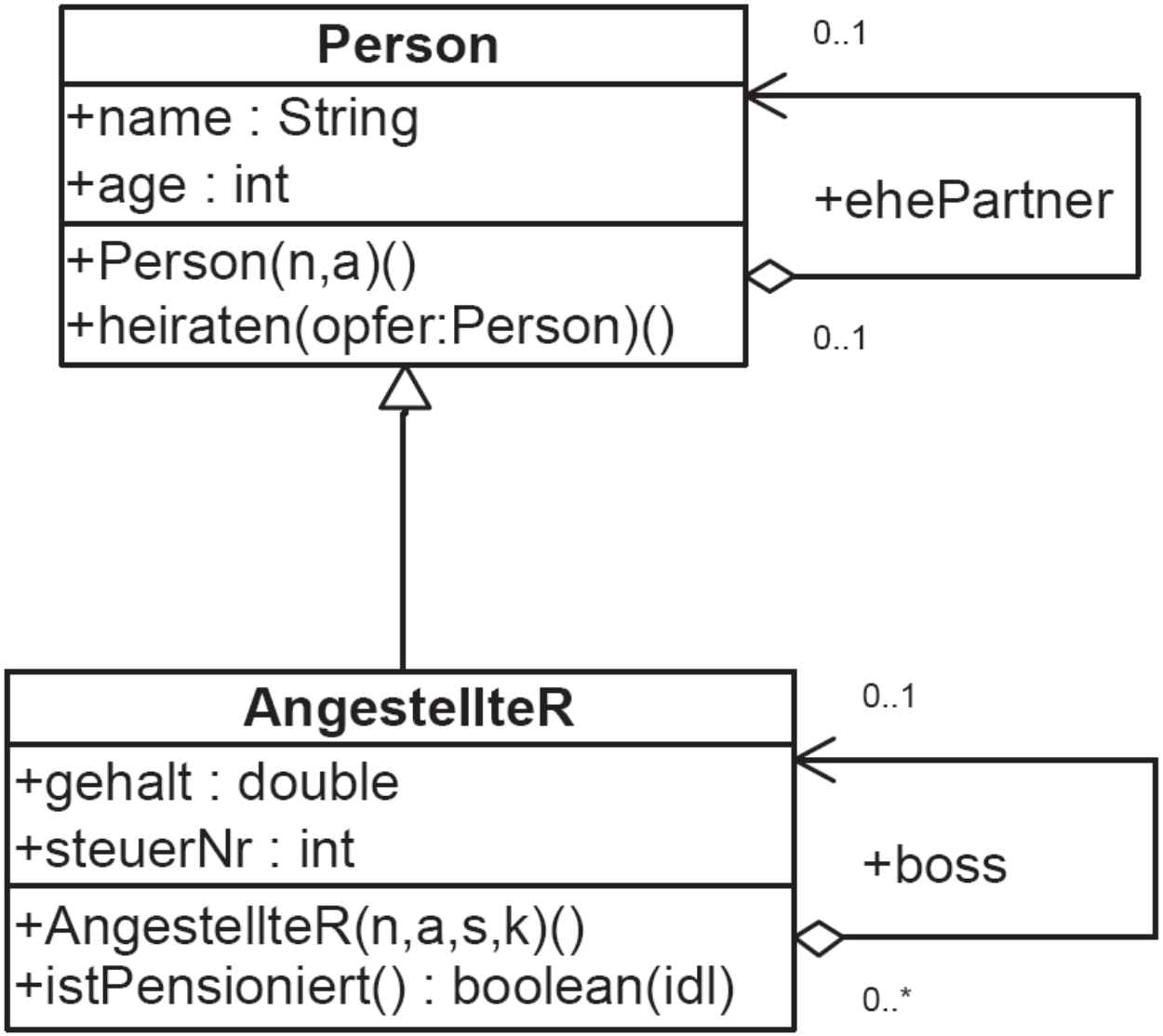
2. *Mangelnde Flexibilität*
   Diese Problem der mangelnden Flexibilität ist noch schwerwiegender. Es wird dadurch verursacht, dass die beiden Klassen völlig isoliert voneinander definiert sind, und die Instanzen der beiden Klassen sich nicht gegenseitig vertreten können. Wir wollen dies an dem Attribut *ehePartner* der Klasse *Person* illustrieren. Dieses Attribut ist genau wie das Argument *partner* der Operation *heiraten* auf Objekte vom Typ *Person* eingeschränkt. Dies bedeutet, dass niemand eine/n Angestellte/n heiraten kann. Das folgende Programmfragment illustriert dieses Problem:

```
Angestellter miniMouse;
Person mickeyMouse;
...
miniMouse.heiraten(mickeyMouse);    // okay, mickeyMouse ist eine Person
mickeyMouse.heiraten(miniMouse);    // illegal, miniMouse ist keine Person
```

Subtypisierung: Overtyp/Untertyp



Generalisierung/Spezialisierung



extends

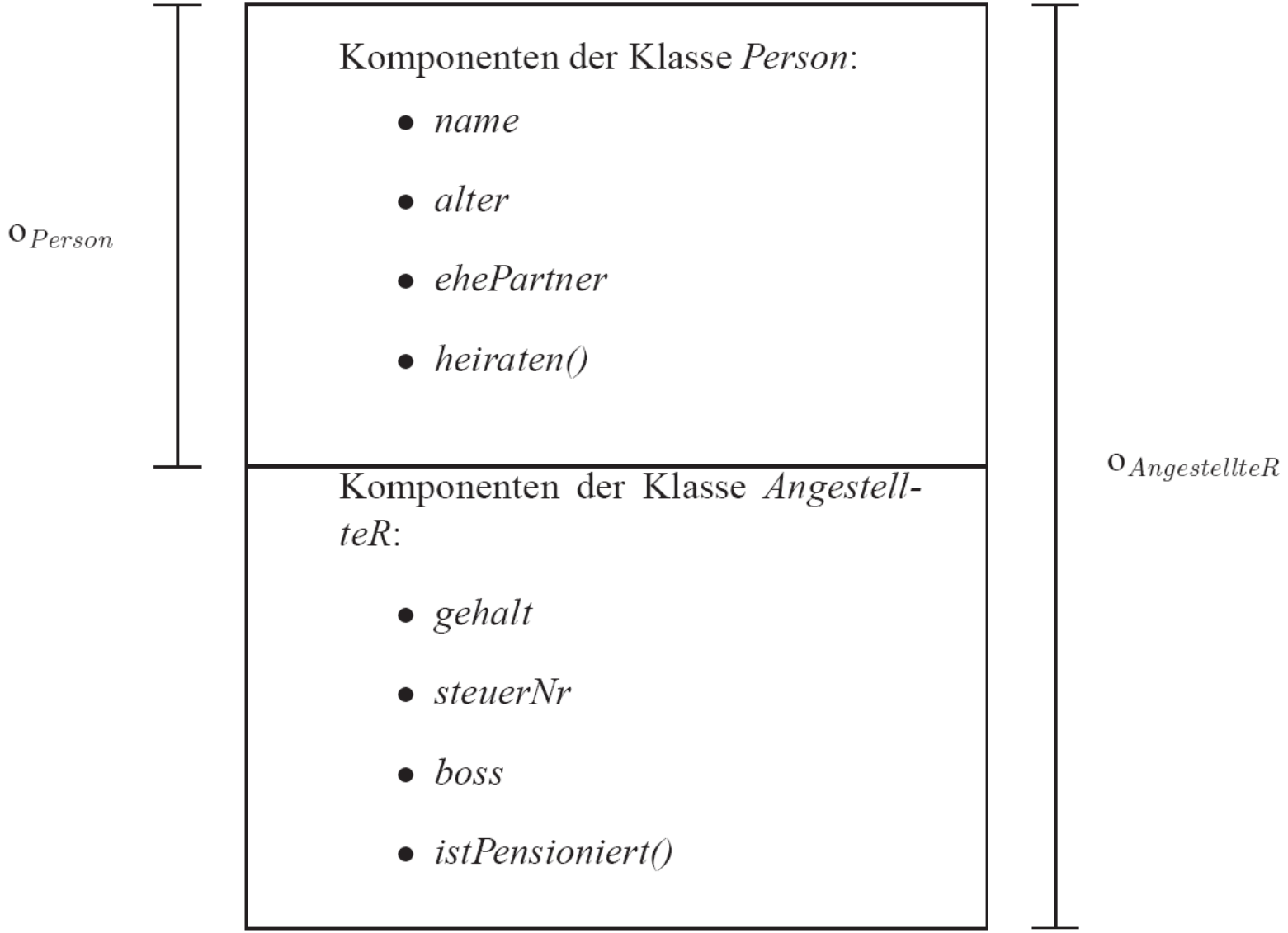
```
class Angestellter extends Person {
    public int steuerNr;
    public double gehalt;
    public Angestellter boss;

    Angestellter(String n, int a, int s, double knete) {
        super(n, a);
        this.steuerNr = s;
        this.gehalt = knete;
    }

    public boolean istPensioniert() {
        return (this.alter > 64);
    }

} // class Angestellter
```

Substituierbarkeit: Typ-Sicherheit immer noch gewährleistet

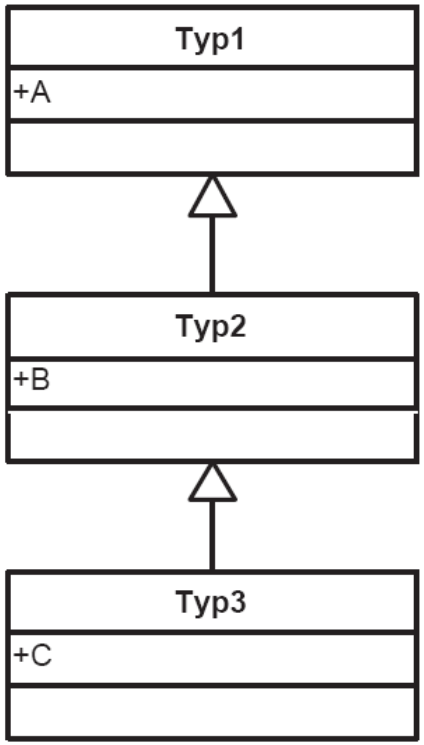
```
miniMouse = new Angestellter("Mini Mouse", 60, 007, 90000.0);
mickeyMouse = new Person("Mickey Mouse", 50);
```

```
Angestellter miniMouse;
Person mickeyMouse;
...
miniMouse.heiraten(mickeyMouse);    // okay, mickeyMouse ist eine Person
mickeyMouse.heiraten(miniMouse);    // jetzt okay, miniMouse ist
                                    als Angestellter auch eine Person
```

Vererbung: am Beispiel erläutert (eine Subtyp-Instanz „kann mehr“)



Generalisierung-Hierarchie



Instanzen

Typ1

id_1

A: . . .

Typ2

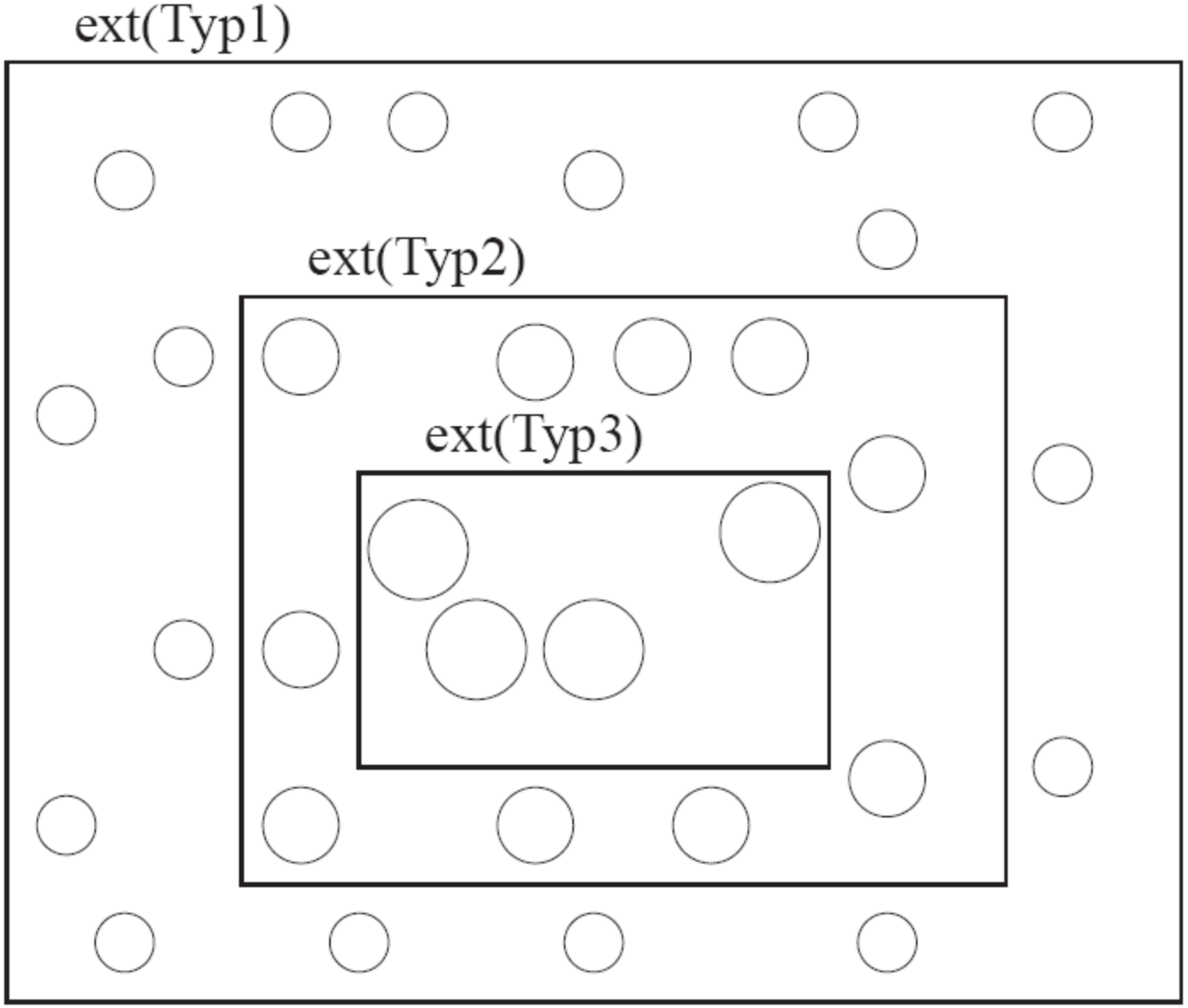
id_2

A: . . .
B: . . .

Typ3

id_3

A: . . .
B: . . .
C: . . .

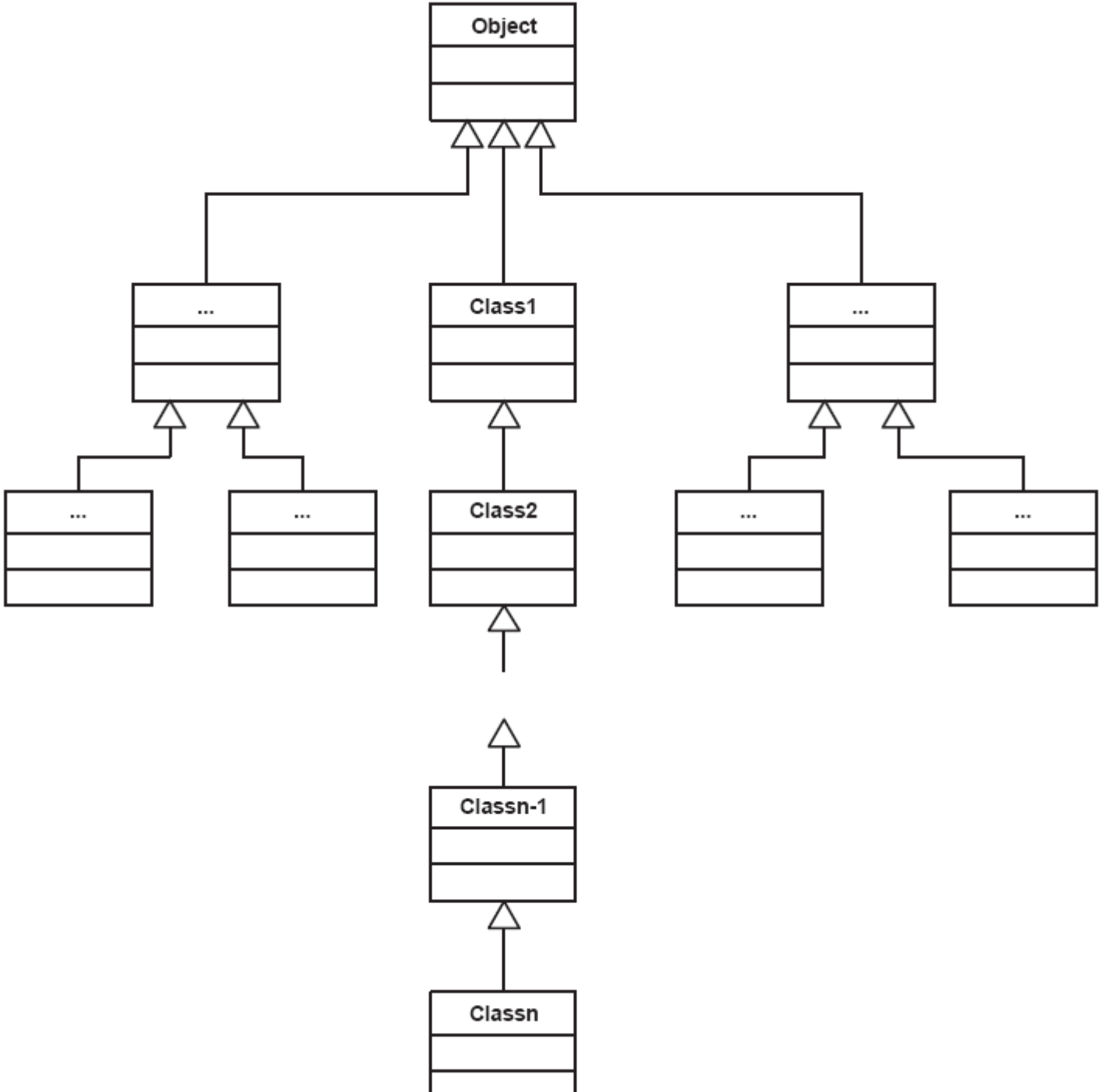


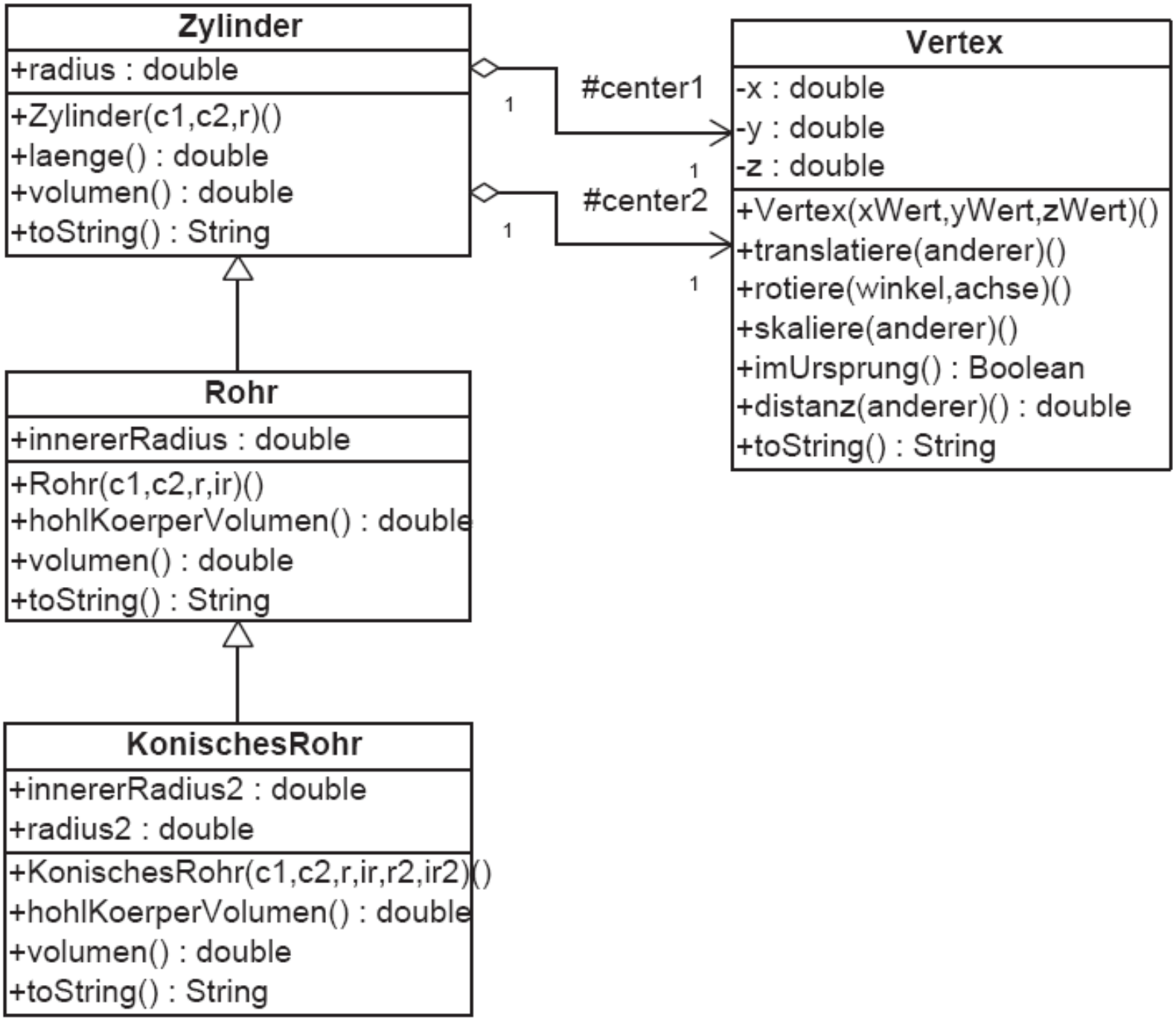
Visualisierung der Subtypisierung/Mengeninklusion

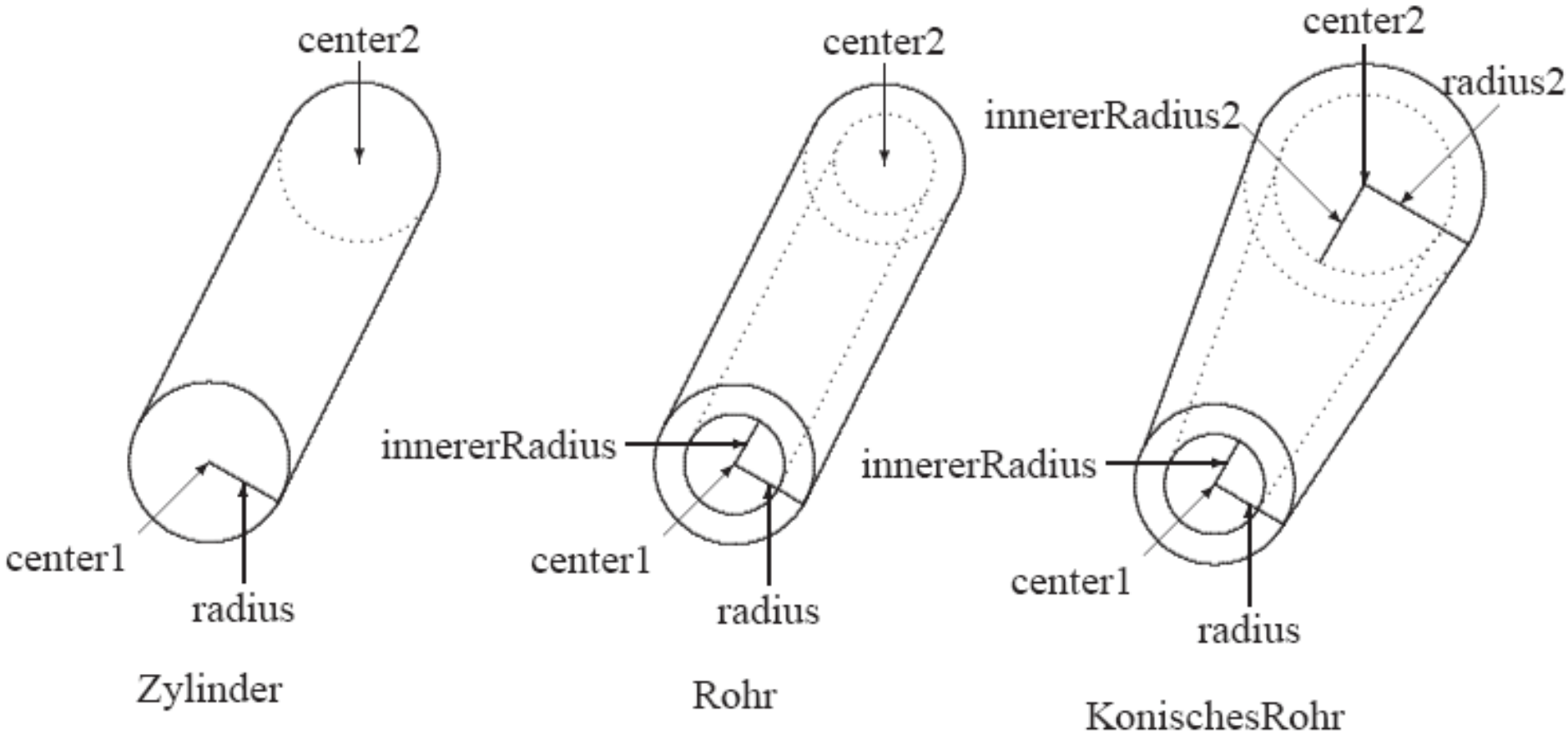
Wurzeltyp: Object

```
class OT {
   public ...;
}
```

```
class OT extends Object {
   public ...;
}
```


Object
...
Class1
...
...
...
Class2
...
...
Classn-1
Classn


Zylinder
+radius : double
+Zylinder(c1,c2,r)()
+laenge() : double
+volumen() : double
+toString() : String
Vertex
-x : double
-y : double
-z : double
+Vertex(xWert,yWert,zWert)()
+translatiere(anderer)()
+rotiere(winkel,achse)()
+skaliere(anderer)()
+imUrsprung() : Boolean
+distanz(anderer)() : double
+toString() : String
#center1
#center2
1
1
1
1
Rohr
+innererRadius : double
+Rohr(c1,c2,r,ir)()
+hohlKoerperVolumen() : double
+volumen() : double
+toString() : String
KonischesRohr
+innererRadius2 : double
+radius2 : double
+KonischesRohr(c1,c2,r,ir,r2,ir2)()
+hohlKoerperVolumen() : double
+volumen() : double
+toString() : String


center2
center1
radius
Zylinder
center2
innererRadius
center1
radius
Rohr
center2
radius2
innererRadius2
innererRadius
center1
radius
KonischesRohr

```
1  public class Zylinder extends Object {
2      public double radius;
3      protected Vertex center1;
4      protected Vertex center2;
5
6      public Zylinder(Vertex c1, Vertex c2, double r) {
7          this.center1 = c1;
8          this.center2 = c2;
9          this.radius = r;
10     }
11
12     public double laenge() {
13         return this.center1.distanz(this.center2);
14     }
15
16     public double volumen() {
17         return this.radius * this.radius * Math.PI * this.laenge();
18     }
19
20     public String toString() {
21         return "Radius:␣" + this.radius + "␣Center1:" + this.center1 +
22             "␣Center2:␣" + this.center2 + "␣Länge:␣" + this.laenge() +
23             "␣Volumen:␣" + this.volumen();
24     }
25
26     public static void main(String args[]) {
27         Zylinder c = new Zylinder(new Vertex(1,2,3), new Vertex(2,3,4), 5.5);
28         System.out.println(c);
29     }
30 }
```

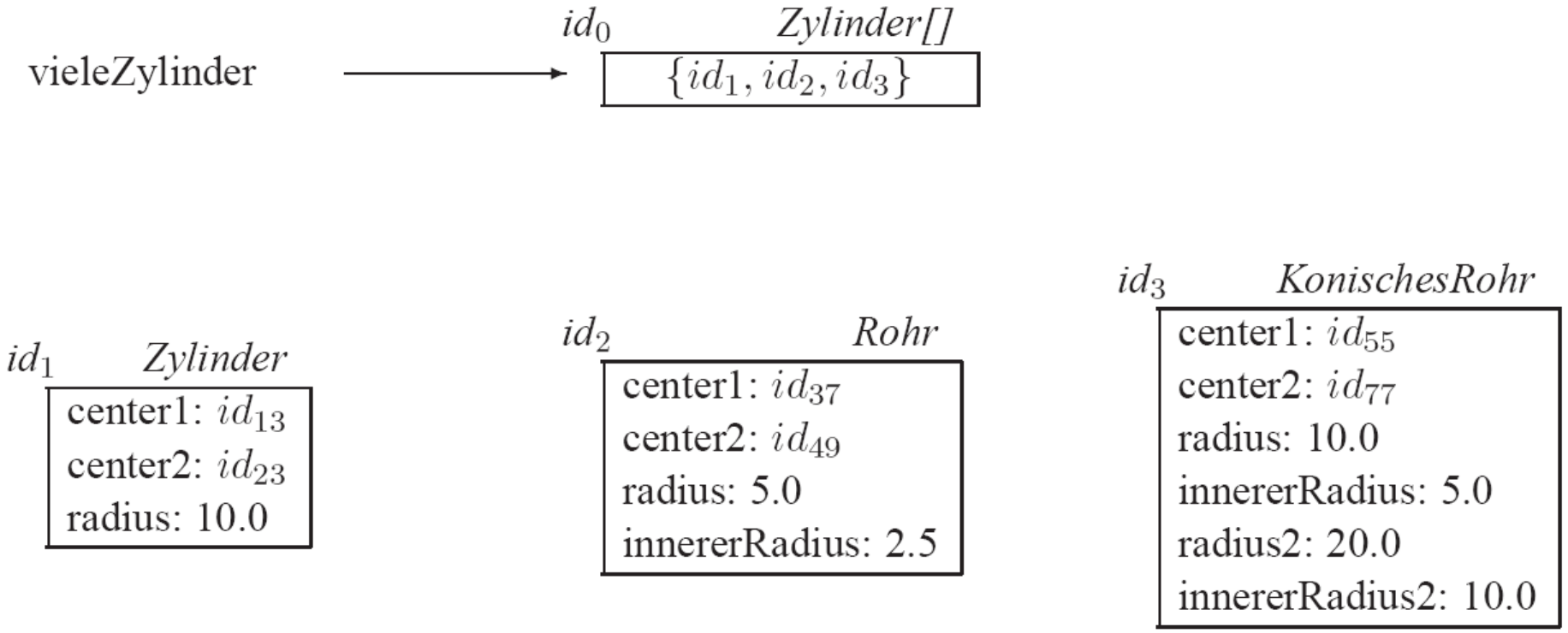
```
1  public class Rohr extends Zylinder {
2      public double innererRadius;
3
4      public Rohr(Vertex c1, Vertex c2, double r, double ir) {
5          super(c1, c2, r);
6          this.innererRadius = ir;
7      }
8
9      public double hohlKoerperVolumen() {
10         return this.innererRadius * this.innererRadius * Math.PI *
11                this.laenge();
12     }
13
14     public double volumen() {
15         return super.volumen() - this.hohlKoerperVolumen();
16     }
17
18     public String toString() {
19         return super.toString() + "␣Hohlkörper-Volumen:␣" +
20                this.hohlKoerperVolumen();
21     }
22
23     public static void main(String args[]) {
24         Rohr p = new Rohr(new Vertex(1,2,3), new Vertex(2,3,4), 7, 6);
25         System.out.println(p);
26     }
27 }
```

Verfeinerung / refinement

```
1  public class KonischesRohr extends Rohr {
2      public double innererRadius2;
3      public double radius2;
4
5      public KonischesRohr(Vertex c1, Vertex c2, double r,
6                           double ir, double r2, double ir2) {
7          super(c1, c2, r, ir);
8          radius2 = r2;
9          innererRadius2 = ir2;
10     }
11
12     public double hohlKoerperVolumen() {
13         return ((Math.PI * this.laenge() / 3) *
14                 (Math.pow(this.innererRadius, 2) +
15                  this.innererRadius * this.innererRadius2 +
16                  Math.pow(this.innererRadius2, 2)));
17     }
18
19     public double volumen() {  // Math.pow(x,n) berechnet x hoch n
20         return (((Math.PI * this.laenge() / 3) *
21                  (Math.pow(this.radius, 2) + this.radius * this.radius2 +
22                   Math.pow(this.radius2, 2))) - this.hohlKoerperVolumen());
23     }
24
25     public String toString() {
26         return super.toString() + "␣Hohlkörper-Volumen:␣" +
27                this.hohlKoerperVolumen();
28     }
29
30     public static void main(String args[]) {
31         KonischesRohr p = new KonischesRohr(new Vertex(1,2,3),
32                                             new Vertex(2,3,4), 7, 6, 5, 4);
33         System.out.println(p);
34     }
35 }
```

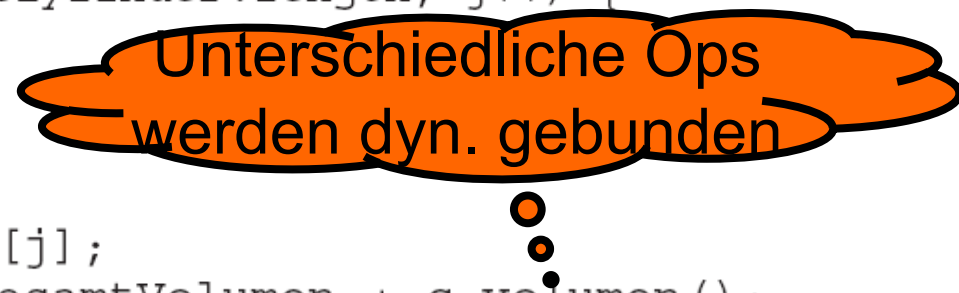
Verfeinerung / refinement nochmals

Dynamisches Binden



- Das Objekt mit der OID id_1 ist vom Typ *Zylinder*
- Das Objekt mit der OID id_2 ist vom Typ *Rohr*
- Das Objekt mit der OID id_3 ist vom Typ *KonischesRohr*

```
1  class DynBindingTest {
2      public static void main(String args[]) {
3          Zylinder cyl = new Zylinder(new Vertex(1,2,3),
4                                      new Vertex(2,3,4), 10.0);
5          Rohr p = new Rohr(new Vertex(1,2,3),
6                            new Vertex(2,3,4), 5.0, 2.5);
7          KonischesRohr cp = new KonischesRohr(new Vertex(1,2,3),
8                                               new Vertex(2,3,4),
9                                               10.0, 5.0, 20.0, 10.0);
10         Zylinder[] vieleZylinder = {cyl,p,cp};
11         double gesamtVolumen = 0.0;
12         Zylinder c;
13         for (int j = 0; j < vieleZylinder.length; j++) {
14             c = vieleZylinder[j];
15             gesamtVolumen = gesamtVolumen + c.volumen();
16           }
17           System.out.println("Gesamtvolumen: " + gesamtVolumen);
18       }
19   }
```



Substituierbarkeit: Eine Untertyp-Instanz kann eine Obertyp-Instanz substituieren – nicht umgekehrt

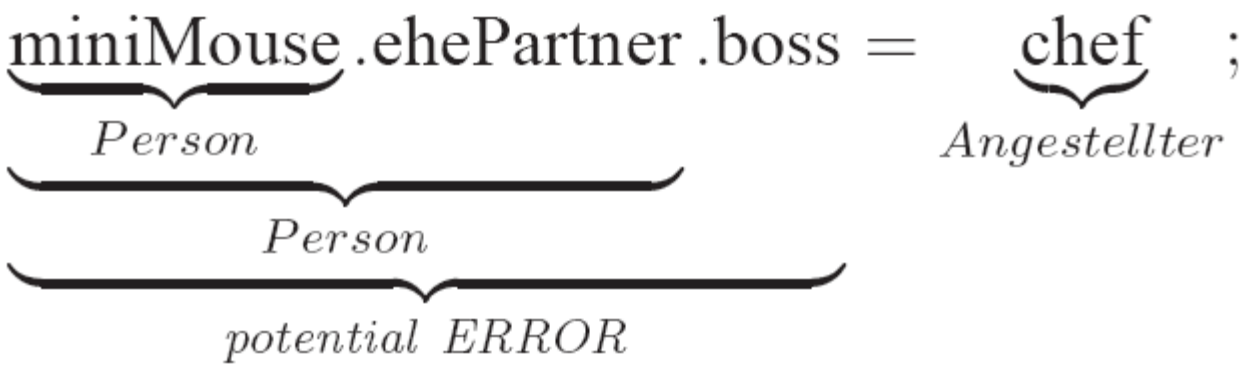
```
   Person einePerson;
   Angestellter einAngestellter;
   ...
(1)einePerson = einAngestellter;
(2)...
(3)einAngestellter = einePerson;          // nicht erlaubt!
```

$$\underbrace{\text{einAngestellter}}_{Angestellter} = \underbrace{\text{einePerson}}_{Person};$$

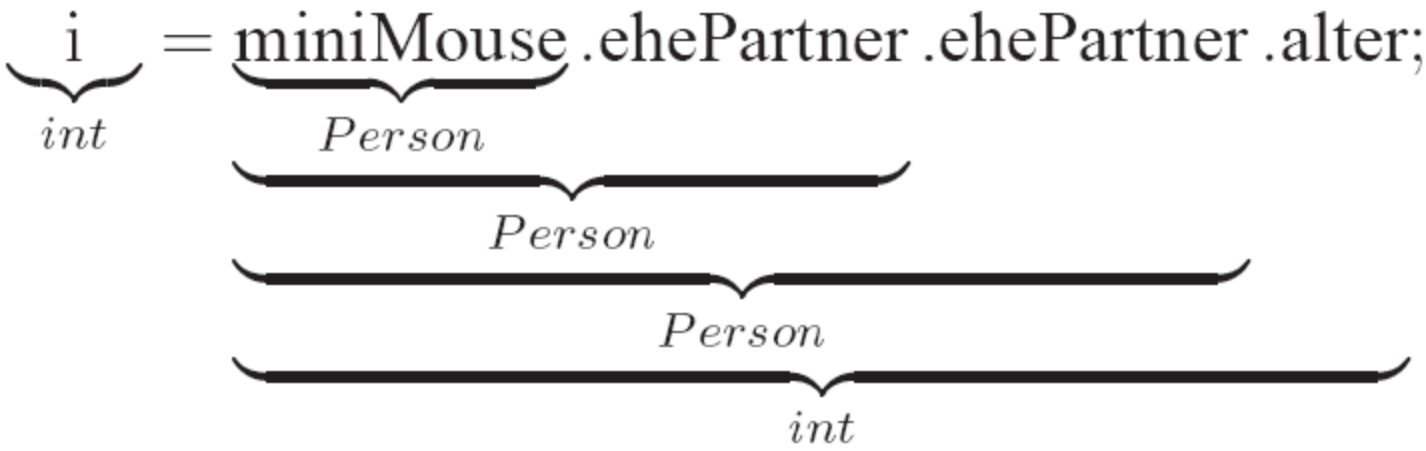
```
(1)einePerson.name;          // okay
(2)einePerson.gehalt;        // nicht erlaubt!
(3)einAngestellter.gehalt; // okay
```

Beispiele für die Typisierungsregeln

```
   Person miniMouse;
   Angestellter mickeyMouse;
   Angestellter chef;
   int i;
   ...
(1)mickeyMouse.ehePartner = miniMouse;          // okay
(2)miniMouse.ehePartner = mickeyMouse;          // okay
(3)mickeyMouse.boss = chef;                     // okay
(4)miniMouse.ehePartner.boss = chef;            // nicht erlaubt!
(5)i = mickeyMouse.boss.steuerNr;               // okay
(6)i = miniMouse.ehePartner.boss.steuerNr;      // nicht erlaubt!
(7)i = miniMouse.ehePartner.ehePartner.alter;   // okay
(8)i = mickeyMouse.ehePartner.boss.steuerNr;    // nicht erlaubt!
(9)mickeyMouse.boss.ehePartner.heiraten(chef);  // okay
```



Weiteres Beispiel (7)



Abstrakte/Virtuelle Klassen & Schnittstellen/Interfaces

- Deklaration von Methoden ohne deren Implementierung schon durchzuführen
  - Interface:
    - nur Deklarationen
    - Eine Klasse kann mehrere Schnittstellen implementieren
  - Abstrakte Klasse:
    - Einige Methoden können schon implementiert werden
    - Andere nur deklariert
    - Virtuelle Klasse kann man nicht instanziieren

Abstrakte Klasse: GeoPrimitive

```
1  public abstract class GeoPrimitive {
2      public int geoID;
3      public Material mat;
4      public String farbe;
5
6      public double spezGewicht() {
7          return mat.spezGewicht;
8      }
9
10     public abstract String toString();
11
12     public abstract double volumen();
13
14     public double gewicht() {
15         return this.volumen() * this.spezGewicht();
16     }
17
18 } // end abstract class GeoPrimitive
```

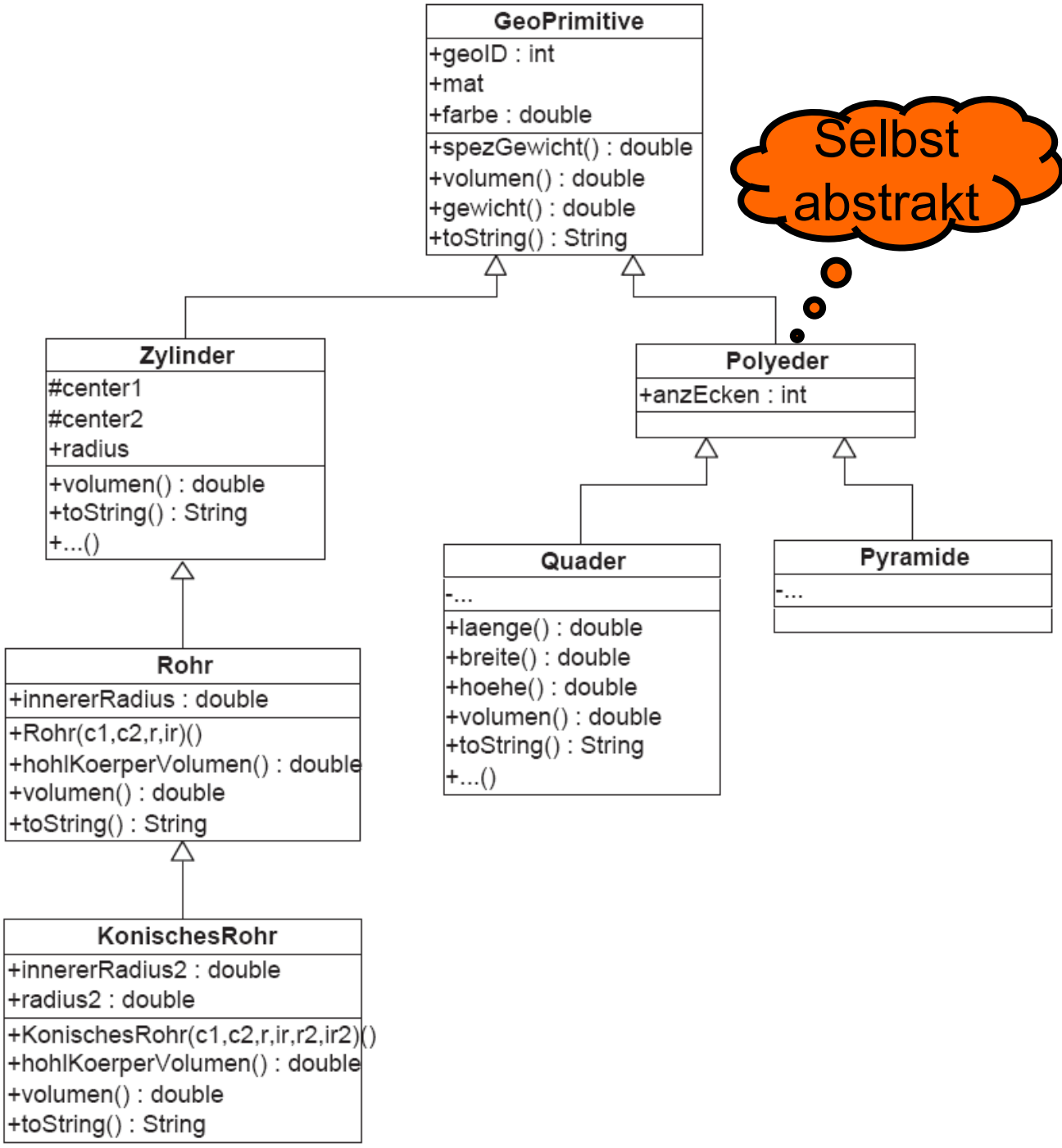
Erweiterung der abstrakten Klasse

```
class Zylinder extends GeoPrimitive {
    public double radius;
    protected Vertex center2;

    public Zylinder(Vertex c1, Vertex c2, double r) {
        center1 = c1;
        center2 = c2;
        radius = r;
    }

    public double laenge() {
        return center1.distanz(center2);
    }

    public double volumen() {
        return this.radius * this.radius * Math.PI * this.laenge();
    }

    public String toString() {
        return "Zylinder mit dem  Radius: " + this.radius +
               " Center1:" + this.center1 + " Center2: " +
               this.center2 + " Länge: " + this.laenge() +
               " Volumen: " + this.volumen();
    }
}
```

Nutzung der abstrakten Klasse

```
GeoPrimitive b;
double gesamtGewicht = 0.0;
double gesamtVolumen = 0.0;
for (int i = 0; i < basisTeile.length; i++) {
    b = basisTeile[i];
    gesamtGewicht = gesamtGewicht + b.gewicht();
    gesamtVolumen = gesamtVolumen + b.volumen();
}
```

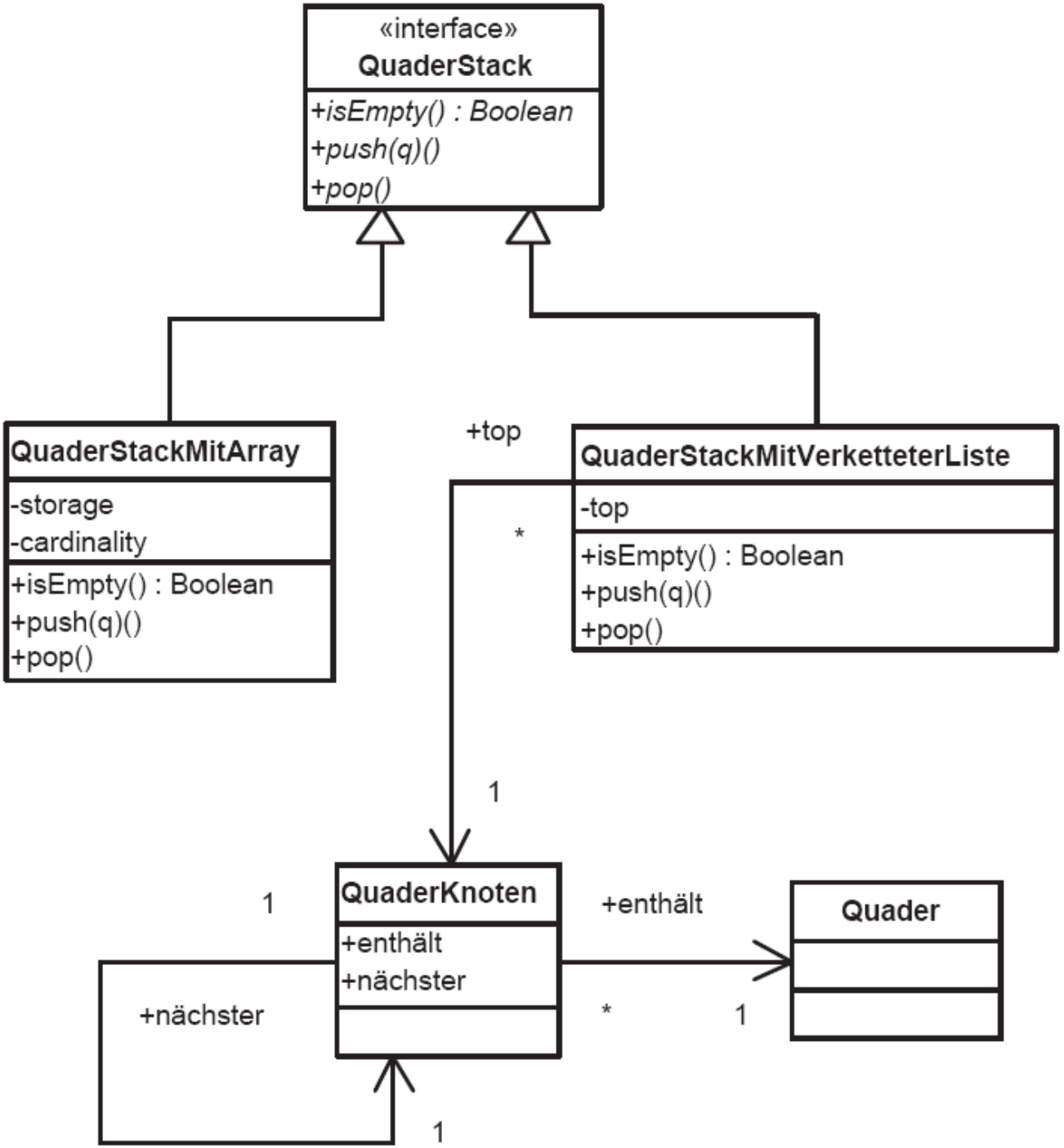
Hierarchie mit abstrakten Klassen



```
public class Quader extends Polyeder{
    private Vertex v1, v2, v3, v4, v5, v6, v7, v8;

    public Quader(Material m) {  // Konstruktor: Einheitswürfel
        this.v1 = new Vertex(0.0,0.0,0.0);
        this.v2 = new Vertex(1.0,0.0,0.0);
        this.v3 = new Vertex(1.0,1.0,0.0);
        this.v4 = new Vertex(0.0,1.0,0.0);
        this.v5 = new Vertex(0.0,0.0,1.0);
        this.v6 = new Vertex(1.0,0.0,1.0);
        this.v7 = new Vertex(1.0,1.0,1.0);
        this.v8 = new Vertex(0.0,1.0,1.0);
        this.mat = m; this.anzahlEcken = 8;
    }

    public double laenge() {
        return this.v1.distanz(this.v5);
    }
    public double breite() {
        return this.v1.distanz(this.v2);
    }
    public double hoehe() {
        return this.v1.distanz(this.v4);
    }
    public double volumen() {
        return this.laenge() * this.hoehe() * this.breite();
    }
    public double gewicht() {
        return this.volumen() * this.mat.spezGewicht;
    }

    public String toString() {
       return("Quader der Dimension " + this.laenge() + " X "
              + this.breite() + " X " + this.hoehe());
    }
```

Schnittstellen / Interfaces

```
interface QuaderStack {
    public boolean isEmpty();
    public void push(Quader c);
    public Quader pop();
}

class QuaderStackMitArray implements QuaderStack {
    private Quader[] storage;
    private int cardinality;
    ...
}

class QuaderStackMitVerketteteterListe implements QuaderStack {
    private QuaderKnoten top;  // class QuaderKnoten muss
    private int cardinality;   // als "Container" für einen Quader
        ...                    // und einem Attr. QuaderKnoten definiert sein
}
...
QuaderStack turmLinks, turmMitte, turmRechts;
turmLinks = new QuaderStackMitArray(...);
turmMitte = new QuaderStackMitVerketteteterListe(...);
turmRechts = new QuaderStackMitArray(...);
...
turmLinks.push(new Quader(...));
turmMitte.push(turmLinks.pop());
...
```


«interface»
QuaderStack
+isEmpty() : Boolean
+push(q)()
+pop()
QuaderStackMitArray
-storage
-cardinality
+isEmpty() : Boolean
+push(q)()
+pop()
+top
*
QuaderStackMitVerketteterListe
-top
+isEmpty() : Boolean
+push(q)()
+pop()
1
1
QuaderKnoten
+enthält
+nächster
+enthält
*
1
Quader
+nächster
1

Typ-Anfragen und Type-Casting

```
Object preistraeger;    // sollte eine Person sein ... but who knows
Car jaguar = new Car("Jaguar",340,250);  // 340 PS, 250 km/h
double notenSchnittBonus = 0.9;   // default notenSchnitt-bonus
double gehaltBonus = 1.1;  // default gehalt-bonus
   // ...
        if (preistraeger instanceof Manager) {
            Manager m = (Manager)preistraeger;
            m.dienstWagen = jaguar;
            System.out.println("new car");
        }
        else if (preistraeger instanceof Angestellter) {
            Angestellter e = (Angestellter)preistraeger;
            e.gehalt = e.gehalt * gehaltBonus;
            System.out.println("Gehalt erhöht.");
        }
        else if (preistraeger instanceof Student) {
            Student s = (Student)preistraeger;
            s.notenSchnitt = s.notenSchnitt * notenSchnittBonus;
            System.out.println("NOTENSCHNITT upgrade.");
        }
        else if (preistraeger instanceof  Person)
            System.out.println("Großartig -- Danke!");
        else
            System.out.println("Deinen Typ kenne ich nicht!");
    }
```